lakesuperior Documentation

Release 1.0.0a22

Stefano Cossu

Aug 16, 2019

Getting Started

Lakesuperior is an alternative Fedora Repository implementation.

Fedora is a mature repository software system historically adopted by major cultural heritage institutions. It exposes an LDP endpoint to manage any type of binary files and their metadata in Linked Data format.

CHAPTER 1

Indices and tables

1.1 About Lakesuperior

Lakesuperior is a repository system that stores binary files and their metadata as Linked Data. It is a Fedora Repository implementation focused on efficiency, stability and integration with Python.

Fedora is a mature repository software system historically adopted by major cultural heritage institutions. It exposes an LDP endpoint to manage any type of binary files and their metadata in Linked Data format.

1.1.1 Guiding Principles

Lakesuperior aims at being an efficient and flexible Fedora 4 implementation.

Its main goals are:

- **Reliability:** Based on solid technologies with stability in mind.
- **Efficiency:** Small memory and CPU footprint, high scalability.
- **Ease of management:** Tools to perform monitoring and maintenance included.
- **Simplicity of design:** Straight-forward architecture, robustness over features.

1.1.2 Key features

- Drop-in replacement for Fedora4 (with some *caveats*)
- Very stable persistence layer based on LMDB and filesystem. Fully ACID-compliant writes guarantee consistency of data.
- Term-based search and SPARQL Query API + UI
- No performance penalty for storing many resources under the same container; no kudzu pairtree segmentation.
- Extensible *provenance metadata* tracking

- *Multi-modal access*: HTTP (REST), command line interface and native Python API.
- Fits in a pocket: you can carry 64M triples in a 32Gb memory stick[1].

Implementation of the official Fedora API specs and OCFL are currently being considered as the next major development steps.

Please make sure you read the *Delta document* for divergences with the official Fedora4 implementation.

1.1.3 Target Audience

Lakesuperior is for anybody who cares about preserving data in the long term.

Less vaguely, Lakesuperior is targeted at who needs to store large quantities of highly linked metadata and documents.

Its Python/C environment and API make it particularly well suited for academic and scientific environments who would be able to embed it in a Python application as a library or extend it via plug-ins.

Lakesuperior is able to be exposed to the Web as a Linked Data Platform server. It also acts as a SPARQL query (read-only) endpoint, however it is not meant to be used as a full-fledged triplestore at the moment.

In its current status, Lakesuperior is aimed at developers and hands-on managers who are interested in evaluating this project.

1.1.4 Status and development

Lakesuperior is in **alpha** status. Please see the project issues list for a rudimentary road map.

1.1.5 Acknowledgements & Caveat

Most of this code has been written on the Chicago CTA Blue Line train and, more recently, on the Los Angeles Metro 734 bus. The author would like to thank these companies for providing an office on wheels for this project.

Potholes on Sepulveda street may have caused bugs and incorrect documentation. Please report them if you find any.

1.2 Installation & Configuration

1.2.1 Quick Install: Running in Docker

You can run Lakesuperior in Docker for a hands-off quickstart.

Docker is a containerization platform that allows you to run services in lightweight virtual machine environments without having to worry about installing all of the prerequisites on your host machine.

1. Install the correct Docker Community Edition for your operating system.
2. Clone the Lakesuperior git repository: `git clone --recurse-submodules https://github.com/scossu/lakesuperior.git`
3. `cd` into repo folder
4. Run `docker-compose up`

[1] Your mileage may vary depending on the variety of your triples.

Lakesuperior should now be available at `http://localhost:8000/`.

The provided Docker configuration includes persistent storage as a self-container Docker volume, meaning your data will persist between runs. If you want to clear the decks, simply run `docker-compose down -v`.

1.2.2 Manual Install (a bit less quick, a bit more power)

Note: These instructions have been tested on Linux. They may work on Darwin with little modification, and possibly on Windows with some modifications. Feedback is welcome.

Dependencies

1. Python 3.6 or greater.
2. A message broker supporting the STOMP protocol. For testing and evaluation purposes, CoilMQ is included with the dependencies and should be automatically installed.

Installation steps

Start in an empty project folder. If you are feeling lazy you can copy and paste the lines below in your console.

```
python3 -m venv venv
source venv/bin/activate
pip install lakesuperior
# Start the message broker. If you have another
# queue manager listening to port 61613 you can either configure a
# different port on the application configuration, or use the existing
# message queue.
coilmq&
# Bootstrap the repo
lsup-admin bootstrap # Confirm manually
# Run the thing
fcrepo
```

Test if it works:

```
curl http://localhost:8000/ldp/
```

1.2.3 Advanced Install

A "developer mode" install is detailed in the *Development Setup* section.

1.2.4 Configuration

The app should run for testing and evaluation purposes without any further configuration. All the application data are stored by default in the `data` directory of the Python package.

This setup is not recommended for anything more than a quick look at the application. If more complex interaction is needed, or upgrades to the package are foreseen, it is strongly advised to set up proper locations for configuration and data.

To change the default configuration you need to:

1. Copy the `etc.default` folder to a separate location
2. Set the configuration folder location in the environment: `export FCREPO_CONFIG_DIR=<your config dir location>` (you can add this line at the end of your virtualenv `activate` script)
3. Configure the application
4. Bootstrap the app or copy the original data folders to the new location if any loction options changed
5. (Re)start the server: `fcrepo`

The configuration options are documented in the files.

One thing worth noting is that some locations can be specified as relative paths. These paths will be relative to the `data_dir` location specified in the `application.yml` file.

If `data_dir` is empty, as it is in the default configuration, it defaults to the `data` directory inside the Python package. This is the option that one may want to change before anything else.

1.2.5 Production deployment

If you like fried repositories for lunch, deploy before 11AM.

1.3 Sample Usage

1.3.1 LDP (REST) API

The following are very basic examples of LDP interaction. For a more complete reference, please consult the Fedora API guide.

Note: At the moment the LDP API only support the Turtle format for serializing and deserializing RDF.

Create an empty LDP container (LDPC)

```
curl -X POST http://localhost:8000/ldp
```

Create a resource with RDF payload

```
curl -X POST -H'Content-Type:text/turtle' --data-binary '<> <urn:ns:p1> <urn:ns:o1> .
↪' http://localhost:8000/ldp
```

Create a resource at a specific location

```
curl -X PUT http://localhost:8000/ldp/res1
```

Create a binary resource

```
curl -X PUT -H'Content-Type:image/png' --data-binary '@/home/me/image.png' http://
↪localhost:8000/ldp/bin1
```

Retrieve an RDF resource (LDP-RS)

```
curl http://localhost:8000/ldp/res1
```

Retrieve a non-RDF source (LDP-NR)

```
curl http://localhost:8000/ldp/bin1
```

Or:

```
curl http://localhost:8000/ldp/bin1/fcr:content
```

Or:

```
curl -H'Accept:image/png' http://localhost:8000/ldp/bin1
```

Retrieve RDF metadata of a LDP-NR

```
curl http://localhost:8000/ldp/bin1/fcr:metadata
```

Or:

```
curl -H'Accept:text/turtle' http://localhost:8000/ldp/bin1
```

Soft-delete a resource

```
curl -X DELETE http://localhost:8000/ldp/bin1
```

Restore ("resurrect") a resource

```
curl -X POST http://localhost:8000/ldp/bin1/fcr:tombstone
```

Permanently delete ("forget") a soft-deleted resource

Note: the following command cannot be issued after the previous one. It has to be issued on a soft-deleted, non-resurrected resource.

```
curl -X DELETE http://localhost:8000/ldp/bin1/fcr:tombstone
```

Immediately forget a resource

```
curl -X DELETE -H'Prefer:no-tombstone' http://localhost:8000/ldp/res1
```

1.3.2 Admin REST API

Fixity check

Check the fixity of a resource, i.e. if the checksum stored in the metadata corresponds to the current checksum of the stored file. This requires a checksum calculation and may take a long time depending on the file size and the hashing algorithm chosen:

```
curl http://localhost:8000/admin/<resource UID>/fixity
```

The response is a JSON document with two keys: `uid` indicating the UID of the resource checked; and `pass` that can be True or False depending on the outcome of the check.

1.3.3 Python API

Set up the environment

Before using the API, either do:

```
>>> import lakesuperior.env_setup
```

Or, to specify an alternative configuration:

```
>>> from lakesuperior import env
>>> from lakesuperior.config_parser import parse_config
>>> from lakesuperior.globals import AppGlobals
>>> config = parse_config('/my/custom/config_dir')
Reading configuration at /my/custom/config_dir
>>> env.app_globals = AppGlobals(config)
```

Create and replace resources

Create an LDP-RS (RDF reseouce) providng a Graph object:

```
>>> from rdflib import Graph, URIRef
>>> uid = '/rsrc_from_graph'
>>> gr = Graph().parse(data='<> a <http://ex.org/type#A> .',
...     format='text/turtle', publicID=nsc['fcres'][uid])
>>> rsrc_api.create_or_replace(uid, init_gr=gr)
```

Issuing a `create_or_replace()` on an existing UID will replace the existing property set with the provided one (PUT style).

Create an LDP-NR (non-RDF source):

```
>>> uid = '/test_ldpnr01'
>>> data = b'Hello. This is some dummy content.'
>>> rsrc_api.create_or_replace(
...     uid, stream=BytesIO(data), mimetype='text/plain')
'_create_'
```

Create or replace providing a serialized RDF byte stream:

```
>>> uid = '/rsrc_from_rdf'
>>> rdf = b'<#a1> a <http://ex.org/type#B> .'
>>> rsrc_api.create_or_replace(uid, rdf_data=rdf, rdf_fmt='turtle')
```

Relative URIs such as `<#a1>` will be resolved relative to the resource URI.

Create under a known parent, providing a slug (POST style):

```
>>> rsrc_api.create('/rsrc_from_stream', 'res1')
```

This will create `/rsrc_from_stream/res1` if not existing; otherwise the resource URI will have a random UUID4 instead of `res1`.

To use a random UUID by default, use `None` for the second argument.

Retrieve Resources

Retrieve a resource:

```
>>> rsrc = rsrc_api.get('/rsrc_from_stream')
>>> rsrc.uid
'/rsrc_from_stream'
>>> rsrc.uri
rdflib.term.URIRef('info:fcres/rsrc_from_stream')
>>> set(rsrc.metadata)
{(rdflib.term.URIRef('info:fcres/rsrc_from_stream'),
  rdflib.term.URIRef('http://fedora.info/definitions/v4/repository#created'),
  rdflib.term.Literal('2018-04-06T03:30:49.460274+00:00', datatype=rdflib.term.URIRef(
↪'http://www.w3.org/2001/XMLSchema#dateTime'))),
[...]
```

Retrieve non-RDF content:

```
>>> ldpnr = rsrc_api.get('/test_ldpnr01')
>>> ldpnr.content.read()
b'Hello. This is some dummy content.'
```

See the *API docs* for more details on resource methods.

Update Resources

Using a SPARQL update string:

```
>>> uid = '/test_delta_patch_wc'
>>> uri = nsc['fcres'][uid]
>>> init_trp = {
...     (URIRef(uri), nsc['rdf'].type, nsc['foaf'].Person),
...     (URIRef(uri), nsc['foaf'].name, Literal('Joe Bob')),
...     (URIRef(uri), nsc['foaf'].name, Literal('Joe Average Bob')),
... }

>>> update_str = '''
... DELETE {}
... INSERT { <> foaf:name "Joe Average 12oz Bob" . }
... WHERE {}
... '''
```

Using add/remove triple sets:

```
>>> remove_trp = {
...     (URIRef(uri), nsc['foaf'].name, None),
... }
>>> add_trp = {
...     (URIRef(uri), nsc['foaf'].name, Literal('Joan Knob')),
... }

>>> gr = Graph()
>>> gr += init_trp
>>> rsrc_api.create_or_replace(uid, graph=gr)
>>> rsrc_api.update_delta(uid, remove_trp, add_trp)
```

Note above that wildcards can be used, only in the remove triple set. Wherever `None` is used, all matches will be removed (in this example, all values of `foaf:name`.

Generally speaking, the delta approach providing a set of remove triples and/or a set of add triples is more convenient than SPARQL, which is a better fit for complex query/update scenarios.

1.4 Getting Help

Discussion is on the lakesuperior Google group.

You can report bugs or feature requests on the Github issues page. Please start a conversation in the Google group before filing an issue, especially for feature requests.

1.5 Application Configuration Reference

app_mode

> Application mode.
>
> One of `prod`, `test` or `dev`. `prod` is normal running mode. 'test' is used for running test suites. `dev` is similar to normal mode but with reload and debug enabled.

data_dir

> Base data directory.
>
> This contains both volatile files such as PID files, and persistent ones, such as resource data. LDP-NRs will be stored under `<basedir>/ldpnr_store` and LDP-RSs under ``<basedir>/ldprs_store`.
>
> If different data files need to be running on different storage hardware, the individual subdirectories can be mounted on different file systems.
>
> If unset, it will default to <lakesuperior package root>/data.

uuid Configuration for binary path and fixity check generation. The hash is a checksumn of the contents of the file.

algo

> Algorithm used to calculate the hash that generates the content path.
>
> This can be any one of the Python hashlib functions: https://docs.python.org/3/library/hashlib.html

This needs to be `sha1` if a compatibility with the Fedora4 file layout is needed, however in security-sensitive environments it is strongly advised to use a stronger algorithm, since SHA1 is known to be vulnerable to counterfeiting: see https://shattered.io/

`blake2b` is a strong, fast cryptographic alternative to SHA2/3: https://blake2.net/

store Data store configuration.

ldp_rs

The semantic store used for persisting LDP-RS (RDF Source) resources.

MUST support SPARQL 1.1 query and update.

layout

Store layout.

At the moment, only `rsrc_centric_layout` is supported.

referential_integrity

Enable referential integrity checks.

Whether to check if the object of a client-provided triple is the uri of a repository-managed resource and veify if that exists. if set to false, properties are allowed to point to resources in the repositoy that do not exist. also, if a resource is deleted, inbound relationships may not be cleaned up. this can be one of `False`, `lenient` or `strict`. `False` does not check for referential integrity. `lenient` quietly drops a user-provided triple if its object violates referential integrity. `strict` raises an exception.

Changes to this parameter require a full migration.

ldp_nr

The path used to persist LDP-NR (bitstreams).

This is for now a POSIX filesystem. Other solutions such as HDFS may be possible in the future.

layout

See store.ldp_rs.layout.

pairtree_branch_length

How to split the balanced pairtree to generate a path.

The hash string is defined by the uuid.algo parameter value. This parameter defines how many characters are in each branch. 2-4 is the recommended setting. NOTE: a value of 2 will generate up to 256 sub-folders in a folder; 3 will generate max. 4096 and 4 will generate max. 65536. Check your filesystem capabilities before setting this to a non-default value.

Changes to this parameter require a full migration.

pairtree_branches

Max. number of branches to generate.

0 will split the string until it reaches the end.

E.g. if the hash value is 0123456789abcdef01234565789abcdef and the branch length value is 2, and the branch number is 4, the path will be 01/23/45/67/89abcdef01234565789abcdef. For a value of 0 it will be 01/23/45/67/89/ab/cd/ef/01/23/45/67/89/ab/cd/ef. Be aware that deeply nested directory structures may tax some of the operating system's services that scan for files, such as *updatedb*. Check your system capabilities for maximum nested directories before changing the default.

Changes to this parameter require a full migration.

messaging

Messaging configuration.

routes

List of channels to send messages to.

Each channel must define the *endpoint* and the *level* parameters.

handler

Output handler. Currently only `StompHandler` is supported.

active

Activate this route.

If `False`, no messages will be emitted for this route.

protocol

Protocol version. One of `10`, `11` or `12`.

host

Host IP address.

port

Host port.

username

User name for authentication.

Credentials are optional.

password

Password for authentication.

destination

Message topic.

formatter

Message format: at the moment the following are supported:

- `ASResourceFormatter`: Sends information about a resource being created, updated or deleted, by who and when, with no further information about what changed.
- `ASDeltaFormatter`: Sends the same information as `ASResourceFormatter` with the addition of the triples that were added and the ones that were removed in the request. This may be used to send rich provenance data to a preservation system.

1.6 Resource Discovery & Query

Lakesuperior offers several way to programmatically discover resources and data.

1.6.1 LDP Traversal

The method compatible with the standard Fedora implementation and other LDP servers is to simply traverse the LDP tree. While this offers the broadest compatibility, it is quite expensive for the client, the server and the developer.

For this method, please consult the dedicated LDP specifications and Fedora API specs.

1.6.2 SPARQL Query

A SPARQL endpoint is available in Lakesuperior both as an API and a Web UI.

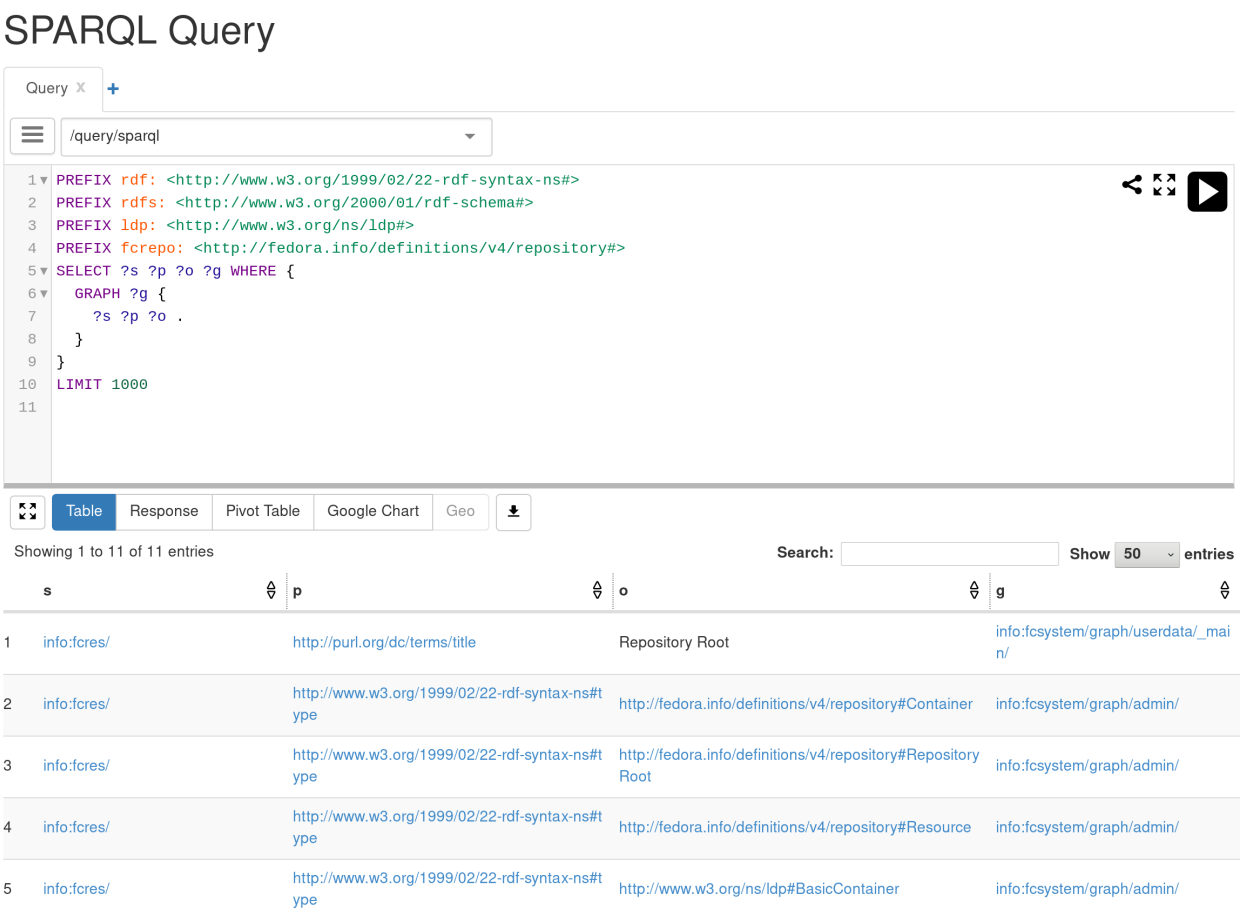

Fig. 1: Lakesuperior SPARQL Query Window

The UI is based on YASGUI.

Note that:

1. The SPARQL endpoint only supports the SPARQL 1.1 Query language. SPARQL updates are not, and will not be, supported.
2. The LAKEshore data model has an added layer of structure that is not exposed through the LDP layer. The SPARQL endpoint exposes this low-level structure and it is beneficial to understand its layout. See *Lakesuperior Content Model Rationale* for details in this regard.
3. The underlying RDF structure is mostly in the RDF named graphs. Querying only triples will give a quite uncluttered view of the data, as close to the LDP representation as possible.

SPARQL Caveats

The SPARQL query facility has not yet been tested thoroughly. the RDFLib implementation that it is based upon can be quite efficient for certain queries but has some downsides. For example, do **not** attempt the following query in a graph with more than a few thousands resources:

```
SELECT ?p ?o {
  GRAPH ?g {
    <info:fcres/my-uid> ?p ?o .
  }
}
```

What the RDFLib implementation does is going over every single graph in the repository and perform the `?s ?p ?o` query on each of them. Since Lakesuperior creates several graphs per resource, this can run for a very long time in any decently sized data set.

The solution to this is either to omit the graph query, or use a term search, or a native Python method if applicable.

1.6.3 Term Search

This feature provides a discovery tool focused on resource subjects and based on individual term match and comparison. It tends to be more manageable than SPARQL but also uses some SPARQL syntax for the terms.

Multiple search conditions can be entered and processed with AND or OR logic.

The obtained results are resource URIs relative to the endpoint.

Please consult the search page itself for detailed instructions on how to enter query terms.

The term search is also available via REST API. E.g.:

```
curl -i -XPOST http://localhost:8000/query/term_search -d '{"terms": [{"pred":
↪"rdf:type", "op": "_id", "val": "ldp:Container"}], "logic": "and"}' -H'Content-
↪Type:application/json'
```

1.7 Divergencies between lakesuperior and FCREPO4

This is a (vastly incomplete) list of discrepancies between the current FCREPO4 implementation and Lakesuperior. More will be added as more clients will use it.

1.7.1 Not yet implemented (but in the plans)

- Various headers handling (partial)
- AuthN and WebAC-based authZ
- Fixity check
- Blank nodes (at least partly working, but untested)
- Multiple byte ranges for the `Range` request header

1.7.2 Potentially breaking changes

The following divergences may lead into incompatibilities with some clients.

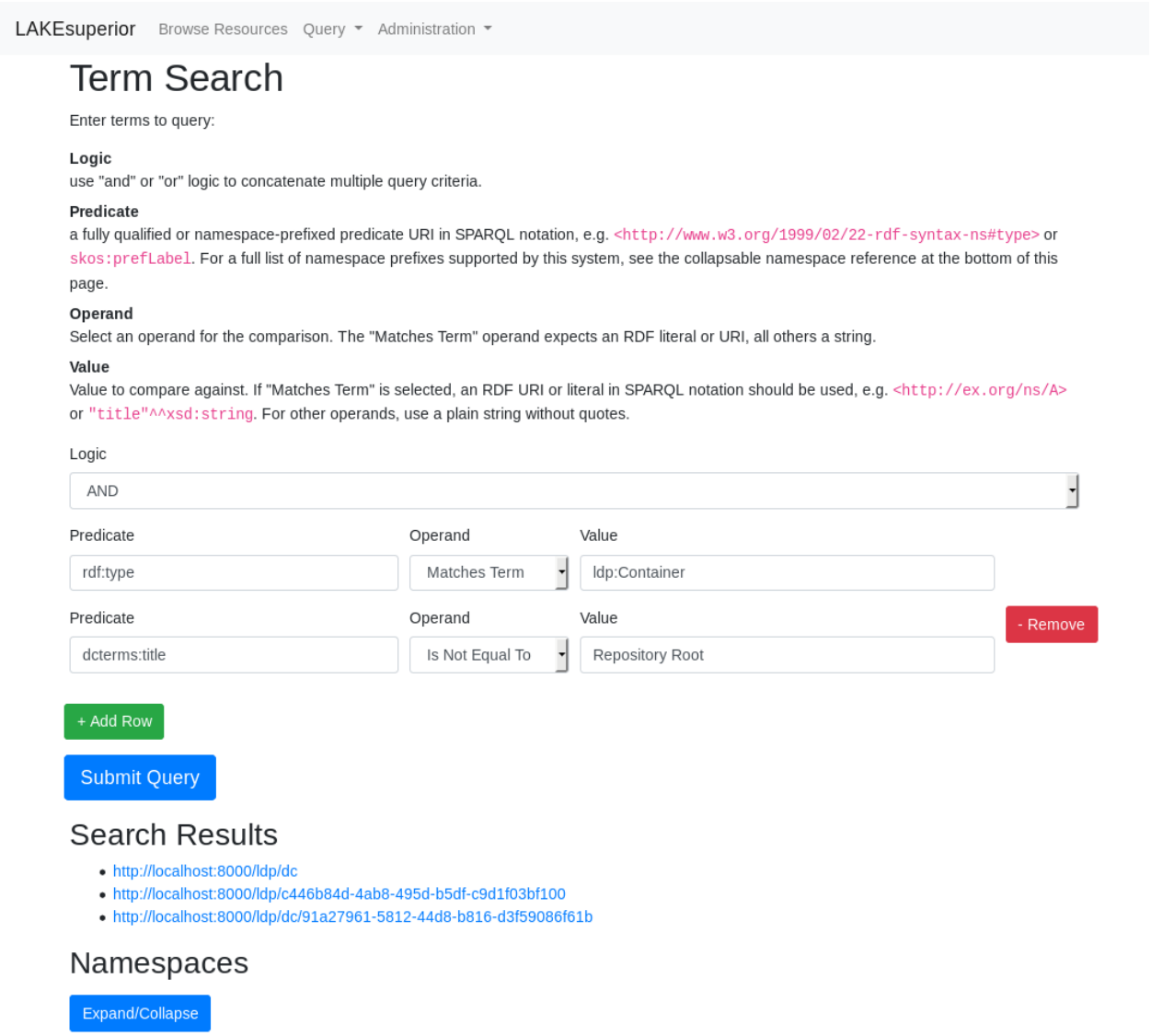

Fig. 2: Lakesuperior Term Search Window

ETags

"Weak" ETags for LDP-RSs (i.e. RDF graphs) are not implemented. Given the possible many interpretations of how any kind of checksum for an LDP resource should be calculated (see discussion), and also given the relatively high computation cost necessary to determine whether to send a `304 Not Modified` vs. a `200 OK` for an LDP-RS request, this feature has been considered impractical to implement with the limited resources available at the moment.

As a consequence, LDP-RS requests will never return a `304` and will never include an `ETag` header. Clients should not rely on that header for non-binary resources.

That said, calculating RDF chacksums is still an academically interesting topic and may be valuable for practical purposes such as metadata preservation.

Atomicity

FCREPO4 supports batch atomic operations whereas a transaction can be opened and a number of operations (i.e. multiple R/W requests to the repository) can be performed. The operations are persisted in the repository only if and when the transaction is committed.

LAKesuperior only supports atomicity for a single HTTP request. I.e. a single HTTTP request that should result in multiple write operations to the storage layer is only persisted if no exception is thrown. Otherwise, the operation is rolled back in order to prevent resources to be left in an inconsistent state.

Tombstone methods

If a client requests a tombstone resource in FCREPO4 with a method other than DELETE, the server will return `405 Method Not Allowed` regardless of whether the tombstone exists or not.

Lakesuperior will return `405` only if the tombstone actually exists, `404` otherwise.

`Limit` Header

Lakesuperior does not support the `Limit` header which in FCREPO can be used to limit the number of "child" resources displayed for a container graph. Since this seems to have a mostly cosmetic function in FCREPO to compensate for performance limitations (displaying a page with many thousands of children in the UI can take minutes), and since Lakesuperior already offers options in the `Prefer` header to not return any children, this option is not implemented.

Web UI

FCREPO4 includes a web UI for simple CRUD operations.

Such a UI is not in the immediate Lakesuperior development plans. However, a basic UI is available for read-only interaction: LDP resource browsing, SPARQL query and other search facilities, and administrative tools. Some of the latter *may* involve write operations, such as clean-up tasks.

Automatic path segment generation

A `POST` request without a slug in FCREPO4 results in a pairtree consisting of several intermediate nodes leading to the automatically minted identifier. E.g.

```
POST /rest
```

results in `/rest/8c/9a/07/4e/8c9a074e-dda3-5256-ea30-eec2dd4fcf61` being created.

The same request in Lakesuperior would create `/rest/8c9a074e-dda3-5256-ea30-eec2dd4fcf61` (obviously the identifiers will be different).

This seems to break Hyrax at some point, but might have been fixed. This needs to be verified further.

Allow PUT requests with empty body on existing resources

FCREPO4 returns a `409 Conflict` if a PUT request with no payload is sent to an existing resource.

Lakesuperior allows to perform this operation, which would result in deleting all the user-provided properties in that resource.

If the original resource is an LDP-NR, however, the operation will raise a `415 Unsupported Media Type` because the resource will be treated as an empty LDP-RS, which cannot replace an existing LDP-NR.

1.7.3 Non-standard client breaking changes

The following changes may be incompatible with clients relying on some FCREPO4 behavior not endorsed by LDP or other specifications.

Pairtrees

FCREPO4 generates "pairtree" resources if a resource is created in a path whose segments are missing. E.g. when creating `/a/b/c/d`, if `/a/b` and `/a/b/c` do not exist, FCREPO4 will create two Pairtree resources. POSTing and PUTting into Pairtrees is not allowed. Also, a containment triple is established between the closest LDPC and the created resource, e.g. if `a` exists, a `</a> ldp:contains </a/b/c/d>` triple is created.

Lakesuperior does not employ Pairtrees. In the example above Lakesuperior would create a fully qualified LDPC for each missing segment, which can be POSTed and PUT to. Containment triples are created between each link in the path, i.e. `</a> ldp:contains </a/b>`, `</a/b> ldp:contains </a/b/c>` etc. This may potentially break clients relying on the direct containment model.

The rationale behind this change is that Pairtrees are the byproduct of a limitation imposed by Modeshape and introduce complexity in the software stack and confusion for the client. Lakesuperior aligns with the more intuitive UNIX filesystem model, where each segment of a path is a "folder" or container (except for the leaf nodes that can be either folders or files). In any case, clients are discouraged from generating deep paths in Lakesuperior without a specific purpose because these resources create unnecessary data.

Non-mandatory, non-authoritative slug in version POST

FCREPO4 requires a `Slug` header to POST to `fcr:versions` to create a new version.

Lakesuperior adheres to the more general FCREPO POST rule and if no slug is provided, an automatic ID is generated instead. The ID is a UUID4.

Note that internally this ID is not called "label" but "uid" since it is treated as a fully qualified identifier. The `fcrepo:hasVersionLabel` predicate, however ambiguous in this context, will be kept until the adoption of Memento, which will change the retrieval mechanisms.

Another notable difference is that if a POST is issued on the same resource `fcr:versions` location using a version ID that already exists, Lakesuperior will just mint a random identifier rather than returning an error.

1.7.4 Deprecation track

Lakesuperior offers some "legacy" options to replicate the FCREPO4 behavior, however encourages new development to use a different approach for some types of interaction.

Endpoints

The FCREPO root endpoint is `/rest`. The Lakesuperior root endpoint is `/ldp`.

This should not pose a problem if a client does not have `rest` hard-coded in its code, but in any event, the `/rest` endpoint is provided for backwards compatibility.

Future implementations of the Fedora API specs may employ a "versioned" endpoint scheme that allows multiple Fedora API versions to be available to the client, e.g. `/ldp/fc4` for the current LDP API version, `/ldp/fc5` for Fedora version 5.x, etc.

Automatic LDP class assignment

Since Lakesuperior rejects client-provided server-managed triples, and since the LDP types are among them, the LDP container type is inferred from the provided properties: if the `ldp:hasMemberRelation` and `ldp:membershipResource` properties are provided, the resource is a Direct Container. If in addition to these the `ldp:insertedContentRelation` property is present, the resource is an Indirect Container. If any of the first two are missing, the resource is a Container.

Clients are encouraged to omit LDP types in PUT, POST and PATCH requests.

Lenient handling

FCREPO4 requires server-managed triples to be expressly indicated in a PUT request, unless the `Prefer` header is set to `handling=lenient; received="minimal"`, in which case the RDF payload must not have any server-managed triples.

Lakesuperior works under the assumption that client should never provide server-managed triples. It automatically handles PUT requests sent to existing resources by returning a 412 if any server managed triples are included in the payload. This is the same as setting `Prefer` to `handling=strict`, which is the default.

If `Prefer` is set to `handling=lenient`, all server-managed triples sent with the payload are ignored.

Clients using the `Prefer` header to control PUT behavior as advertised by the specs should not notice any difference.

1.7.5 Optional improvements

The following are improvements in performance or usability that can only be taken advantage of if client code is adjusted.

LDP-NR content and metadata

FCREPO4 relies on the `/fcr:metadata` identifier to retrieve RDF metadata about an LDP-NR. Lakesuperior supports this as a legacy option, but encourages the use of content negotiation to do the same while offering explicit endpoints for RDF and non-RDF content retrieval.

Any request to an LDP-NR with an `Accept` header set to one of the supported RDF serialization formats will yield the RDF metadata of the resource instead of the binary contents.

The `fcr:metadata` URI returns the RDF metadata of a LDP-NR.

The `fcr:content` URI returns the non-RDF content.

The two optionsabove return an HTTP error if requested for a LDP-RS.

"Include" and "Omit" options for children

Lakesuperior offers an additional `Prefer` header option to exclude all references to child resources (i.e. by removing all the `ldp:contains` triples) while leaving the other server-managed triples when retrieving a resource:

```
Prefer: return=representation; [include | omit]="http://fedora.info/definitions/v4/
↪repository#Children"
```

The default behavior is to include all children URIs.

Soft-delete and purge

NOTE: The implementation of this section is incomplete and debated.

In FCREPO4 a deleted resource leaves a tombstone deleting all traces of the previous resource.

In Lakesuperior, a normal DELETE creates a new version snapshot of the resource and puts a tombstone in its place. The resource versions are still available in the `fcr:versions` location. The resource can be "resurrected" by issuing a POST to its tombstone. This will result in a `201`.

If a tombstone is deleted, the resource and its versions are completely deleted (purged).

Moreover, setting the `Prefer:no-tombstone` header option on DELETE allows to delete a resource and its versions directly without leaving a tombstone.

1.8 Lakesuperior Messaging

Lakesuperior implements a messaging system based on ActivityStreams, as indicated by the Fedora API specs. The metadata set provided is currently quite minimal but can be easily enriched by extending the *`Messenger`* class.

STOMP is the only supported protocol at the moment. More protocols may be made available at a later time.

Lakesuperior can send messages to any number of destinations: see *Installation & Configuration*.

By default, CoilMQ is provided for testing purposes and listens to `localhost:61613`. The default route sends messages to `/topic/fcrepo`.

A small command-line utility, also provided with the Python dependencies, allows to watch incoming messages. To monitor messages, enter the following *after activating your virtualenv*:

```
stomp -H localhost -P 61613 -L /topic/fcrepo
```

See the stomp.py library reference page for details.

1.8.1 Disabing messaging

Messaging is enabled by default in Lakesuperior. If you are not interested in interacting with an integration framework, you can save yourself some I/O and complexity and turn messaging off completely. In order to do that, set all entries in the `routes` section of `application.yml` to not active, e.g.:

```
[...]
messaging:
    routes:
    - handler: StompHandler
      active: False # ← Disable the route
          protocol: '11'
          host: 127.0.0.1
          port: 61613
          username:
          password:
          destination: '/topic/fcrepo'
          formatter: ASResourceFormatter
```

A message queue does not need to be running in order for Lakesuperior to operate, even if one or more routes are active. In that case, the application will throw a few ugly mssages and move on. *TODO: This should be handled more gracefully.*

1.9 Migration, Backup & Restore

All Lakesuperior data is by default fully contained in a folder. This means that only the data, configurations and code folders are needed for it to run. No Postgres, Redis, or such. Data and configuration folders can be moved around as needed.

1.9.1 Migration Tool

Migration is the process of importing and converting data from a different Fedora or LDP implementation into a new Lakesuperior instance. This process uses the HTTP/LDP API of the original repository. A command-line utility is available as part of the `lsup-admin` suite to assist in such operation.

A repository can be migrated with a one-line command such as:

```
lsup-admin migrate http://source-repo.edu/rest /local/dest/folder
```

For more options, enter

```
lsup-admin migrate --help
```

The script will crawl through the resources and crawl through outbound links within them. In order to do this, resources are added as raw triples, i.e. no consistency checks are made.

This script will create a full dataset in the specified destination folder, complete with a default configuration that allows to start the Lakesuperior server immediately after the migration is complete.

Two approaches to migration are possible:

1. By providing a starting point on the source repository. E.g. if the repository you want to migrate is at `http://repo.edu/rest/prod` you can add the `-s /prod` option to the script to avoid migrating irrelevant branches. Note that the script will still reach outside of the starting point if resources are referencing other resources outside of it.
2. By providing a file containing a list of resources to migrate. This is useful if a source repository cannot produce a full list (e.g. the root node has more children than the server can handle) but a list of individual resources is available via an external index (Solr, triplestore, etc.). The resources can be indicated by their fully qualified URIs or paths relative to the repository root. (*TODO latter option needs testing*)

Consistency check can (and should) be run after the migration:

```
lsup-admin check_refint
```

This is critical to ensure that all resources in the repository are referencing to other repository resources that are actually existing.

This feature has been added in alpha9.

TODO: The output of ''check_refint'' is somewhat crude. Improvements can be made to output integrity violations to a machine-readable log and integrate with the migration tool.

1.9.2 Backup And Restore

A back up of a LAKEshore repository consists in copying the RDF and non-RDF data folders. These folders are indicated in the application configuration. The default commands provided by your OS (`cp`, `rsync`, `tar` etc. for Unix) are all is needed.

1.10 Command Line Reference

Lakesuperior comes with some command-line tools aimed at several purposes.

If Lakesuperior is installed via `pip`, all tools can be invoked as normal commands (i.e. they are in the virtualenv `PATH`).

The tools are currently not directly available on Docker instances (*TODO add instructions and/or code changes to access them*).

1.10.1 `lsup-server`

Single-threaded server. Use this for testing, debugging, or to multiplex via WSGI in a Windows environment. For non-Windows production environments, use `fcrepo`.

1.10.2 `fcrepo`

This is the main server command. It has no parameters. The command spawns Gunicorn workers (as many as set up in the configuration) and can be sent in the background, or started via init script.

The tool must be run in the same virtual environment Lakesuperior was installed in (if it was)—i.e.:

```
source <virtualenv root>/bin/activate
```

must be run before running the server.

Note that if `app_mode` is set to `prod` in *application.yml*, the server will just print the configuration and immediately go in the background without logging anything on screen (daemon mode).

In the case an init script is used, `coilmq` (belonging to a 3rd party package) needs to be launched as well; unless a message broker is already set up, or if messaging is disabled in the configuration.

Note: This command is not available in Windows because GUnicorn is not available in Windows. Windows users should look for alternative WSGI servers to run the single-threaded service (`lsup-server`) over multiple processes and/or threads.

1.10.3 lsup-admin

`lsup-admin` is the principal repository management tool. It is self-documented, so this is just a redundant overview:

```
$ lsup-admin
Usage: lsup-admin [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  bootstrap     Bootstrap binary and graph stores.
  check_fixity  [STUB] Check fixity of a resource.
  check_refint  Check referential integrity.
  cleanup       [STUB] Clean up orphan database items.
  migrate       Migrate an LDP repository to Lakesuperior.
  stats         Print repository statistics.
```

All entries marked `[STUB]` are not yet implemented, however the `lsup_admin <command> --help` command will issue a description of what the command is meant to do. Check the issues page for what's on the radar.

All of the above commands are also available via, and based upon, the native Python API.

1.10.4 lsup-benchmark

This command is used to run performance tests in a predictable way.

The command line options can be queried with the `--help` option:

```
Usage: lsup-benchmark [OPTIONS]

  Run the benchmark.

Options:
  -m, --mode TEXT           Mode of ingestion. One of `ldp`, `python`. With
                            the former, the HTTP/LDP web server is used. With
                            the latter, the Python API is used, in which case
                            the server need not be running. Default:
                            http://localhost:8000/ldp
  -e, --endpoint TEXT       LDP endpoint. Only meaningful with `ldp` mode.
                            Default: http://localhost:8000/ldp
  -c, --count INTEGER       Number of resources to ingest. Default: {def_ct}
  -p, --parent TEXT         Path to the container resource under which the new
                            resources will be created. It must begin with a
                            slash (`/`) character. Default: /pomegranate
  -d, --delete-container    Delete container resource and its children if
                            already existing. By default, the container is not
                            deleted and new resources are added to it.
  -X, --method TEXT         HTTP method to use. Case insensitive. Either PUT
                            or POST. Default: PUT
  -s, --graph-size INTEGER  Number of triples in each random graph, rounded
                            down to a multiple of 8. Default: 200
  -S, --image-size INTEGER  Size of random square image, in pixels for each
                            dimension, rounded down to a multiple of 8.
                            Default: 1024
  -t, --resource-type TEXT  Type of resources to ingest. One of `r` (only LDP-
                            RS, i.e. RDF), `n` (only  LDP-NR, i.e. binaries),
```

(continues on next page)

(continued from previous page)

```
                              or `b` (50/50% of both). Default: r
  -P, --plot                  Plot a graph of ingest timings. The graph figure
                              is displayed on screen with basic manipulation and
                              save options.
  --help                      Show this message and exit.
```

The benchmark tool is able to create RDF sources, or non-RDF, or an equal mix of them, via POST or PUT, in a given lDP endpoint. It runs single threaded.

The RDF sources are randomly generated graphs of consistent size and complexity. They include a mix of in-repository references, literals, and external URIs. Each graph has 200 triples by default.

The non-RDF sources are randomly generated 1024x1024 pixel PNG images.

You are warmly encouraged to run the script and share the performance results (*TODO add template for posting results*).

1.10.5 `lsup-profiler`

This command launches a single-threaded HTTP server (Flask) on port 5000 that logs profiling information. This is useful for analyzing application performance.

For more information, consult the Python profilers guide.

Do not launch this while a WSGI server (`fcrepo`) is already running, because that also launches a Flask server on port 5000.

1.10.6 Locust (experimental)

Locust is an HTTP load tester. It can launch many requests on an HTTP endpoint. A rudimentary Locust file is currently available.

To run Locust against Lakesuperior or FCREPO, run in the project root:

```
locust -f lakesuperior/util/locustfile.py http://localhost:8000/
```

1.11 Contributing to Lakesuperior

Lakesuperior has been so far a single person's off-hours project (with much valuable input from several sides). In order to turn into anything close to a Beta release and eventually to a production-ready implementation, it needs some community love.

Contributions are welcome in all forms, including ideas, issue reports, or even just spinning up the software and providing some feedback. Lakesuperior is meant to live as a community project.

1.11.1 Development Setup

To set up the software for developing code, documentation, or tests, start in an empty project folder:

```
python3 -m venv venv
source venv/bin/activate
git clone --recurse-submodules https://github.com/scossu/lakesuperior.git src
cd src
pip install -e .
```

This will allow to alter the code without having to re-run `pip install` after changes (unless one is changing the Cython modules; see below).

1.11.2 Modifying Cython Modules

Cython files must be recompiled into C files and then into binary files every time they are changed. To recompile Lakesuperior modules, run:

```
python setup.py --build_ext --inplace
```

For a faster compilation while testing, the environment variable `CFLAGS` can set to `-O0` to turn off compiler optimization. The runtime code may run slower so this is not recommended for performance benchmarking.

Refer to the Cython documentation for a detailed description of the Cython compilation process.

1.11.3 Contribution Guidelines

You can contribute by (from least to most involved):

- Installing the repository and reporting any issues
- Testing on other platforms (OS X, Windows, other Linux distros)
- Loading some real-world data set and sharing interesting results
- Amending incorrect documentation or adding missing one
- Adding test coverage (**HOT**)
- Browsing the list of open issues and picking a ticket that you may find interesting and within your reach
- Suggesting new functionality or improvements and implementing them

Please open a ticket and discuss the issue you are raising before opening a PR.

Documentation is critical. If you implement new modules, classes or methods, or modify them, please document them thoroughly and verify that the API docs are displaying and linking correctly.

Likewise, please add mindful testing to new fatures or bug fixes.

Development is done on the `master` branch. If you have an addition to the code that you have previously discussed by opening an issue, please fork the repo, create a new branch for your topic and open a pull request against master.

Last but not least, read carefully the Code of Conduct.

1.12 Release Notes

1.12.1 1.0 Alpha 20

April 08, 2019

After 6 months and almost 200 commits, this release completes a major effort to further move performance-critical sections of the code to C libraries.

The storage layer has been simplified by moving from 5-byte `char*` keys to `size_t` integers (8 bytes in most architectures). This means that this version requires a data migration from previous vresions.

Performance benchmarks have been updated with new results and charts.

1.12.2 1.0 Alpha 19 HOTFIX

October 10, 2018

A hotfix release was necessary to adjust settings for the source to build correctly on Read The Docs and Docker Hub, and to package correctly on PyPI.

1.12.3 1.0 Alpha 18

October 10, 2018

This release represents a major rewrite of many parts of the application, which took several months of research and refactoring. The main change is a much more efficient storage engine. The storage module and ancillary modules were completely rewritten in Cython to use the LMDB C API rather than the LMDB Python bindings[1]. Most of the stack was also modified to accommodate the new interface.

Most of the performance gains are visible in the Python API. Further optimizations would be more involved, including refactoring RDF serialization and deserialization libraries, and/or SPARQL language parsers. That may be done at the appropriate time.

Note that from this version on, Lakesuperior is distributed with C extensions (the Cython modules). This requires having a C compiler installed. Most Linux distributions come with gcc. The C sources generated by Cython are distributed with the package to avoid a dependency on Cython. They are very large files. Adopters and most contributors should not be concerned with these files.

New Features

- New Python API objects (`SimpleGraph` and `Imr`) for simple and resource-efficient handling of sets of triples.
- New features in benchmark script:
  - Command line options that deprecate interactive input
  - Request/time graph plotting

Enhancements

- New storage layer providing significant performance gains, especially in read operations. See *Performance benchmark results*.
- Test coverage has increased (but is still not satisfactory).

[1] Nothing wrong with @dw's excellent Python LMDB library; however, Lakesuperior performs heavy manipulation on data retrieved from the store which is more efficiently done in C/Cython.

Bug fixes

Several pre-existing bugs were resolved in the course of refactoring the code and writing tests for it:

- faulty lookup method involving all-unbound triples
- Triples clean-up after delete
- Other minor bugs

Regressions

- Removed ETags from LDP-RS resources. Read the *Delta document* for more explanation. This feature may be restored once clear specifications are laid out.
- Increased size of the index file. This is a necessary price to pay for faster retrieval. The size is still quite small: see *Performance* for details.

Other Significant Changes

- The `fcrepo` shell script, which launches the multi-threaded gunicorn web server, is now only available for Unix system. It would not work in Windows in previous versions anyways. Note that now this script is not in the `$PATH` environment variable of the virtualenv and must be invoked by its full path.
- Main LDP-RS data and index are now in a single file. This simplifed the code significantly. The previous decision to have separate files was made for possible scenarios where the indices could be written asynchronously, but that will not be pursued in the foreseeable future because not corruption-proof.
- Release notes are now in a self-standing document (this one) and are referred to in Github releases. This is part of a progressive effort to make the project more independent from vendor-specific features (unrelated from Github's recent ownership change).

1.12.4 1.0 Alpha 17 HOTFIX

May 11, 2018

Hotfix resolving an issue with version files resulting in an error in the UI homepage.

1.12.5 1.0 Alpha 16

April 28, 2018

This release was triggered by accidentally merging a PR into master instead of devleopment, which caused CI to push the a16 release, whose name cannot be reused…

In any case, all tests pass and the PR actually brings in a new important feature, i.e. support for multiple RDF serialization formats, so might as well consider it a full release.

1.12.6 1.0 Alpha 15

April 27, 2018

Alpha 15 completes version handling and deletion & restore of resources, two key features for the beta track. It also addresses a regression issue with LDP-NR POSTs.

All clients are encouraged to upgrade to this last version which fixes a critical issue.

New Features

- Complete bury, resurrect and forget resources
- Complete reverting to version (#21)

Enhancements

- Dramatic performance increase in GET fcr:versions (#20)
- Refactor and simplify deletion-related code (#20)
- Minimize number of triples copied on version creation
- Complete removing SPARQL statements from model and store layout; remove redundant methods

Bug Fixes

- LDP-NR POST returns 500 (#47)

Other Changes

- Add PyPI package badge in README

Acknowledgments

Thanks to @acoburn for reporting and testing issues.

1.12.7 1.0 Alpha 14

April 23, 2018

Alpha 14 implements Term Search, one of the key features necessary to move toward a Beta release. Documentation about this new feature, which is available both in the UI and REST API, is at http://lakesuperior.readthedocs.io/en/latest/discovery.html#term-search and in the LAKEsuperior term search page itself.

This release also addresses issues with Direct and Indirect Containers, as well as several other server-side and client-side improvements. Client making use of LDP-DC and LDP-IC resources are encouraged to upgrade to this version.

New Features

- Term search (#19)
- Allow creating resources by providing a serialized RDF bytestring (#59)

Enhancements

- Upgrade UI libraries to Bootstrap 4
- Write tests for Direct and Indirect Containers (#22)

Bug Fixes

- LDP-RS creation with POST and Turtle payload results in a LDP-NR (#56)
- Cannot add children to direct containers (#57)

Acknowledgments

- Thanks to @acoburn for reporting issues.

1.12.8 1.0 Alpha 13

April 14, 2018

Alpha 13 addressed a number of internal issues and restructured some core components, most notably configuration and globals handling.

New features

- Report file for referential integrity check (#43)
- Support PATCH operations on root node (#44)
- Version number is now controlled by a single file
- Version number added to home page

Enhancements

- Better handling of thread-local and global variables
- Prevent migration script from failing if an HTTP requests fails
- Light LMDB store optimizations

Bug fixes

- Removed extraneous characters from `anchor` link in output headers (#48)

Other changes

- Added template for release notes (this document). This is not a feature supported by Github, but the template can be manually copied and pasted from `.github/release_template.md`.

Notes & caveats

- Custom configurations may need to be adapted to the new configuration scheme. Please look at changes in `lakesuperior/etc.defaults`. Most notably, there is now a single `data_dir` location, and `test.yml` file is now deprecated.

Acknowledgments

Thanks to @acoburn for testing and reporting several isssues.

1.12.9 1.0 Alpha 12

April 7, 2018

Alpha 12 addresses some substantial enhancements to the Python API and code refactoring, additional documentation and integration.

Features

- Integrate Travis with PyPI. Builds are now deployed to PyPI at every version upgrade.
- Allow updating resources with triple deltas in Python API.

Enhancements

- Streamlined resource creation logic, removed redundant methods.
- Allow PUT with empty payload on existing resources.

Bug Fixes

- Referential integrity did not parse fragment URIs correctly.

Other

- Added documentation for discovery and query, and Python API usage examples.

1.12.10 1.0 Alpha 11

April 4, 2018

Alpha 11 introduces some minor documentation amendments and fixes an issue with the distribution package.

Features

None with this release.

Enhancements

- Documentation improvements.

Bug Fixes

- Issue with config files in wheel build.

1.12.11 1.0 Alpha 10

April 3, 2018

Alpha 10 introduces a completely revamped documentation and integration with setuptools to generate Python packages on PyPI. It incorporates the unreleased alpha9.

Features

- Translate all documentation pages to .rst
- Several new documentation pages
- Translate all docstrings to .rst (#32)
- Documentation now automatically generated by Sphinx
- Setuptools integration to create Python wheels

Enhancements

- Moved several files, including default config, into lakesuperior package
- Reworked WSGI (gunicorn) server configuration, now exposed to user as .yml rather than .py
- Made most scripts non-executable (executables are now generated by setuptools)
- CI uses setup.py for testing
- Web server no longer aborts if STOMP service is not accessible

Bug Fixes

None with this release.

Other

- Documentation now available on https://lakesuperior.readthedocs.io and updated with each release
- Python package hosted on https://pypi.org/project/lakesuperior/. Please make sure you read the updated install instructions.
- Switch semantic version tag naming to a format compatible with PyPI.

1.12.12 1.0 Alpha 8

March 26, 2018

Alpha 8 introduces a migration tool and other enhancements and bug fixes.

Features

- Migration tool (#23)
- Referential integrity checks (#31)

Enhancements

- More efficient and cleaner handling of data keys (#17)
- Enhanced resource view in UI
- Simplified and more efficient PATCH operations
- Zero configuration startup
- More minor enhancements

Bug Fixes

- STOMP protocol mismatch
- Missing UID slash when POSTing to root (#26)
- Running out of readers in long-running processes

Other

- Travis and Docker integration

1.12.13 1.0 Alpha 7.1

March 9, 2018

1.12.14 1.0 Alpha 7

March 6, 2018

This is the first publicly advertised release of LAKEsuperior.

Some major features are missing and test coverage is very low but the application is proven to perform several basic operations on its own and with Hyrax 2.0.

1.12.15 1.0 Alpha 6

February 28, 2018

1.12.16 1.0 Alpha 5

February 14, 2018

1.12.17 1.0 Alpha 4

January 13, 2018

1.12.18 1.0 Alpha 3

January 9, 2018

1.12.19 1.0 Alpha 2

Dec 23, 2017

1.12.20 1.0 Alpha 1

Nov 24, 2017

1.13 API Documentation

1.13.1 Main Interface

The Lakesuperior API modules of most interest for a client are:

- *lakesuperior.api.resource*
- lakesupeiror.api.query
- *lakesuperior.api.admin*

1.13.2 Lower-Level Interfaces

lakesuperior.model.ldp handles the concepts of LDP resources, containers, binaries, etc.

lakesuperior.store.ldp_rs.rsrc_centric_layout handles the "layout" of LDP resources as named graphs in a triplestore. It is possible (currently not without changes to the core libraries) to devise a different layout for e.g. a more sparse, or richer, data model.

Similarly, *lakesuperior.store.ldp_nr.base_non_rdf_layout* offers an interface to handle the layout of LDPR resources. Currently only one implementation is available but it is also possible to create a new module to e.g. handle files in an S3 bucket, a Cassandra database, or create Bagit or OCFL file structures, and configure Lakesuperior to use one, or more, of those persistence methods.

1.13.3 Deep Tissue

Some of the Cython libraries in *lakesuperior.model.structures*, *lakesuperior.model.rdf*, and *lakesuperior.store* have Python-accessible methods for high-performance manipulation. The *lakesuperior.model.rdf.graph.Graph* class is an example of that.

1.13.4 Full API Documentation

lakesuperior

lakesuperior package

class lakesuperior.**Env**
Bases: object

lakesuperior.**basedir = '/home/docs/checkouts/readthedocs.org/user_builds/lakesuperior/envs/**
Base directory for the module.

This can be used by modules looking for configuration and data files to be referenced or copied with a known path relative to the package root.

Return type str

lakesuperior.**env = <lakesuperior.Env object>**
A pox on "globals are evil".

All-purpose bucket for storing global variables. Different environments (e.g. webapp, test suite) put the appropriate value in it. The most important values to be stored are app_conf (either from lakesuperior.config_parser.config or lakesuperior.config_parser.test_config) and app_globals (obtained by an instance of lakesuperior.globals.AppGlobals).

e.g.:

```
>>> from lakesuperior.config_parser import config
>>> from lakesuperior.globals import AppGlobals
>>> from lakesuperior import env
>>> env.app_globals = AppGlobals(config)
```

This is automated in non-test environments by importing *lakesuperior.env_setup*.

Return type Object

lakesuperior.**thread_env = <_thread._local object>**
Thread-local environment.

This is used to store thread-specific variables such as start/end request timestamps.

Return type threading.local

Subpackages

lakesuperior.api package

Submodules

lakesuperior.api.admin module

Admin API.

This module contains maintenance utilities and stats.

lakesuperior.api.admin.**fixity_check**(*uid*)
Check fixity of a resource.

This calculates the checksum of a resource and validates it against the checksum stored in its metadata (premis:hasMessageDigest).

Parameters **uid** (*str*) – UID of the resource to be checked.

Return type None

Raises lakesuperior.exceptions.ChecksumValidationError: the cecksums do not match. This indicates corruption.

Raises lakesuperior.exceptions.IncompatibleLdpTypeError: if the resource is not an LDP-NR.

lakesuperior.api.admin.**integrity_check**()
Check integrity of the data set.

At the moment this is limited to referential integrity. Other checks can be added and triggered by different argument flags.

lakesuperior.api.admin.**migrate**(*src*, *dest*, *start_pts=None*, *list_file=None*, ***kwargs*)
Migrate an LDP repository to a new Lakesuperior instance.

See `Migrator.__init__()`.

lakesuperior.api.admin.**stats**()
Get repository statistics.

Return type dict

Returns Store statistics, resource statistics.

lakesuperior.api.query module

lakesuperior.api.query.**fulltext_lookup**(*pattern*)
Look up one term by partial match.

TODO: reserved for future use. A 'Whoosh <https://whoosh.readthedocs.io/>'__ or similar full-text index is necessary for this.

lakesuperior.api.query.**operands = ('_id', '=', '!=', '<', '>', '<=', '>=')**
Available term comparators for term query.

The `_uri` term is used to match URIRef terms, all other comparators are used against literals.

lakesuperior.api.query.**sparql_query**(*qry_str*, *fmt*)
Send a SPARQL query to the triplestore.

Parameters

- **qry_str** (*str*) – SPARQL query string. SPARQL 1.1 Query Language (https://www.w3.org/TR/sparql11-query/) is supported.
- **fmt** (*str*) – Serialization format. This varies depending on the query type (SELECT, ASK, CONSTRUCT, etc.). [TODO Add reference to RDFLib serialization formats]

Return type BytesIO

Returns Serialized SPARQL results.

lakesuperior.api.query.**term_query**(*terms*, *or_logic=False*)
Query resources by predicates, comparators and values.

Comparators can be against literal or URIRef objects. For a list of comparators and their meanings, see the documentation and source for *operands*.

Parameters

- **terms** (*list(tuple{3})*) – List of 3-tuples containing:
  - Predicate URI (rdflib.URIRef)
  - Comparator value (str)
  - Value to compare to (rdflib.URIRef or rdflib.Literal or str)

- **or_logic** (*bool*) – Whether to concatenate multiple query terms with OR logic (uses SPARQL `UNION` statements). The default is False (i.e. terms are concatenated as standard SPARQL statements).

`lakesuperior.api.query.`**triple_match**(*s=None*, *p=None*, *o=None*, *return_full=False*)

Query store by matching triple patterns.

Any of the `s`, `p` or `o` terms can be None to represent a wildcard.

This method is for triple matching only; it does not allow to query, nor exposes to the caller, any context.

Parameters

- **s** (*rdflib.term.Identifier*) – Subject term.
- **p** (*rdflib.term.Identifier*) – Predicate term.
- **o** (*rdflib.term.Identifier*) – Object term.
- **return_full** (*bool*) – if `False` (the default), the returned values in the set are the URIs of the resources found. If True, the full set of matching triples is returned.

Return type set(tuple(rdflib.term.Identifier){3}) or set(rdflib.URIRef)

Returns Matching resource URIs if `return_full` is false, or matching triples otherwise.

lakesuperior.api.resource module

Primary API for resource manipulation.

Quickstart:

```
>>> # First import default configuration and globals--only done once.
>>> import lakesuperior.default_env
>>> from lakesuperior.api import resource
>>> # Get root resource.
>>> rsrc = resource.get('/')
>>> # Dump graph.
>>> with rsrc.imr.store.txn_ctx():
>>>     print({*rsrc.imr.as_rdflib()})
{(rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://purl.org/dc/terms/title'),
  rdflib.term.Literal('Repository Root')),
 (rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
  rdflib.term.URIRef('http://fedora.info/definitions/v4/repository#Container')),
 (rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
  rdflib.term.URIRef('http://fedora.info/definitions/v4/repository#RepositoryRoot')),
 (rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
  rdflib.term.URIRef('http://fedora.info/definitions/v4/repository#Resource')),
 (rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
  rdflib.term.URIRef('http://www.w3.org/ns/ldp#BasicContainer')),
 (rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
  rdflib.term.URIRef('http://www.w3.org/ns/ldp#Container')),
 (rdflib.term.URIRef('info:fcres/'),
  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
  rdflib.term.URIRef('http://www.w3.org/ns/ldp#RDFSource'))}
```

lakesuperior.api.resource.**create**(*parent*, *slug=None*, ***kwargs*)
Mint a new UID and create a resource.

The UID is computed from a given parent UID and a "slug", a proposed path relative to the parent. The application will attempt to use the suggested path but it may use a different one if a conflict with an existing resource arises.

Parameters

- **parent** (*str*) – UID of the parent resource.
- **slug** (*str*) – Tentative path relative to the parent UID.
- ****kwargs** – Other parameters are passed to the `from_provided()` method.

Return type tuple(str, lakesuperior.model.ldp.ldpr.Ldpr)

Returns A tuple of: 1. Event type (str): whether the resource was created or updated. 2. Resource (lakesuperior.model.ldp.ldpr.Ldpr): The new or updated resource.

lakesuperior.api.resource.**create_or_replace**(*uid*, ***kwargs*)
Create or replace a resource with a specified UID.

Parameters

- **uid** (*string*) – UID of the resource to be created or updated.
- ****kwargs** – Other parameters are passed to the `from_provided()` method.

Return type tuple(str, lakesuperior.model.ldp.ldpr.Ldpr)

Returns

A tuple of: 1. Event type (str): whether the resource was created or updated. 2. Resource (lakesuperior.model.ldp.ldpr.Ldpr): The new or updated

resource.

lakesuperior.api.resource.**create_version**(*uid*, *ver_uid*)
Create a resource version.

Parameters

- **uid** (*string*) – Resource UID.
- **ver_uid** (*string*) – Version UID to be appended to the resource URI. NOTE: this is a "slug", i.e. the version URI is not guaranteed to be the one indicated.

Return type str

Returns Version UID.

lakesuperior.api.resource.**delete**(*uid*, *soft=True*, *inbound=True*)
Delete a resource.

Parameters

- **uid** (*string*) – Resource UID.
- **soft** (*bool*) – Whether to perform a soft-delete and leave a tombstone resource, or wipe any memory of the resource.

lakesuperior.api.resource.**exists**(*uid*)
Return whether a resource exists (is stored) in the repository.

Parameters **uid** (*string*) – Resource UID.

lakesuperior.api.resource.**get**(*uid*, *repr_options={}*)
Get an LDPR resource.

The resource comes preloaded with user data and metadata as indicated by the *repr_options* argument. Any further handling of this resource is done outside of a transaction.

Parameters

- **uid** (*string*) – Resource UID.
- **repr_options** – (dict(bool)) Representation options. This is a dict that is unpacked downstream in the process. The default empty dict results in default values. The accepted dict keys are:

- incl_inbound: include inbound references. Default: False.
- incl_children: include children URIs. Default: True.
- embed_children: Embed full graph of all child resources. Default: False

lakesuperior.api.resource.**get_metadata**(*uid*)
Get metadata (admin triples) of an LDPR resource.

Parameters **uid** (*string*) – Resource UID.

lakesuperior.api.resource.**get_version**(*uid*, *ver_uid*)
Get version metadata (fcr:versions).

lakesuperior.api.resource.**get_version_info**(*uid*)
Get version metadata (fcr:versions).

lakesuperior.api.resource.**resurrect**(*uid*)
Reinstate a buried (soft-deleted) resource.

Parameters **uid** (*str*) – Resource UID.

lakesuperior.api.resource.**revert_to_version**(*uid*, *ver_uid*)
Restore a resource to a previous version state.

Parameters

- **uid** (*str*) – Resource UID.
- **ver_uid** (*str*) – Version UID.

lakesuperior.api.resource.**transaction**(*write=False*)
Handle atomic operations in a store.

This wrapper ensures that a write operation is performed atomically. It also takes care of sending a message for each resource changed in the transaction.

ALL write operations on the LDP-RS and LDP-NR stores go through this wrapper.

lakesuperior.api.resource.**update**(*uid*, *update_str*, *is_metadata=False*, *handling='strict'*)
Update a resource with a SPARQL-Update string.

Parameters

- **uid** (*string*) – Resource UID.
- **update_str** (*string*) – SPARQL-Update statements.
- **is_metadata** (*bool*) – Whether the resource metadata are being updated.

- **handling** (*str*) – How to handle server-managed triples. `strict` (the default) rejects the update with an exception if server-managed triples are being changed. `lenient` modifies the update graph so offending triples are removed and the update can be applied.

Raises ***InvalidResourceError*** – If `is_metadata` is False and the resource being updated is a LDP-NR.

`lakesuperior.api.resource.`**`update_delta`**(*uid*, *remove_trp*, *add_trp*)

Update a resource graph (LDP-RS or LDP-NR) with sets of add/remove triples.

A set of triples to add and/or a set of triples to remove may be provided.

Parameters

- **uid** (*string*) – Resource UID.
- **remove_trp** (*set (tuple (rdflib.term.Identifier))*) – Triples to remove, as 3-tuples of RDFLib terms.
- **add_trp** (*set (tuple (rdflib.term.Identifier))*) – Triples to add, as 3-tuples of RDFLib terms.

lakesuperior.cy_include package

lakesuperior.dictionaries package

Submodules

lakesuperior.dictionaries.namespaces module

lakesuperior.dictionaries.srv_mgd_terms module

lakesuperior.endpoints package

Submodules

lakesuperior.endpoints.admin module

`lakesuperior.endpoints.admin.`**`admin_tools`**()

Admin tools.

@TODO stub.

`lakesuperior.endpoints.admin.`**`fixity_check`**(*uid*)

Check the fixity of a resource.

`lakesuperior.endpoints.admin.`**`stats`**()

Get repository statistics.

lakesuperior.endpoints.ldp module

`lakesuperior.endpoints.ldp.`**`DEFAULT_RDF_MIMETYPE = 'text/turtle'`**

Fallback serialization format used when no acceptable formats are specified.

`lakesuperior.endpoints.ldp.`**`bp_url_defaults`**(*endpoint*, *values*)

lakesuperior.endpoints.ldp.**bp_url_value_preprocessor**(*endpoint*, *values*)

lakesuperior.endpoints.ldp.**delete_resource**(*uid*)
Delete a resource and optionally leave a tombstone.

This behaves differently from FCREPO. A tombstone indicated that the resource is no longer available at its current location, but its historic snapshots still are. Also, deleting a resource with a tombstone creates one more version snapshot of the resource prior to being deleted.

In order to completely wipe out all traces of a resource, the tombstone must be deleted as well, or the `Prefer:no-tombstone` header can be used. The latter will forget (completely delete) the resource immediately.

lakesuperior.endpoints.ldp.**get_resource**(*uid*, *out_fmt=None*)
https://www.w3.org/TR/ldp/#ldpr-HTTP_GET

Retrieve RDF or binary content.

Parameters

- **uid** (*str*) – UID of resource to retrieve. The repository root has an empty string for UID.
- **out_fmt** (*str*) – Force output to RDF or non-RDF if the resource is a LDP-NR. This is not available in the API but is used e.g. by the `\*/fcr:metadata` and `\*/fcr:content` endpoints. The default is False.

lakesuperior.endpoints.ldp.**get_version**(*uid*, *ver_uid*)
Get an individual resource version.

Parameters

- **uid** (*str*) – Resource UID.
- **ver_uid** (*str*) – Version UID.

lakesuperior.endpoints.ldp.**get_version_info**(*uid*)
Get version info (*fcr:versions*).

Parameters **uid** (*str*) – UID of resource to retrieve versions for.

lakesuperior.endpoints.ldp.**instantiate_req_vars**()

lakesuperior.endpoints.ldp.**ldp = <flask.blueprints.Blueprint object>**
Blueprint for LDP REST API. This is what is usually found under `/rest/` in standard fcrepo4. Here, it is under `/ldp` but initially `/rest` will be kept for backward compatibility.

lakesuperior.endpoints.ldp.**log_request_end**(*rsp*)

lakesuperior.endpoints.ldp.**log_request_start**()

lakesuperior.endpoints.ldp.**parse_repr_options**(*retr_opts*)
Set options to retrieve IMR.

Ideally, IMR retrieval is done once per request, so all the options are set once in the *imr()* property.

:param dict retr_opts:: Options parsed from *Prefer* header.

lakesuperior.endpoints.ldp.**patch_resource**(*uid*, *is_metadata=False*)
https://www.w3.org/TR/ldp/#ldpr-HTTP_PATCH

Update an existing resource with a SPARQL-UPDATE payload.

lakesuperior.endpoints.ldp.**patch_resource_metadata**(*uid*)

lakesuperior.endpoints.ldp.**patch_version**(*uid*, *ver_uid*)
Revert to a previous version.

NOTE: This creates a new version snapshot.

Parameters

- **uid** (*str*) – Resource UID.
- **ver_uid** (*str*) – Version UID.

lakesuperior.endpoints.ldp.**post_resource**(*parent_uid*)
https://www.w3.org/TR/ldp/#ldpr-HTTP_POST

Add a new resource in a new URI.

lakesuperior.endpoints.ldp.**post_version**(*uid*)
Create a new resource version.

lakesuperior.endpoints.ldp.**put_resource**(*uid*)
https://www.w3.org/TR/ldp/#ldpr-HTTP_PUT

Add or replace a new resource at a specified URI.

lakesuperior.endpoints.ldp.**rdf_parsable_mimetypes = {'application/ld+json', 'application/n-**
MIMEtypes that can be parsed into RDF.

lakesuperior.endpoints.ldp.**rdf_serializable_mimetypes = {'application/ld+json', 'applicatio**
MIMEtypes that RDF can be serialized into.

These are not automatically derived from RDFLib because only triple (not quad) serializations are applicable.

lakesuperior.endpoints.ldp.**set_post_put_params**()
Sets handling and content disposition for POST and PUT by parsing headers.

lakesuperior.endpoints.ldp.**std_headers = {'Accept-Patch': 'application/sparql-update', 'Ac**
Predicates excluded by view.

lakesuperior.endpoints.ldp.**tombstone**(*uid*)
Handle all tombstone operations.

The only allowed methods are POST and DELETE; any other verb will return a 405.

lakesuperior.endpoints.main module

lakesuperior.endpoints.main.**index**()
Homepage.

lakesuperior.endpoints.main.**ldp_constraints**()
LDP term constraints.

lakesuperior.endpoints.query module

lakesuperior.endpoints.query.**sparql**()
Perform a direct SPARQL query on the underlying triplestore.

Parameters **qry** (*str*) – SPARQL query string.

lakesuperior.endpoints.query.**term_search**()
Search by entering a search term and optional property and comparison term.

lakesuperior.messaging package

Submodules

lakesuperior.messaging.formatters module

class lakesuperior.messaging.formatters.**ASDeltaFormatter**(*rsrc_uri*, *ev_type*, *timestamp*, *rsrc_type*, *actor*, *data=None*)

Bases: *lakesuperior.messaging.formatters.BaseASFormatter*

Sends the same information as *ASResourceFormatter* with the addition of the triples that were added and the ones that were removed in the request. This may be used to send rich provenance data to a preservation system.

class lakesuperior.messaging.formatters.**ASResourceFormatter**(*rsrc_uri*, *ev_type*, *timestamp*, *rsrc_type*, *actor*, *data=None*)

Bases: *lakesuperior.messaging.formatters.BaseASFormatter*

Sends information about a resource being created, updated or deleted, by who and when, with no further information about what changed.

class lakesuperior.messaging.formatters.**BaseASFormatter**(*rsrc_uri*, *ev_type*, *timestamp*, *rsrc_type*, *actor*, *data=None*)

Bases: object

Format message as ActivityStreams.

This is not really a *logging.Formatter* subclass, but a plain string builder.

ev_names = {'_create_': 'Resource Creation', '_delete_': 'Resource Deletion', '_upda

ev_types = {'_create_': 'Create', '_delete_': 'Delete', '_update_': 'Update'}

lakesuperior.messaging.handlers module

class lakesuperior.messaging.handlers.**StompHandler**(*conf*)

Bases: logging.Handler

Send messages to a remote queue broker using the STOMP protocol.

This module is named and configured separately from standard logging for clarity about its scope: while logging has an informational purpose, this module has a functional one.

emit(*record*)

Send the message to the destination endpoint.

lakesuperior.messaging.messenger module

class lakesuperior.messaging.messenger.**Messenger**(*config*)

Bases: object

Very simple message sender using the standard Python logging facility.

send(**args*, ***kwargs*)

Send one or more external messages.

lakesuperior.model package

Subpackages

lakesuperior.model.rdf package

Model for RDF entities: Term, Triple, Graph.

Members of this package are the core building blocks of the Lakesuperior RDF model. They are C extensions mostly used in higher layers of the application, but some of them also have a public Python API to allow efficient manipulation of large RDF datasets.

See individual modules for detailed documentation:

- *lakesuperior.model.rdf.term*
- *lakesuperior.model.rdf.triple*
- *lakesuperior.model.rdf.graph*

Submodules

lakesuperior.model.rdf.graph module

Graph class and factories.

class lakesuperior.model.rdf.graph.**Graph**

Bases: object

Fast implementation of a graph.

Most functions should mimic RDFLib's graph with less overhead. It uses the same funny but functional slicing notation.

A Graph contains a *lakesuperior.model.structures.keyset.Keyset* at its core and is bound to a *LmdbTriplestore*. This makes lookups and boolean operations very efficient because all these operations are performed on an array of integers.

In order to retrieve RDF values from a Graph, the underlying store must be looked up. This can be done in a different transaction than the one used to create or otherwise manipulate the graph.

Similarly, any operation such as adding, changing or looking up triples needs a store transaction.

Boolean operations between graphs (union, intersection, etc) and other operations that don't require an explicit term as an input or output (e.g. __repr__ or size calculation) don't require a transaction to be opened.

Every time a term is looked up or added to even a temporary graph, that term is added to the store and creates a key. This is because in the majority of cases that term is likely to be stored permanently anyway, and it's more efficient to hash it and allocate it immediately. A cleanup function to remove all orphaned terms (not in any triple or context index) can be later devised to compact the database.

Even though any operation may involve adding new terms to the store, a read-only transaction is sufficient. Lakesuperior will open a write transaction automatically only if necessary and only for the time needed to enter the new terms.

An instance of this class can be created from a RDF python string with the *from_rdf()* factory function or converted to a rdflib.Graph instance.

add
Add triples to the graph.

This method checks for duplicates.

Parameters **triples** (*iterable*) – iterable of 3-tuple triples.

as_rdflib
Return the data set as an RDFLib Graph.

Return type rdflib.Graph

capacity

copy
Create copy of the graph with a different (or no) URI.

Parameters **uri** (*str*) – URI of the new graph. This should be different from the original.

data

empty_copy
Create an empty copy with same capacity and store binding.

Parameters **uri** (*str*) – URI of the new graph. This should be different from the original.

keys
keys: lakesuperior.model.structures.keyset.Keyset

lookup
Look up triples by a pattern.

This function converts RDFLib terms into the serialized format stored in the graph's internal structure and compares them bytewise.

Any and all of the lookup terms msy be `None`.

Return type *Graph*

Returns New Graph instance with matching triples.

remove
Remove triples by pattern.

The pattern used is similar to `LmdbTripleStore.delete()`.

set
Set a single value for subject and predicate.

Remove all triples matching `s` and `p` before adding `s p o`.

store

terms_by_type
Get all terms of a type: subject, predicate or object.

Parameters **type** (*str*) – One of `s`, `p` or `o`.

txn_ctx

uri
uri: object

value
Get an individual value for a given predicate.

Parameters

- **p** (*rdflib.termNode*) – Predicate to search for.
- **strict** (*bool*) – If set to `True` the method raises an error if more than one value is found. If `False` (the default) only the first found result is returned.

Return type rdflib.term.Node

`lakesuperior.model.rdf.graph.`**`from_rdf`**

Create a Graph from a serialized RDF string.

This factory function takes the same arguments as `rdflib.Graph.parse()`.

Parameters

- **store** – see `Graph.__cinit__()`.
- **uri** – see `Graph.__cinit__()`.
- ***args** – Positional arguments passed to RDFlib's `parse`.
- ****kwargs** – Keyword arguments passed to RDFlib's `parse`.

Return type *Graph*

lakesuperior.model.rdf.term module

Term model.

`Term` is not defined as a Cython or Python class. It is a C structure, hence only visible by the Cython layer of the application.

Terms can be converted from/to RDFlib terms, and deserialized from, or serialized to, binary buffer structures. This is the form that terms are stored in the data store.

If uses require a public API, a proper Term Cython class with a Python API could be developed in the future.

lakesuperior.model.rdf.triple module

Triple model.

This is a very light-weight implementation of a Triple model, available as C structures only. Two types of structures are defined: `Triple`, with pointers to **:py:model:‘lakesuperior.model.rdf.term‘** objects, and `BufferTriple`, with pointers to byte buffers of serialized terms.

lakesuperior.model.structures package

Submodules

lakesuperior.model.structures.hash module

C hashing functions used with Cython models.

The hashing algorithm is SpookyHash which produces up to 128-bit (16-byte) digests.

lakesuperior.model.structures.keyset module

class lakesuperior.model.structures.keyset.**Keyset**
Bases: object

Memory-contiguous array of ‘‘TripleKey‘‘s.

The keys are size_t values that are linked to terms in the triplestore. Therefore, a triplestore lookup is necessary to view or use the terms, but several types of manipulation and filtering can be done very efficiently without looking at the term values.

The set is not checked for duplicates all the time: e.g., when creating from a single set of triples coming from the store, the duplicate check is turned off for efficiency and because the source is guaranteed to provide unique values. When merging with other sets, duplicate checking should be turned on.

Since this class is based on a contiguous block of memory, it is best not to do targeted manipulation. Several operations involve copying the whole data block, so e.g. bulk removal and intersection are much more efficient than individual record operations.

Submodules

lakesuperior.model.base module

Basic model typedefs, constants and common methods.

lakesuperior.model.callbacks module

Callback methods for various loop functions.

lakesuperior.store package

Subpackages

lakesuperior.store.ldp_nr package

Submodules

lakesuperior.store.ldp_nr.base_non_rdf_layout module

class lakesuperior.store.ldp_nr.base_non_rdf_layout.**BaseNonRdfLayout**(*config*)
Bases: object

Abstract class for setting the non-RDF (bitstream) store layout.

Differerent layouts can be created by implementing all the abstract methods of this class. A non-RDF layout is not necessarily restricted to a traditional filesystem—e.g. a layout persisting to HDFS can be written too.

delete(*id*)
Delete a stream by its identifier (i.e. checksum).

file_ct
Calculated the store size on disk.

local_path(*uuid*)
Return the local path of a file.

persist(*stream*)
Store the stream in the designated persistence layer.

store_size
Calculated the store size on disk.

lakesuperior.store.ldp_nr.default_layout module

class lakesuperior.store.ldp_nr.default_layout.**DefaultLayout**(**args*, ***kwargs*)
Bases: *lakesuperior.store.ldp_nr.base_non_rdf_layout.BaseNonRdfLayout*

Default file layout.

This is a simple filesystem layout that stores binaries in pairtree folders in a local filesystem. Parameters can be specified for the

bootstrap()
Initialize binary file store.

delete(*uuid*)
See BaseNonRdfLayout.delete.

static local_path(*root*, *uuid*, *bl=4*, *bc=4*)
Generate the resource path splitting the resource checksum according to configuration parameters.

Parameters **uuid** (*str*) – The resource UUID. This corresponds to the content checksum.

persist(*uid*, *stream*, *bufsize=8192*, *prov_cksum=None*, *prov_cksum_algo=None*)
Store the stream in the file system.

This method handles the file in chunks. for each chunk it writes to a temp file and adds to a checksum. Once the whole file is written out to disk and hashed, the temp file is moved to its final location which is determined by the hash value.

Parameters

- **uid** (*str*) – UID of the resource.
- **stream** (*IOstream*) – file-like object to persist.
- **bufsize** (*int*) – Chunk size. 2**12 to 2**15 is a good range.
- **prov_cksum** (*str*) – Checksum provided by the client to verify that the content received matches what has been sent. If None (the default) no verification will take place.
- **prov_cksum_algo** (*str*) – Verification algorithm to validate the integrity of the user-provided data. If this is different from the default hash algorithm set in the application configuration, which is used to calclate the checksum of the file for storing purposes, a separate hash is calculated specifically for validation purposes. Clearly it's more efficient to use the same algorithm and avoid a second checksum calculation.

lakesuperior.store.ldp_rs package

Submodules

lakesuperior.store.ldp_rs.lmdb_store module

class lakesuperior.store.ldp_rs.lmdb_store.**LmdbStore**(*path*, *identifier=None*, *create=True*)

Bases: *lakesuperior.store.ldp_rs.lmdb_triplestore.LmdbTriplestore*, rdflib.store.Store

LMDB-backed store.

This is an implementation of the RDFLib Store interface: https://github.com/RDFLib/rdflib/blob/master/rdflib/store.py

Handles the interaction with a LMDB store and builds an abstraction layer for triples.

This store class uses two LMDB environments (i.e. two files): one for the main (preservation-worthy) data and the other for the index data which can be rebuilt from the main database.

There are 4 main data sets (preservation worthy data):

- `t:st` (term key: serialized term; 1:1)
- `spo:c` (joined S, P, O keys: context key; dupsort, dupfixed)
- `c:` (context keys only, values are the empty bytestring; 1:1)
- `pfx:ns` (prefix: pickled namespace; 1:1)

And 6 indices to optimize lookup for all possible bound/unbound term combination in a triple:

- `th:t` (term hash: term key; 1:1)
- `s:po` (S key: joined P, O keys; dupsort, dupfixed)
- `p:so` (P key: joined S, O keys; dupsort, dupfixed)
- `o:sp` (O key: joined S, P keys; dupsort, dupfixed)
- `c:spo` (context → triple association; dupsort, dupfixed)
- `ns:pfx` (pickled namespace: prefix; 1:1)

The default graph is defined in `rdflib.graph.RDFLIB_DEFAULT_GRAPH_URI`. Adding triples without context will add to this graph. Looking up triples without context (also in a SPARQL query) will look in the union graph instead of in the default graph. Also, removing triples without specifying a context will remove triples from all contexts.

bind(*prefix*, *namespace*)

Bind a prefix to a namespace.

Parameters

- **prefix** (*str*) – Namespace prefix.
- **namespace** (*rdflib.URIRef*) – Fully qualified URI of namespace.

close(*commit_pending_transaction=False*)

Close the database connection.

Do this at server shutdown.

context_aware = True

formula_aware = False

graph_aware = True

namespace(*prefix*)
Get the namespace for a prefix. :param str prefix: Namespace prefix.

namespaces()
Get an iterator of all prefix: namespace bindings.

Return type Iterator(tuple(str, rdflib.Namespace))

open(*configuration=None*, *create=True*)
Open the store environment.

Parameters

- **configuration** (*str*) – If not specified on init, indicate the path to use for the store.
- **create** (*bool*) – Create the file and folder structure for the store environment.

prefix(*namespace*)
Get the prefix associated with a namespace.

Note: A namespace can be only bound to one prefix in this implementation.

Parameters **namespace** (*rdflib.Namespace*) – Fully qualified namespace.

Return type str or None

remove(*triple_pattern*, *context=None*)
Remove triples by a pattern.

Parameters

- **triple_pattern** (*tuple*) – 3-tuple of either RDF terms or None, indicating the triple(s) to be removed. `None` is used as a wildcard.
- **context** (*rdflib.term.Identifier or None*) – Context to remove the triples from. If None (the default) the matching triples are removed from all contexts.

remove_graph(*graph*)
Remove all triples from graph and the graph itself.

Parameters **graph** (*rdflib.URIRef*) – URI of the named graph to remove.

transaction_aware = True

lakesuperior.store.ldp_rs.lmdb_triplestore module

class lakesuperior.store.ldp_rs.lmdb_triplestore.**LmdbTriplestore**
Bases: *lakesuperior.store.base_lmdb_store.BaseLmdbStore*

Low-level triplestore layer.

This class extends the general-purpose `BaseLmdbStore` and maps triples and contexts to key-value records in LMDB. It can be used in the application context (`env.app_globals.rdf_store`), or an independent instance can be spun up in an arbitrary disk location.

This class provides the base for the RDFlib-compatible backend in the *lakesuperior.store.ldp_rs.lmdb_store.LmdbStore*.

add
Add a triple and start indexing.

Parameters

- **triple** (*tuple(rdflib.Identifier)*) – Tuple of three identifiers.

- **context** (*rdflib.Identifier or None*) – Context identifier. `None` inserts in the default graph.
- **quoted** (*bool*) – Not used.

add_graph
Add a graph (context) to the database.

This creates an empty graph by associating the graph URI with the pickled *None* value. This prevents from removing the graph when all triples are removed.

Parameters **graph** (*rdflib.URIRef*) – URI of the named graph to add.

all_namespaces
Return all registered namespaces.

all_terms
Return all terms of a type (`s`, `p`, or `o`) in the store.

contexts
Get a list of all contexts.

Return type set(URIRef)

dbi_flags = {b'c:_____': 10, b'c:spo__': 94, b'o:sp___': 94, b'p:so___': 94, b'po:

dbi_labels = [b't:st___', b'spo:c__', b'c:_____', b'pfx:ns_', b'ns:pfx_', b'th:t___',

flags = 0

options = {'map_size': 1099511627776}

stats
Gather statistics about the database.

triple_keys
Top-level lookup method.

This method is used by *triples* which returns native Python tuples, as well as by other methods that need to iterate and filter triple keys without incurring in the overhead of converting them to triples.

Parameters

- **triple_pattern** (*tuple*) – 3 RDFLib terms
- **context** (*rdflib.term.Identifier or None*) – Context graph or URI, or None.

triples
Generator over matching triples.

Parameters

- **triple_pattern** (*tuple*) – 3 RDFLib terms
- **context** (*rdflib.Graph or None*) – Context graph, if available.

Return type Iterator

Returns

Generator over triples and contexts in which each result has the following format:

```
(s, p, o), generator(contexts)
```

Where the contexts generator lists all context that the triple appears in.

lakesuperior.store.ldp_rs.rsrc_centric_layout module

class lakesuperior.store.ldp_rs.rsrc_centric_layout.**RsrcCentricLayout**(*config*)
Bases: object

This class exposes an interface to build graph store layouts. It also provides the basics of the triplestore connection.

Some store layouts are provided. New ones aimed at specific uses and optimizations of the repository may be developed by extending this class and implementing all its abstract methods.

A layout is implemented via application configuration. However, once contents are ingested in a repository, changing a layout will most likely require a migration.

The custom layout must be in the lakesuperior.store.rdf package and the class implementing the layout must be called *StoreLayout*. The module name is the one defined in the app configuration.

E.g. if the configuration indicates *simple_layout* the application will look for *lakesuperior.store.rdf.simple_layout.SimpleLayout*.

ask_rsrc_exists(*uid*)
See base_rdf_layout.ask_rsrc_exists.

attr_map = {Namespace('info:fcsystem/graph/admin'): {'p': {rdflib.term.URIRef('http:
Human-manageable map of attribute routes.

This serves as the source for *attr_routes*.

attr_routes
This is a map that allows specific triples to go to certain graphs. It is a machine-friendly version of the static attribute *attr_map* which is formatted for human readability and to avoid repetition. The attributes not mapped here (usually user-provided triples with no special meaning to the application) go to the *fcmain:* graph.

The output of this is a dict with a similar structure:

```
{
    'p': {
        <Predicate P1>: <destination graph G1>,
        <Predicate P2>: <destination graph G1>,
        <Predicate P3>: <destination graph G1>,
        <Predicate P4>: <destination graph G2>,
        [...]
    },
    't': {
        <RDF Type T1>: <destination graph G1>,
        <RDF Type T2>: <destination graph G3>,
        [...]
    }
}
```

bootstrap()
Delete all graphs and insert the basic triples.

count_rsrc()
Return a count of first-class resources, subdivided in "live" and historic snapshots.

delete_rsrc(*uid*, *historic=False*)
Delete all aspect graphs of an individual resource.

Parameters

- **uid** – Resource UID.
- **historic** (*bool*) – Whether the UID is of a historic version.

find_refint_violations()
Find all referential integrity violations.

This method looks for dangling relationships within a repository by checking the objects of each triple; if the object is an in-repo resource reference, and no resource with that URI results to be in the repo, that triple is reported.

Return type set

Returns Triples referencing a repository URI that is not a resource.

forget_rsrc(*uid*, *inbound=True*, *children=True*)
Completely delete a resource and (optionally) its children and inbound references.

NOTE: inbound references in historic versions are not affected.

get_descendants(*uid*, *recurse=True*)
Get descendants (recursive children) of a resource.

Parameters **uid** (*str*) – Resource UID.

Return type Iterator(rdflib.URIRef)

Returns Subjects of descendant resources.

get_imr(*uid*, *ver_uid=None*, *strict=True*, *incl_inbound=False*, *incl_children=True*, ***kwargs*)
See base_rdf_layout.get_imr.

get_inbound_rel(*subj_uri*, *full_triple=True*)
Query inbound relationships for a subject.

This can be a list of either complete triples, or of subjects referring to the given URI. It excludes historic version snapshots.

Parameters

- **subj_uri** (*rdflib.URIRef*) – Subject URI.
- **full_triple** (*boolean*) – Whether to return the full triples found or only the subjects. By default, full triples are returned.

Return type Iterator(tuple(rdflib.term.Identifier) or rdflib.URIRef)

Returns Inbound triples or subjects.

get_last_version_uid(*uid*)
Get the UID of the last version of a resource.

This can be used for tombstones too.

get_metadata(*uid*, *ver_uid=None*, *strict=True*)
This is an optimized query to get only the administrative metadata.

get_raw(*subject*, *ctx=None*)
Get a raw graph of a non-LDP resource.

The graph is queried across all contexts or within a specific one.

Parameters

- **subject** (*rdflib.term.URIRef*) – URI of the subject.

- **ctx** (*rdflib.term.URIRef*) – URI of the optional context. If None, all named graphs are queried.

Return type *Graph*

get_user_data(*uid*)
Get all the user-provided data.

Parameters **uid** (*string*) – Resource UID.

Return type rdflib.Graph

get_version_info(*uid*)
Get all metadata about a resource's versions.

Parameters **uid** (*string*) – Resource UID.

Return type *Graph*

graph_ns_types = {Namespace('info:fcsystem/graph/admin'): rdflib.term.URIRef('info:fc
RDF types of graphs by prefix.

ignore_vmeta_preds = {rdflib.term.URIRef('http://xmlns.com/foaf/0.1/primaryTopic')}
Predicates of version metadata to be ignored in output.

ignore_vmeta_types = {rdflib.term.URIRef('info:fcsystem/AdminGraph'), rdflib.term.URIR
RDF types of version metadata to be ignored in output.

modify_rsrc(*uid*, *remove_trp={}*, *add_trp={}*)
Modify triples about a subject.

This method adds and removes triple sets from specific graphs, indicated by the term router. It also adds metadata about the changed graphs.

patch_rsrc(*uid*, *qry*)
Patch a resource with SPARQL-Update statements.

The statement(s) is/are executed on the user-provided graph only to ensure that the scope is limited to the resource.

Parameters

- **uid** (*str*) – UID of the resource to be patched.
- **qry** (*dict*) – Parsed and translated query, or query string.

raw_query(*qry_str*)
Perform a straight query to the graph store.

snapshot_uid(*uid*, *ver_uid*)
Create a versioned UID string from a main UID and a version UID.

truncate_rsrc(*uid*)
Remove all user-provided data from a resource and only leave admin and structure data.

uri_to_uid(*uri*)
Convert an internal URI to a UID.

Submodules

lakesuperior.store.base_lmdb_store module

class lakesuperior.store.base_lmdb_store.**BaseLmdbStore**(*env_path*, *open_env=True*, *create=True*)

Bases: object

Generic LMDB store abstract class.

This class contains convenience method to create an LMDB store for any purpose and provides some convenience methods to wrap cursors and transactions into contexts.

Example usage:

```
>>> class MyStore(BaseLmdbStore):
...     path = '/base/store/path'
...     dbi_flags = ('db1', 'db2')
...
>>> ms = MyStore()
>>> # "with" wraps the operation in a transaction.
>>> with ms.cur(index='db1', write=True):
...     cur.put(b'key1', b'val1')
True
```

abort
Abort main transaction.

begin
Begin a transaction manually if not already in a txn context.

The *txn_ctx()* context manager should be used whenever possible rather than this method.

close_env

commit
Commit main transaction.

dbi_flags = {}

dbi_labels = []

delete
Delete one single value by key. Python-facing method.

destroy
Destroy the store.

https://www.youtube.com/watch?v=lIVq7FJnPwg

Parameters **_path** (*str*) – unused. Left for RDFLib API compatibility. (actually quite dangerous if it were used: it could turn into a general-purpose recursive file and folder delete method!)

env_flags = 0

env_path

env_perms = 416

get_data
Get a single value (non-dup) for a key (Python-facing method).

is_open

is_txn_open

is_txn_rw

key_exists
Return whether a key exists in a database (Python-facing method).

Wrap in a new transaction. Only use this if a transaction has not been opened.

open_env
Open, and optionally create, store environment.

options = {}

put
Put one key/value pair (Python-facing method).

readers

readers_mult = 4

rollback
Alias for *abort()*

stats
Gather statistics about the database.

txn_ctx(*self*, *write=False*)
Transaction context manager.

Open and close a transaction for the duration of the functions in the context. If a transaction has already been opened in the store, a new one is opened only if the current transaction is read-only and the new requested transaction is read-write.

If a new write transaction is opened, the old one is kept on hold until the new transaction is closed, then restored. All cursors are invalidated and must be restored as well if one needs to reuse them.

Parameters **write** (*bool*) – Whether a write transaction is to be opened.

Return type lmdb.Transaction

txn_id

exception lakesuperior.store.base_lmdb_store.**InvalidParamError**
Bases: *lakesuperior.store.base_lmdb_store.LmdbError*

exception lakesuperior.store.base_lmdb_store.**KeyExistsError**
Bases: *lakesuperior.store.base_lmdb_store.LmdbError*

exception lakesuperior.store.base_lmdb_store.**KeyNotFoundError**
Bases: *lakesuperior.store.base_lmdb_store.LmdbError*

exception lakesuperior.store.base_lmdb_store.**LmdbError**
Bases: Exception

lakesuperior.store.metadata_store module

class lakesuperior.store.metadata_store.**MetadataStore**
Bases: *lakesuperior.store.base_lmdb_store.BaseLmdbStore*

LMDB store for RDF metadata.

Note that even though this store connector uses LMDB as the :py:LmdbStore class, it is separate because it is not part of the RDFLib store implementation and carries higher-level concepts such as LDP resource URIs.

dbi_labels = ['checksums', 'event_queue']
Currently implemented:

- `checksums`: registry of LDP resource graphs, indicated in the key by their UID, and their cryptographic hashes.

delete_checksum(*uri*)
Delete the stored checksum of a resource.

Parameters **uri** (*str*) – Resource URI (`info:fcres...`).

get_checksum(*uri*)
Get the checksum of a resource.

Parameters **uri** (*str*) – Resource URI (`info:fcres...`).

Return type bytes

update_checksum(*uri*, *cksum*)
Update the stored checksum of a resource.

Parameters

- **uri** (*str*) – Resource URI (`info:fcres...`).
- **cksum** (*bytes*) – Checksum bytestring.

lakesuperior.util package

Submodules

lakesuperior.util.benchmark module

lakesuperior.util.generators module

lakesuperior.util.ingest_random_image module

lakesuperior.util.locustfile module

lakesuperior.util.toolbox module

Utility to translate and generate strings and other objects.

class lakesuperior.util.toolbox.**RequestUtils**
Bases: object

Utilities that require access to an HTTP request context.

Initialize this within a Flask request context.

globalize_graph(*gr*)
Globalize a graph.

globalize_imr(*imr*)
Globalize an Imr.

Return type rdflib.Graph

globalize_rsrc(*rsrc*)
Globalize a resource.

globalize_string(*s*)
Convert URNs into URIs in a string using the application base URI.

Parameters **s** (*string*) – Input string.

Return type string

globalize_term(*urn*)
Convert an URN into an URI using the application base URI.

Parameters **urn** (*rdflib.URIRef*) – Input URN.

Return type rdflib.URIRef

globalize_triple(*trp*)
Globalize terms in a triple.

Parameters **trp** (*tuple(rdflib.URIRef)*) – The triple to be converted

Return type tuple(rdflib.URIRef)

localize_ext_str(*s*, *urn*)
Convert global URIs to local in a SPARQL or RDF string.

Also replace empty URIs (<>) with a fixed local URN and take care of fragments and relative URIs.

This is a 3-pass replacement. First, global URIs whose webroot matches the application ones are replaced with internal URIs. Then, relative URIs are converted to absolute using the internal URI as the base; finally, the root node is appropriately addressed.

localize_graph(*gr*)
Localize a graph.

localize_payload(*data*)
Localize an RDF stream with domain-specific URIs.

Parameters **data** (*bytes*) – Binary RDF data.

Return type bytes

localize_term(*uri*)
Localize an individual term.

Parameters **rdflib.URIRef** – urn Input URI.

Return type rdflib.URIRef

localize_triple(*trp*)
Localize terms in a triple.

Parameters **trp** (*tuple(rdflib.URIRef)*) – The triple to be converted

Return type tuple(rdflib.URIRef)

localize_uri_string(*s*)
Convert URIs into URNs in a string using the application base URI.

Parameters **str** – s Input string.

Return type str

uid_to_uri(*uid*)
Convert a UID to a URI.

Return type rdflib.URIRef

uri_to_uid(*uri*)
Convert an absolute URI (internal or external) to a UID.

Return type str

lakesuperior.util.toolbox.**fsize_fmt**(*num*, *suffix='b'*)
Format an integer into 1024-block file size format.

Adapted from Python 2 code on https://stackoverflow.com/a/1094933/3758232

Parameters

- **num** (*int*) – Size value in bytes.
- **suffix** (*str*) – Suffix label (defaults to `b`).

Return type str

Returns Formatted size to largest fitting unit.

lakesuperior.util.toolbox.**get_tree_size**(*path*, *follow_symlinks=True*)
Return total size of files in given path and subdirs.

Ripped from https://www.python.org/dev/peps/pep-0471/

lakesuperior.util.toolbox.**parse_rfc7240**(*h_str*)
Parse `Prefer` header as per https://tools.ietf.org/html/rfc7240

The `cgi.parse_header` standard method does not work with all possible use cases for this header.

Parameters **h_str** (*str*) – The header(s) as a comma-separated list of Prefer statements, excluding the `Prefer:` token.

lakesuperior.util.toolbox.**rel_uri_to_urn**(*uri*, *uid*)
Convert a URIRef with a relative location (e.g. <>) to an URN.

Parameters

- **uri** (*URIRef*) – The URI to convert.
- **uid** (*str*) – Resource UID that the URI should be relative to.

Returns Converted URN if the input is relative, otherwise the unchanged URI.

Return type URIRef

lakesuperior.util.toolbox.**rel_uri_to_urn_string**(*string*, *uid*)
Convert relative URIs in a SPARQL or RDF string to internal URNs.

Parameters **string** (*str*) – Input string.

:param str uid Resource UID to build the base URN from.

Return type str

Returns Modified string.

lakesuperior.util.toolbox.**replace_term_domain**(*term*, *search*, *replace*)
Replace the domain of a term.

Parameters

- **term** (*rdflib.URIRef*) – The term (URI) to change.
- **search** (*str*) – Domain string to replace.
- **replace** (*str*) – Domain string to use for replacement.

Return type rdflib.URIRef

lakesuperior.util.toolbox.**split_uuid**(*uuid*)
Split a UID into pairtree segments. This mimics FCREPO4 behavior.

Parameters **uuid** (*str*) – UUID to split.

Return type str

lakesuperior.wsgi package

GUnicorn WSGI configuration.

GUnicorn reads configuration options from this file by importing it:

```
gunicorn -c python:lakesuperior.wsgi lakesuperior.server:fcrepo
```

This module reads the `gunicorn.yml` configuration and overrides defaults set here. Only some of the GUnicorn optionscan be changed: others have to be set to specific values in order for Lakesuperior to work properly.

Submodules

lakesuperior.app module

lakesuperior.app.**create_app**(*app_conf*)
App factory.

Create a Flask app.

@param app_conf (dict) Configuration parsed from *application.yml* file.

lakesuperior.config_parser module

lakesuperior.config_parser.**default_config_dir = '/home/docs/checkouts/readthedocs.org/user_**
Default configuration directory.

This value falls back to the provided `etc.defaults` directory if the `FCREPO_CONFIG_DIR` environment variable is not set.

This value can still be overridden by custom applications by passing the `config_dir` value to *parse_config()* explicitly.

lakesuperior.config_parser.**parse_config**(*config_dir=None*)
Parse configuration from a directory.

This is normally called by the standard endpoints (`lsup_admin`, web server, etc.) or by a Python client by importing *lakesuperior.env_setup* but an application using a non-default configuration may specify an alternative configuration directory.

The directory must have the same structure as the one provided in `etc.defaults`.

Parameters **config_dir** – Location on the filesystem of the configuration directory. The default is set by the `FCREPO_CONFIG_DIR` environment variable or, if this is not set, the `etc.defaults` stock directory.

lakesuperior.env_setup module

Default configuration.

Import this module to initialize the configuration for a production setup:

```
>>> import lakesuperior.env_setup
```

Will load the default configuration.

lakesuperior.exceptions module

Put all exceptions here.

exception lakesuperior.exceptions.**ChecksumValidationError**(*uid*, *prov_cksum*, *calc_cksum*, *msg=None*)

Bases: *lakesuperior.exceptions.ResourceError*

Raised in an attempt to create a resource a URI that already exists and is not supposed to.

This usually surfaces at the HTTP level as a 409.

exception lakesuperior.exceptions.**IncompatibleLdpTypeError**(*uid*, *mimetype*, *msg=None*)

Bases: *lakesuperior.exceptions.ResourceError*

Raised when a LDP-NR resource is PUT in place of a LDP-RS and vice versa.

This usually surfaces at the HTTP level as a 415.

exception lakesuperior.exceptions.**IndigestibleError**(*uid*, *msg=None*)

Bases: *lakesuperior.exceptions.ResourceError*

Raised when an unsupported digest algorithm is requested.

exception lakesuperior.exceptions.**InvalidResourceError**(*uid*, *msg=None*)

Bases: *lakesuperior.exceptions.ResourceError*

Raised when an invalid resource is found.

This usually surfaces at the HTTP level as a 409 or other error.

exception lakesuperior.exceptions.**InvalidTripleError**(*t*)

Bases: RuntimeError

Raised when a triple in a delta is not valid.

This does not necessarily that it is not valid RDF, but rather that it may not be valid for the context it is meant to be utilized.

exception lakesuperior.exceptions.**PathSegmentError**(*uid*, *msg=None*)

Bases: *lakesuperior.exceptions.ResourceError*

Raised when a LDP-NR resource is a path segment.

This may be an expected result and may be handled to return a 200.

exception lakesuperior.exceptions.**RefIntViolationError**(*o*)

Bases: RuntimeError

Raised when a provided data set has a link to a non-existing repository resource. With some setups this is handled silently, with a strict setting it raises this exception that should return a 412 HTTP code.

exception lakesuperior.exceptions.**ResourceError**(*uid*, *msg=None*)
Bases: RuntimeError

Raised in an attempt to create a resource a URI that already exists and is not supposed to.

This usually surfaces at the HTTP level as a 409.

exception lakesuperior.exceptions.**ResourceExistsError**(*uid*, *msg=None*)
Bases: *lakesuperior.exceptions.ResourceError*

Raised in an attempt to create a resource a URI that already exists and is not supposed to.

This usually surfaces at the HTTP level as a 409.

exception lakesuperior.exceptions.**ResourceNotExistsError**(*uid*, *msg=None*)
Bases: *lakesuperior.exceptions.ResourceError*

Raised in an attempt to create a resource a URN that does not exist and is supposed to.

This usually surfaces at the HTTP level as a 404.

exception lakesuperior.exceptions.**ServerManagedTermError**(*terms*, *term_type=None*)
Bases: RuntimeError

Raised in an attempt to change a triple containing a server-managed term.

This usually surfaces at the HTTP level as a 409 or other error.

exception lakesuperior.exceptions.**SingleSubjectError**(*uid*, *subject*)
Bases: RuntimeError

Raised when a SPARQL-Update query or a RDF payload for a PUT contain subjects that do not correspond to the resource being operated on.

exception lakesuperior.exceptions.**TombstoneError**(*uid*, *ts*)
Bases: RuntimeError

Raised when a tombstone resource is found.

It is up to the caller to handle this which may be a benign and expected result.

lakesuperior.globals module

class lakesuperior.globals.**AppGlobals**(*config*)
Bases: object

Application Globals.

This class is instantiated and used as a carrier for all connections and various global variables outside of the Flask app context.

The variables are set on initialization by passing a configuration dict. Usually this is done when starting an application. The instance with the loaded variables is then assigned to the *lakesuperior.env* global variable.

You can either load the default configuration:

```
>>>from lakesuperior import env_setup
```

Or set up an environment with a custom configuration:

```
>>> from lakesuperior import env
>>> from lakesuperior.app_globals import AppGlobals
>>> my_config = {'name': 'value', '...': '...'}
>>> env.app_globals = AppGlobals(my_config)
```

camelcase(*word*)
Convert a string with underscores to a camel-cased one.

Ripped from https://stackoverflow.com/a/6425628

changelog

config
Global configuration.

This is a collection of all configuration options **except** for the WSGI configuration which is initialized at a different time and is stored under `lakesuperior.env.wsgi_options`.

TODO: Update class reference when interface will be separated from implementation.

messenger
Current message handler.

This is an instance of *Messenger*.

nonrdfly
Current non-RDF (binary contents) layout.

This is an instance of *BaseNonRdfLayout*.

rdf_store
Current RDF low-level store.

This is an instance of *LmdbStore*.

rdfly
Current RDF layout.

This is an instance of *RsrcCentricLayout*.

TODO: Update class reference when interface will be separated from implementation.

lakesuperior.globals.**RES_CREATED = '_create_'**
A resource was created.

lakesuperior.globals.**RES_DELETED = '_delete_'**
A resource was deleted.

lakesuperior.globals.**RES_UPDATED = '_update_'**
A resource was updated.

lakesuperior.globals.**ROOT_RSRC_URI = rdflib.term.URIRef('info:fcres/')**
Internal URI of root resource.

lakesuperior.globals.**ROOT_UID = '/'**
Root node UID.

lakesuperior.lsup_admin module

Utility to perform core maintenance tasks via console command-line.

The command-line tool is self-documented. Type:

```
lsup-admin --help
```

for a list of tools and options.

lakesuperior.migrator module

class lakesuperior.migrator.**Migrator**(*src*, *dest*, *src_auth=(None, None)*, *clear=False*, *zero_binaries=False*, *compact_uris=False*, *skip_errors=False*)

Bases: object

Class to handle a database migration.

This class holds state of progress and shared variables as it crawls through linked resources in an LDP server.

Since a repository migration can be a very long operation but it is impossible to know the number of the resources to gather by LDP interaction alone, a progress ticker outputs the number of processed resources at regular intervals.

db_params = {'map_size': 1099511627776, 'meminit': False, 'metasync': False, 'readal

LMDB database parameters.

See lmdb.Environment.__init__()

ignored_preds = (rdflib.term.URIRef('http://fedora.info/definitions/v4/repository#hasP

List of predicates to ignore when looking for links.

migrate(*start_pts=None*, *list_file=None*)

Migrate the database.

This method creates a fully functional and configured Lakesuperior data set contained in a folder from an LDP repository.

Parameters

- **start_pts** (*tuple or list*) – List of starting points to retrieve resources from. It would typically be the repository root in case of a full dump or one or more resources in the repository for a partial one.
- **list_file** (*str*) – path to a local file containing a list of URIs, one per line.

class lakesuperior.migrator.**StoreWrapper**(*store*)

Bases: contextlib.ContextDecorator

Open and close a store.

lakesuperior.profiler module

lakesuperior.profiler.**run**()

lakesuperior.server module

lakesuperior.server.**run**()

1.14 Lakesuperior Architecture

Lakesuperior is written in Python and Cython; the latter for lower-level components that interface with C basic data structures for maximum efficiency.

Aside from an optional dependency on a message queue server, Lakesuperior aims at being self-contained. All persistence is done on an embedded database. This allows a minimum memory and CPU footprint, and a high degree of scalability, from *single-board computers* to multi-core, high-load servers.

Inefficient applications "get the job done" by burning through CPU cycles, memory, storage and electricity, and spew out great amounts of carbon and digits on cloud provider bills. Lakesuperior strives to be mindful of that.

1.14.1 Multi-Modal Access

Lakesuperior services and data are accessible in multiple ways:

- Via HTTP. This is the canonical way to interact with LDP resources and conforms quite closely to the Fedora specs (currently v4).
- Via command line. This method includes long-running admin tasks which are not available via HTTP.
- Via a Python API. This method allows to use Python scripts to access the same methods available to the two methods above in a programmatic way. It is possible to write Python plugins or even to embed Lakesuperior in a Python application, even without running a web server. Also, only this way it is possible to access some of the lower-level application layers that allow to skirt much heavy-handed data processing.

1.14.2 Architecture Overview

The Lakesuperior REST API provides access to the underlying Python API. All REST and CLI operations can be replicated by a Python program accessing this API.

The main advantage of the Python API is that it makes it very easy to maipulate graph and binary data without the need to serialize or deserialize native data structures. This matters when handling large ETL jobs for example.

The Python API is divided in three main areas:

- Resource API: this API in charge of all the resource CRUD operations and implements the majority of the Fedora specs.
- Admin API: exposes utility methods, mostly long-running maintenance jobs.
- Query API: provides several facilities for querying repository data.

See *API documentation* for more details.

1.15 Performance Benchmark Report

The purpose of this document is to provide very broad performance measurements and comparison between Lakesuperior and Fedora/Modeshape implementations.

1.15.1 Environment

Hardware

- MacBook Pro14,2

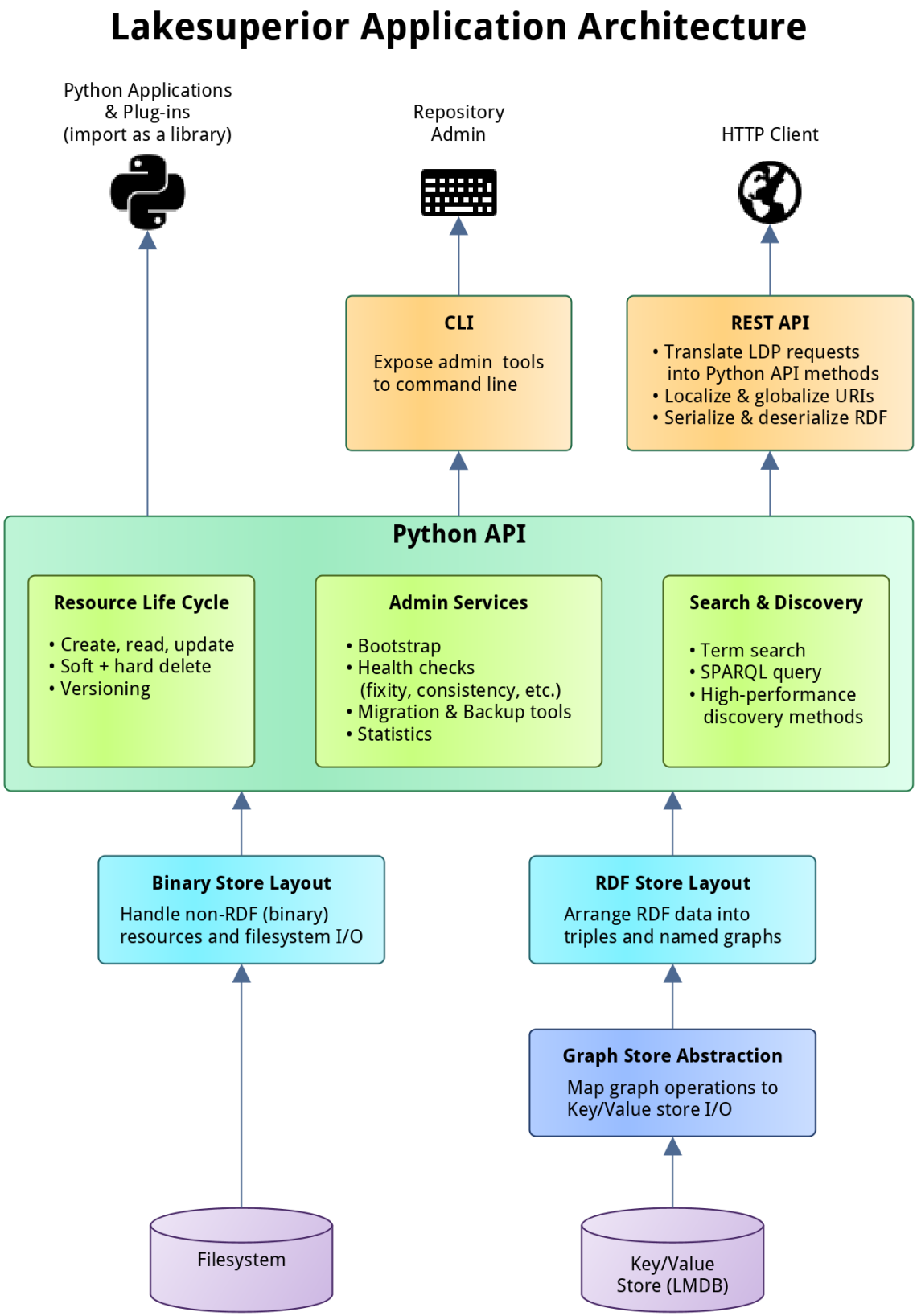

Fig. 3: Lakesuperior Architecture

- 1x Intel(R) Core(TM) i5 @3.1Ghz
- 16Gb RAM
- SSD
- OS X 10.13
- Python 3.7.2
- lmdb 0.9.22

Benchmark script

Source code

The script was run by generating 100,000 children under the same parent. PUT and POST requests were tested separately. The POST method produced pairtrees in Fedora to counter its known issue with many resources as direct children of a container.

The script calculates only the timings used for the PUT or POST requests, not counting the time used to generate the random data.

Data Set

Synthetic graph created by the benchmark script. The graph is unique for each request and consists of 200 triples which are partly random data, with a consistent size and variation:

- 50 triples have an object that is a URI of an external resource (50 unique predicates; 5 unique objects).
- 50 triples have an object that is a URI of a repository-managed resource (50 unique predicates; 5 unique objects).
- 100 triples have an object that is a 64-character random Unicode string (50 unique predicates; 100 unique objects).

The benchmark script is also capable of generating random binaries and a mix of binary and RDF resources; a large-scale benchmark, however, was impractical at the moment due to storage constraints.

LDP Data Retrieval

REST API request:

```
time curl http://localhost:8000/ldp/pomegranate > /dev/null
```

SPARQL Query

The following query was used against the repository after the 100K resource ingest:

```
PREFIX ldp: <http://www.w3.org/ns/ldp#>
SELECT (COUNT(?s) AS ?c) WHERE {
  ?s a ldp:Resource .
  ?s a ldp:Container .
}
```

Raw request:

```
time curl -iXPOST -H'Accept:application/sparql-results+json' \
-H'Content-Type:application/x-www-form-urlencoded; charset=UTF-8' \
-d 'query=PREFIX+ldp:+<http://www.w3.org/ns/ldp#> SELECT+(COUNT(?s)+AS+?c)'\
'+WHERE+{ ++?s+a+ldp:Resource+. ++?s+a+ldp:Container+. }+' \
http://localhost:5000/query/sparql
```

Python API Retrieval

In order to illustrate the advantages of the Python API, a sample retrieval of the container resource after the load has been timed. This was done in an IPython console:

```
In [1]: from lakesuperior import env_setup
In [2]: from lakesuperior.api import resource as rsrc_api
In [3]: %timeit x = rsrc_api.get('/pomegranate').imr.as_rdflib()
```

1.15.2 Results

Software	PUT	POST	Store Size	GET	SPARQL Query
FCREPO 5.0.2	>500ms[1]	65ms (100%)[2]	12Gb (100%)	3m41s (100%)	N/A
Lakesuperior REST	104ms (100%)	123ms (189%)	8.7Gb (72%)	30s (14%)	19.3s (100%)
Lakesuperior Python	69ms (60%)	58ms (89%)	8.7Gb (72%)	6.7s (3%)[3][4]	9.17s (47%)

1.15.3 Charts

1.15.4 Conclusions

Lakesuperior appears to be slower on writes and much faster on reads than Fedora 4-5. Both these factors are very likely related to the underlying LMDB store which is optimized for read performance. The write performance gap is more than filled when ingesting via the Python API.

In a real-world application scenario, in which a client may perform multiple reads before and after storing resources, the write performance gap may decrease. A Python application using the Python API for querying and writing would experience a dramatic improvement in read as well as write timings.

As it may be obvious, these are only very partial and specific results. They should not be taken as a thorough performance assessment, but rather as a starting point to which specific use-case variables may be added.

[1] POST was stopped at 30K resources after the ingest time reached >1s per resource. This is the manifestation of the "many members" issue which is visible in the graph below. The "Store" value is for the PUT operation which ran regularly with 100K resources.

[2] the POST test with 100K resources was conducted with fedora 4.7.5 because 5.0 would not automatically create a pairtree, thereby resulting in the same performance as the PUT method.

[3] Timing based on a warm cache. The first query timed at 22.2s.

[4] The Python API time for the "GET request" (retrieval) without the conversion to Python in alpha20 is 3.2 seconds, versus the 6.7s that includes conversion to Python/RDFlib objects. This can be improved by using more efficient libraries that allow serialization and deserialization of RDF.

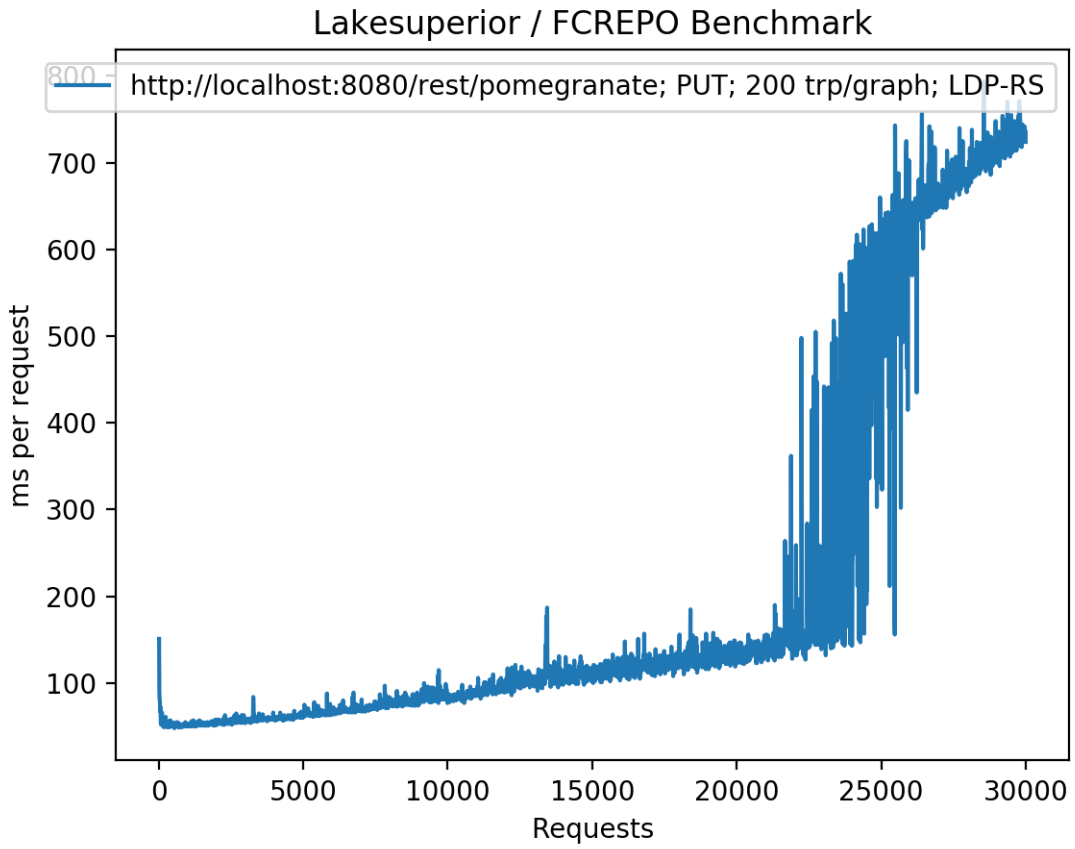

Fig. 4: Fedora/Modeshape using PUT requests under the same parent. The "many members" issue is clearly visible after a threshold is reached.

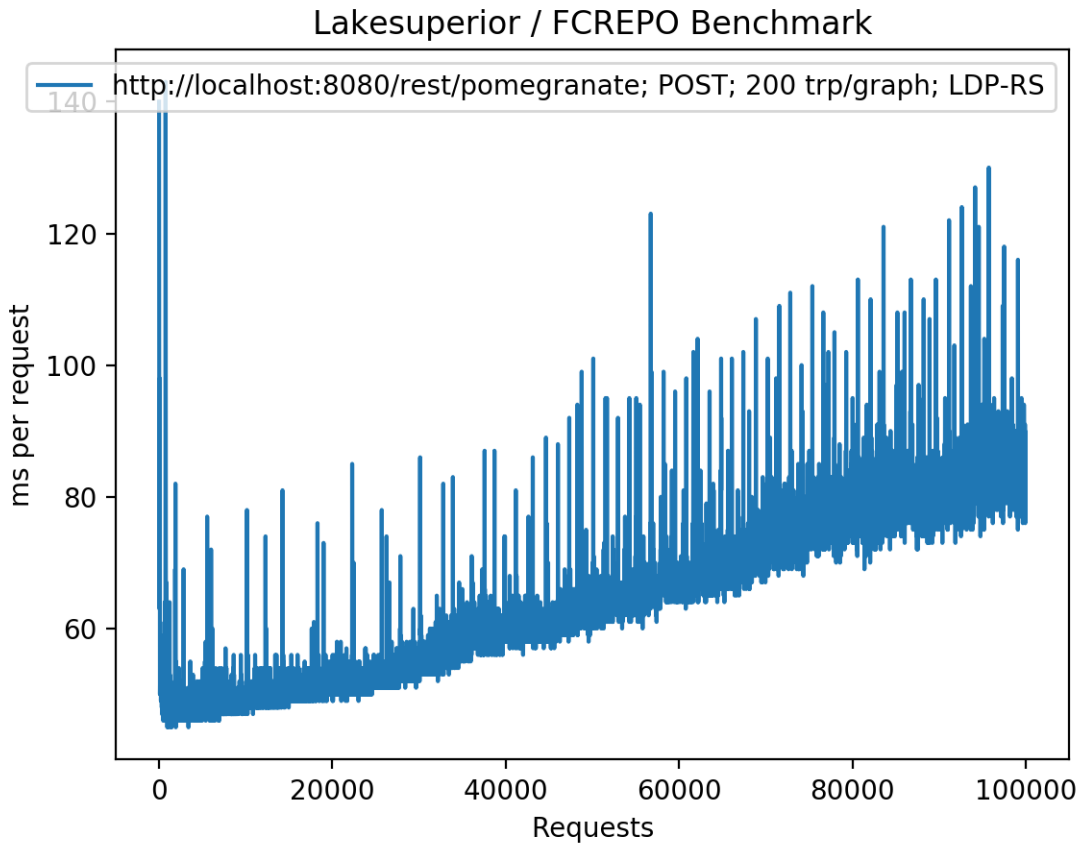

Fig. 5: Fedora/Modeshape using POST requests generating pairtrees. The performance is greatly improved, however the ingest time increases linearly with the repository size (O(n) time complexity)

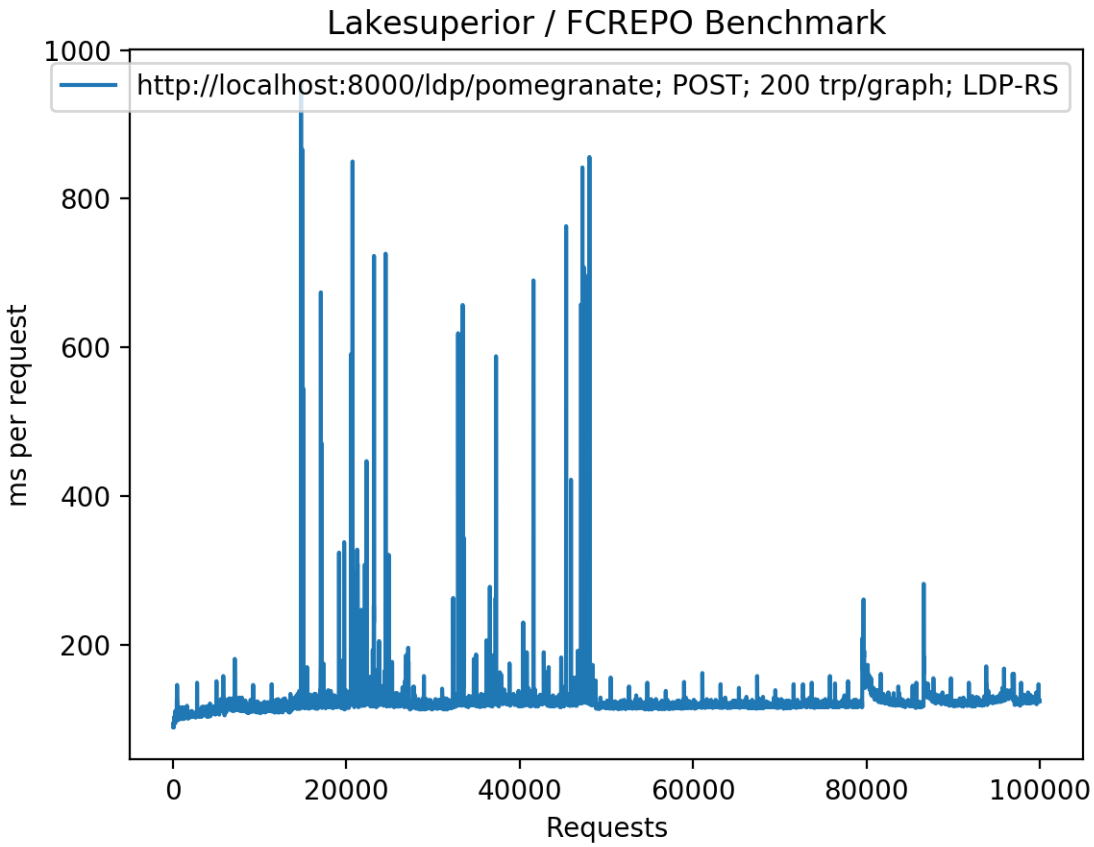

Fig. 6: Lakesuperior using POST requests, NOT generating pairtrees (equivalent to a PUT request). The timing increase is closer to a O(log n) pattern.

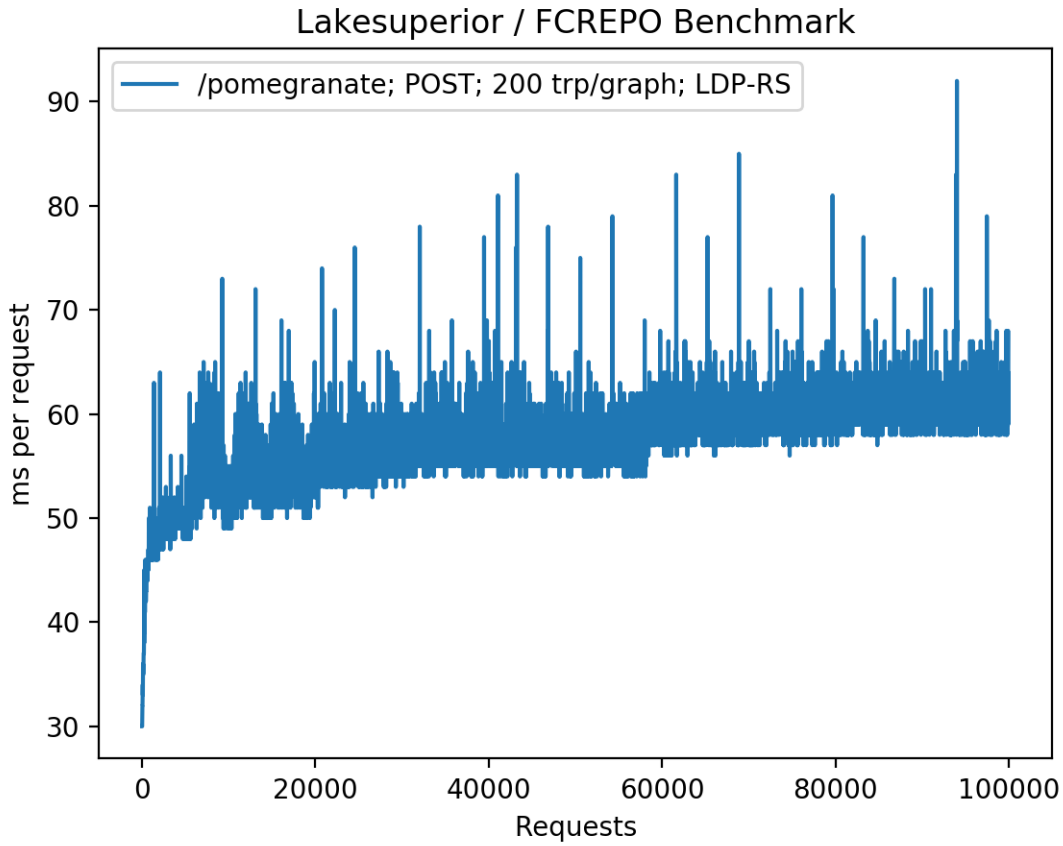

Fig. 7: Lakesuperior using Python API. The pattern is much smoother, with less frequent and less pronounced spikes. The O(log n) performance is more clearly visile here: time increases quickly at the beginning, then slows down as the repository size increases.

1.16 Lakesuperior Content Model Rationale

1.16.1 Internal and Public URIs; Identifiers

Resource URIs are stored internally in Lakesuperior as domain-agnostic URIs with the scheme `info:fcres<resource UID>`. This allows resources to be portable across systems. E.g. a resource with an internal URI of `info:fcres/a/b/c`, when accessed via the `http://localhost:8000/ldp` endpoint, will be found at `http://localhost:8000/ldp/a/b/c`.

The resource UID making up the looks like a UNIX filesystem path, i.e. it always starts with a forward slash and can be made up of multiple segments separated by slashes. E.g. `/` is the root node UID, `/a` is a resource UID just below root. their internal URIs are `info:fcres/` and `info:fcres/a` respectively.

In the Python API, the UID and internal URI of an LDP resource can be accessed via the `uid` and `uri` properties respectively:

```
>>> import lakesuperior.env_setup
>>> from lakesuperior.api import resource
>>> rsrc = resource.get('/a/b/c')
>>> rsrc.uid
/a/b/c
>>> rsrc.uri
rdflib.terms.URIRef('info:fcres/a/b/c')
```

1.16.2 Store Layout

One of the key concepts in Lakesuperior is the store layout. This is a module built with a specific purpose in mind, i.e. allowing fine-grained recording of provenance metadata while providing reasonable performance.

Store layout modules could be replaceable (work needs to be done to develop an interface to allow that). The default (and only at the moment) layout shipped with Lakesuperior is the *`resource-centric layout`*. This layout implements a so-called graph-per-aspect pattern which stores different sets of statements about a resource in separate named graphs.

The named graphs used for each resource are:

- An admin graph (`info:fcsystem/graph/admin<resource UID>`) which stores administrative metadata, mostly server-managed triples such as LDP types, system create/update timestamps and agents, etc.
- A structure graph (`info:fcsystem/graph/structure<resource UID>`) reserved for containment triples. The reason for this separation is purely convenience, since it makes it easy to retrieve all the properties of a large container without its child references.
- One (and, possibly, in the future, more user-defined) named graph for user-provided data (`info:fcsystem/graph/userdata/_main<resource UID>`).

Each of these graphs can be annotated with provenance metadata. The layout decides which triples go in which graph based on the predicate or RDF type contained in the triple. Adding logic to support arbitrary named graphs based e.g. on user agent, or to add more provenance information, should be relatively simple.

1.17 Storage Implementation

Lakesuperior stores non-RDF ("binary") data in the filesystem and RDF data in an embedded key-value store, LMDB.

1.17.1 RDF Storage design

LMDB is a very fast, very lightweight C library. It is inspired by BerkeleyDB but introduces significant improvements in terms of efficiency and stability.

The Lakesuperior RDF store consists of two files: the main data store and the indices (plus two lock files that are generated at runtime). A good amount of effort has been put to develop an indexing strategy that is balanced between write performance, read performance, and data size, with no compromise made on consistency.

The main data store is the one containing the preservation-worthy data. While the indices are necessary for Lakesuperior to function, they can be entirely rebuilt from the main data store in case of file corruption (recovery tools are on the TODO list).

Detailed notes about the various strategies researched can be found here.

1.17.2 Scalability

Since Lakesuperior is focused on design simplicity, efficiency and reliability, its RDF store is embedded and not horizontally scalable. However, Lakesuperior is quite frugal with disk space. About 55 million triples can be stored in 8Gb of space (mileage can vary depending on how heterogeneous the triples are). This makes it easier to use expensive SSD drives for the RDF store, in order to improve performance. A single LMDB environment can reportedly scale up to 128 terabytes.

1.17.3 Maintenance

LMDB has a very simple configuration, and all options are hardcoded in LAKESuperior in order to exploit its features. A database automatically recovers from a crash.

The Lakesuperior RDF store abstraction maintains a registry of unique terms. These terms are not deleted if a triple is deleted, even if no triple is using them, because it would be too expesive to look up for orphaned terms during a delete request. While these terms are relatively lightweight, it would be good to run a periodical clean-up job. Tools will be developed in the near future to facilitate this maintenance task.

1.17.4 Consistency

Lakesuperior wraps each LDP operation in a transaction. The indices are updated synchronously within the same transaction in order to guarantee consistency. If a system loses power or crashes, only the last transaction is lost, and the last successful write will include primary and index data.

1.17.5 Concurrency

LMDB employs MVCC to achieve fully ACID transactions. This implies that during a write, the whole database is locked. Multiple writes can be initiated concurrently, but the performance gain of doing so may be little because only one write operation can be performed at a time. Reasonable efforts have been put to make write transactions as short as possible (and more can be done). Also, this excludes a priori the option to implement long-running atomic operations, unless one is willing to block writes on the application for an indefinite length of time. On the other hand, write operations never block and are never blocked, so an application with a high read-to-write ratio may still benefit from multi-threaded requests.

1.17.6 Performance

The Performance Benchmark Report contains benchmark results.

Write performance is lower than Modeshape/Fedora4; this may be mostly due to the fact that indices are written synchronously in a blocking transaction; also, the LMDB B+Tree structure is optimized for read performance rather than write performance. Some optimizations on the application layer could be made.

Reads are faster than Modeshape/Fedora.

All tests so far have been performed in a single thread.

1.18 RDF Store & Index Design

This is a log of subsequent strategies employed to store triples in LMDB.

Strategy #4a is the one currently used. The rest is kept for historic reasons and academic curiosity (and also because this took too much work to just wipe out of memory).

1.18.1 Storage approach

- Pickle quad and create MD5 or SHA1 hash.
- Store triples in one database paired with key; store indices separately.

Different strategies involve layout and number of databases.

1.18.2 Strategy #1

- kq: key: serialized triple (1:1)
- sk: Serialized subject: key (1:m)
- pk: Serialized predicate: key (1:m)
- ok: Serialized object: key (1:m)
- (optional) lok: Serialized literal object: key (1:m)
- (optional) tok: Serialized RDF type: key (1:m)
- ck: Serialized context: key (1:m)

Retrieval approach

To find all matches for a quad:

- If all terms in the quad are bound, generate the key from the pickled quad and look up the triple in `kt`
- If all terms are unbound, return an iterator of all values in `kt`.
- If some values are bound and some unbound (most common query):
  - Get a base list of keys associated wirh the first bound term
  - For each subsequent bound term, check if each key associated with the term matches a key in the base list
  - Continue through all the bound terms. If a match is not found at any point, continue to the next term

  - If a match is found in all the bound term databases, look up the pickled quad matching the key in `kq` and yield it

More optimization can be introduced later, e.g. separating literal and RDF type objects in separate databases. Literals can have very long values and a database with a longer key setting may be useful. RDF terms can be indexed separately because they are the most common bound term.

Example lookup

Keys and Triples (should actually be quads but this is a simplified version):

- A: - s1 - p1 - o1
- B: - s1 - p2 - o2
- C: - s2 - p3 - o1
- D: - s2 - p3 - o3

Indices:

- SK:
  - s1: A, B
  - s2: C, D
- PK:
  - p1: A
  - p2: B
  - p3: C, D
- OK:
- o1: A, C
- o2: B
- o3: D

Queries:

- s1 ?p ?o → {A, B}
- s1 p2 ?o → {A, B} & {B} = {B}
- ?s ?p o3 → {D}
- s1 p2 o5 → { } (Exit at OK: no term matches 'o5')
- s2 p3 o2 → {C, D} & {C, D} & {B} = { }

1.18.3 Strategy #2

Separate data and indices in two environments.

Main data store

Key to quad; main keyspace; all unique.

Indices

None of these databases is of critical preservation concern. They can be rebuilt from the main data store.

All dupsort and dupfixed.

@TODO The first three may not be needed if computing term hash is fast enough.

- t2k (term to term key)
- lt2k (literal to term key: longer keys)
- k2t (term key to term)
- s2k (subject key to quad key)
- p2k (pred key to quad key)
- o2k (object key to quad key)
- c2k (context key to quad key)
- sc2qk (subject + context keys to quad key)
- po2qk (predicate + object keys to quad key)
- sp2qk (subject + predicate keys to quad key)
- oc2qk (object + context keys to quad key)
- so2qk (subject + object keys to quad key)
- pc2qk (predicate + context keys to quad key)

1.18.4 Strategy #3

Contexts are much fewer (even in graph per aspect, 5-10 triples per graph)

Main data store

Preservation-worthy data

- tk:t (triple key: triple; dupsort, dupfixed)
- tk:c (context key: triple; unique)

Indices

Rebuildable from main data store

- s2k (subject key: triple key)
- p2k (pred key: triple key)
- o2k (object key: triple key)
- sp2k
- so2k
- po2k
- spo2k

Lookup

1. Look up triples by s, p, o, sp, so, po and get keys
2. If a context is specified, for each key try to seek to (context, key) in ct to verify it exists
3. Intersect sets
4. Match triple keys with data using kt

Shortcuts

- Get all contexts: return list of keys from ct
- Get all triples for a context: get all values for a contex from ct and match triple data with kt
- Get one triple match for all contexts: look up in triple indices and match triple data with kt

1.18.5 Strategy #4

Terms are entered individually in main data store. Also, shorter keys are used rather than hashes. These two aspects save a great deal of space and I/O, but require an additional index to put the terms together in a triple.

Main Data Store

- t:st (term key: serialized term; 1:1)
- spo:c (joined S, P, O keys: context key; 1:m)
- c: (context keys only, values are the empty bytestring)

Storage total: variable

Indices

- th:t (term hash: term key; 1:1)
- c:spo (context key: joined triple keys; 1:m)
- s:po (S key: P + O key; 1:m)
- p:so (P key: S + O keys; 1:m)
- o:sp (object key: triple key; 1:m)
- sp:o (S + P keys: O key; 1:m)
- so:p (S + O keys: P key; 1:m)
- po:s (P + O keys: S key; 1:m)

Storage total: 143 bytes per triple

Disadvantages

- Lots of indices
- Terms can get orphaned:
  - No easy way to know if a term is used anywhere in a quad
  - Needs some routine cleanup
  - On the other hand, terms are relatively light-weight and can be reused
  - Almost surely not reusable are UUIDs, message digests, timestamps etc.

1.18.6 Strategy #5

Reduce number of indices and rely on parsing and splitting keys to find triples with two bound parameters.

This is especially important for keeping indexing synchronous to achieve fully ACID writes.

Main data store

Same as Strategy #4:

- t:st (term key: serialized term; 1:1)
- spo:c (joined S, P, O keys: context key; dupsort, dupfixed)
- c: (context keys only, values are the empty bytestring; 1:1)

Storage total: variable (same as #4)

Indices

- th:t (term hash: term key; 1:1)
- s:po (S key: joined P, O keys; dupsort, dupfixed)
- p:so (P key: joined S, O keys; dupsort, dupfixed)
- o:sp (O key: joined S, P keys; dupsort, dupfixed)
- c:spo (context → triple association; dupsort, dupfixed)

Storage total: 95 bytes per triple

Lookup strategy

- ? ? ? c: [c:spo] all SPO for C → split key → [t:st] term from term key
- s p o c: [c:spo] exact SPO & C match → split key → [t:st] term from term key
- s ? ?: [s:po] All PO for S → split key → [t:st] term from term key
- s p ?: [s:po] All PO for S → filter result by P in split key → [t:st] term from term key

Advantages

- Less indices: smaller index size and less I/O

Disadvantages

- Slower retrieval for queries with 2 bound terms

Further optimization

In order to minimize traversing and splittig results, the first retrieval should be made on the term with less average keys. Search order can be balanced by establishing a lookup order for indices.

This can be achieved by calling stats on the index databases and looking up the database with *most* keys. Since there is an equal number of entries in each of the (s:po, p:so, o:sp) indices, the one with most keys will have the least average number of values per key. If that lookup is done first, the initial data set to traverse and filter will be smaller.

1.18.7 Strategy #5a

This is a slightly different implementation of #5 that somewhat simplifies and perhaps speeds up things a bit. The indexing and lookup strtegy is the same; but instead of using a separator byte for splitting compound keys, the logic relies on the fact that keys have a fixed length and are sliced instead. This *should* result in faster key manipulation, also because in most cases `memoryview` buffers can be used directly instead of being copied from memory.

Index storage is 90 bytes per triple.

1.18.8 Strategy #4a

This is a variation of Strategy 4 using fixed-size keys. It is the currently employed solution starting with alpha18.

After using #5a up to alpha17, it was apparent that 2-bound queries were quite penalized in queries which return few results. All the keys for a 1-bound lookup had to be retrieved and iterated over to verify that they contained the second ("filter") term. This approach, instead, only looks up the relevant keys and composes the results. It is slower on writes and nearly doubles the size of the indices, but it makes reads faster and more memory-efficient.

Alpha20 uses the same strategy but keys are treated as `size_t` integers rather than `char*` strings, thus making the code much cleaner.

1.19 Lakesuperior on a Raspberry Pi

Experiment in Progress

Lakesuperior has been successfully installed and ran on a Raspberry Pi 3 board. The software was compiled on Alpine Linux using musl C libraries. (it also run fine with musl on more conventional hardware, but performance benchmarks vis-a-vis libc have not been performed yet.)

Performance is obviously much lower than even a consumer-grade laptop, however the low cost of single-board computers may open up Lakesuperior to new applications that may require to connect many small LDP micro-repositories.

If this experiment proves worthwhile, a disk image contianing the full system can be made available. The image would be flashed to a microSD card and inserted into a Raspberry Pi for a ready-to-use system.

Some tweaks to the software could be made to better adapt it to small repositories. For example, adding a cpmpile-time option to force the use of fixed 32-bit keys on an ARM64 processor rather than the current 64-bit keys (a 32-bit system would use 32-bit keys natively), it would be possible for Lakesuperior to handle half-sized indices and still being capable of holding, in theory, millions of triples.

Cell phones next?

Fig. 8: Look, a Fedora implementation!

More to come on the topic.

- genindex
- modindex
- search

Python Module Index

l

Index

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V