University of Science
and Technology of Hanoi

Bachelor Thesis

Development of Data Lake Core

Author
Nguyễn Gia Phong

Supervisor
Trần Giang Sơn, PhD

July 18, 2021

Development of Data Lake Core

doi:10.5281/zenodo.tbd

Contents

Declaration

I declare that I have composed this thesis in its entirety, as the result of my own work, unless explicitly indicated otherwise via referencing. The presented work has not been submitted for any previous application for a degree or professional qualification.

1 Introduction

1.1 Motivation

Many researchers at University of Science and Technology of Hanoi (USTH) operate with data on a regular basis and often a dataset is studied by multiple researchers from different departments and points in time. Currently the data are organized manually, even on the laboratories' storages, which is prone to duplication and makes data discovery difficult.

A data lake shared among the university's researchers, professors and students will not only save resources but also improve productivity and promote interdisciplinary collaborations. With USTH's goal of growing to be an excellent research university in Việt Nam and in the region [1], building such data lake can be an essential task.

1.2 Background

A *data lake* is a massive repository of multiple types of data in their raw format at scale for a low cost [2]. The data's schema (structure) is defined on read to minimize data modeling and integration costs [2].

For the ease and efficiency of scaling, a microservice architecture could be a good choice. By arranging the data lake as a collection of loosely-coupled services, it becomes possible to scale individual services individually [3]. In this architecture, the *core* microservice is defined as the innermost component, which communicates directly with the storages. The core shall provide an application programming interface (API) for other components to upload, query and extract data.

Append-only storages only allow new data to be appended, whilst ensure the immutability of existing data. As immutable data are thread-safe, they

reduce the complexity of the concurrency model, making it easier to comprehend and reason about [4]. This is particularly useful in large distributed systems with multiple moving parts.

Since the data are immutable, each *content* can be given an identifier (ID), i.e. a content identifier (CID). For end-users, we also introduce a higher level concept: *dataset*, composing of not only the content but also relevant metadata for indexing. Like contents, datasets can also immutable, with changes written as a new revision linking to the previous one.

Append-only storages' operations boil down to two kinds: appending and reading. For the latter, sometimes the data are not wanted in their entirety, but filtered and accumulated. While data of different types usually requires different tools and libraries to query upon, the core API should be providing one single query language for all data types, plus their metadata. In this thesis, such usage is referred to as *query polymorphism*.

1.3 Objectives

The work presented here was done as part of a three-month internship in collaboration with several other students at USTH ICTLab[1] to build a data lake for a better management of the university's data. The internship focused on the lake's core microservice, which abstracts underlying persistent layers and perform relevant metadata transformation and discovery. It should provide an internal interface to other components for data ingestion, (primitive) query and extraction, as well as carrying out tasks for enhancing the discoverability and usability of the aforementioned datasets.

After the internship period, the resulting codebase shall be maintained by ICTLab and future students, so the work must be designed, implemented and documented in a way that ensures such possibility.

1.4 Expected Outcomes

The intended deliverables of the three-month internship are listed as follows:

- Requirement analysis of the data lake core
- Data lake core's architecture and design
- Core API design and specification
- Implementation and integration with other components

[1] `https://ictlab.usth.edu.vn`

2 Methodology

Development followed the evolutionary prototyping model: a robust prototype is built by improving and adding newly understood features [5]. This often took 7–10 days, with requirements, design, implementation and test suite all being refined.

2.1 Requirement Analysis

In this section, from given context and objectives, we analyzed the expected system for a set of features and derived a list of use cases. Supplementary specifications were also added to elaborate on the nonfuntional requirements.

2.1.1 Use-Case Model

As previously introduced, the most basic functions of the data lake core are content uploading and downloading, along with datasets addition and querying. A more advanced (and rather powerful) use case is content extraction, which allows one to fetch only the interested part of the content, e.g. extracting rows matching a certain predicate from (semi-)structured data. Together with logging, the core's use cases are summarized in figure 2.1.

Upload content This use case allows other microservices of the data lake to upload a content. Its flow of events is depicted as follows, where error handling is omitted for brevity, since errors replace the normal response.

1. A content is sent to the core microservice.
2. Core adds the content to the underlying storage and register it to the database (DB).

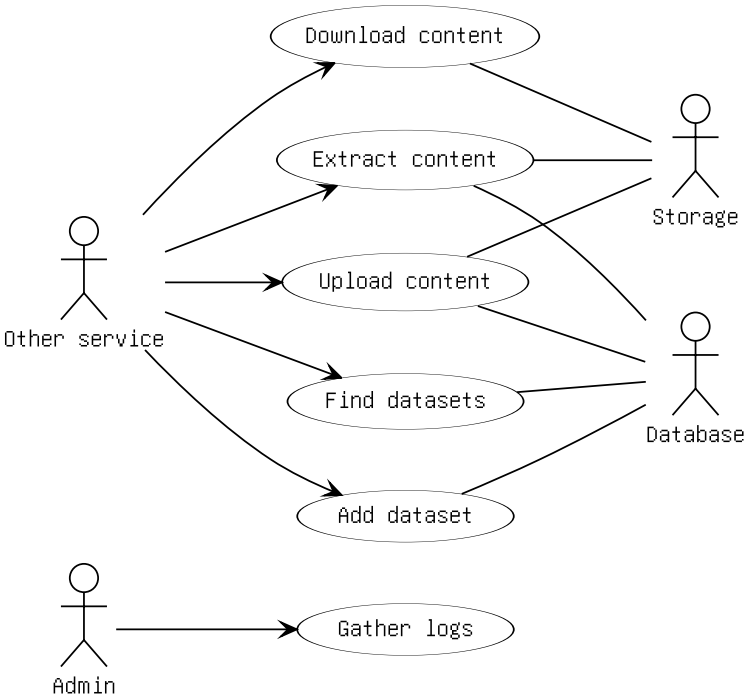

Figure 2.1: Use-case diagram of the core microservice

3. Core responds with the CID of the added content.

Add dataset This use case let other services add a dataset:

1. A dataset is sent to the core microservice.
2. Core adds the dataset to the underlying DB.
3. Core responds with the ID of the added dataset.

Find datasets This use case allows other services to find the datasets whose metadata satisfy a given predicate:

1. A predicate is sent to the core microservice.
2. Core runs a query in the underlying DB to find matching datasets.
3. Core responds with a linear collection of metadata, each of which satisfying the given predicate.

Download content In use case, other services fetch a content from the data lake core.

1. A CID is sent to the core microservice.
2. Core passes the CID to the underlying storage.
3. Core responds with the respective content.

Extract content This use case allows other services to extract a content's parts satisfying a given predicate:

1. A CID and a predicate is sent to the core microservice.
2. Core iterates the content for matching elements.
3. Core responds with the extracted elements.

Gather logs This use case let system admins study events occurring in the core microservice for debugging purposes:

1. A system admin requests logs from core.
2. The admin receives the list of past events.

2.1.2 Supplementary Specification

Besides the functionalities specified in the previous section, the following non-functional requirements were pinned down.

Performance Each instance of the data lake core should be able to respond up to 1000 simultaneous requests, which is approximated from the number of USTH researchers and students, every second. Furthermore, an instance should be able to maintain a high throughput for large datasets', preferably matching common local bandwidth (100 Mb/s to 1 Gb/s).

Supportability The data lake core and its dependencies must be able to run on common operating system (OS), including, but not limited to, GNU/Linux, Windows and macOS. While the microservice is likely to be deployed on the former, it is uncertain if it will be the future maintainers' OS of choice for development. Furthermore, languages used for implementation and depended systems should be either familiar or easy to learn.

Licensing The resulting software must be released under a copyleft license, in order to persist digital freedom in scientific research and promote independence and cooperation in education [6].

2.2 Design

The design to be presented takes inspiration from several prior arts, such as Qri and Kylo, especially for high-level ideas. However, these technologies were deemed as unsuitable for USTH due to their goal and scope differences. Qri is a peer-to-peer application which may have unpredictable performance

for larger but unpopular datasets, and only supports tabular data [7]. Kylo also does not support binary data, while requires substantial resources for each instance [8].

2.2.1 Architecture

Considering ICTLab's dynamic budget, the microservice architecture was chosen by Dr Trần Giang Sơn for the ease of scaling individual services out only when in need. Through his consultancy and discussions with other interns[1], it was decided requests from external clients and most internal services would go through a public API for authentication and authorization before being transformed to comply with and passed to the core API.

With the core service optimized for high input/output (I/O) performance (high throughput and low latency), operations of order of growth higher than linear complexity would be off-loaded to query engines for better horizontal scaling of compute-intensive tasks.

On the other side, the core service encapsulates distributed file system (DFS) and database management system (DBMS) and provides a consistent interface for those storages. Therefore, the data lake core must also include clients to talk to these outside systems. For the content extraction use case, we added an *extractor* component reading from the file system (FS) and DB. The result will be either responded directly through the core API or cached in the DB. The flow directions of data between previously discussed components are illustrated in figure 2.2.

2.2.2 Technology Choices

In a perfect world, choices of technology would be made following all other design decisions. However, existing technologies all have limitations (or at least trade-offs) that we need to be aware of to best decide on low-level details.

2.2.2.1 Logging

Events are logged to standard output/error to be picked up by the logging service provided by the OS for maximum portability. On most GNU/Linux distributions, journald does this for every systemd service and can be configured to upload to any remote endpoint [9] for debugging convenience.

[1] Lê Như Chu Hiệp, Nguyễn Phương Thảo, Nguyễn An Thiết, Trần Minh Hiếu and Nguyễn Quốc Thông

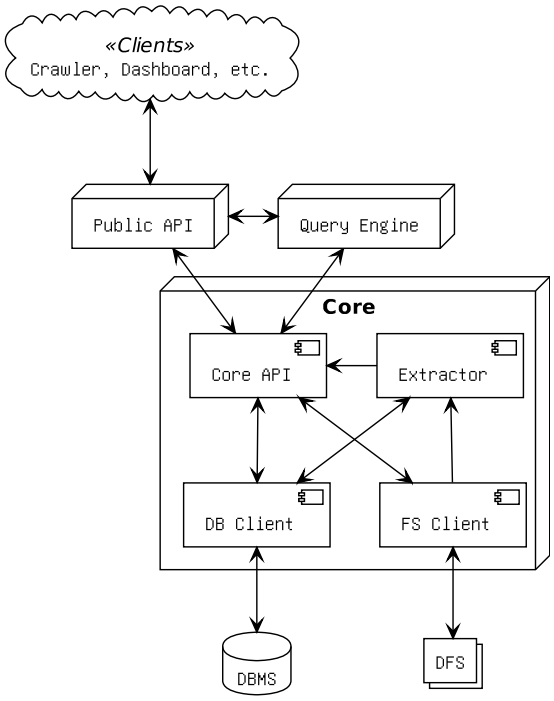

Figure 2.2: Data lake overall architecture with focus on core's components

2.2.2.2 Communication protocol

Google Remote Procedure Calls (gRPC) was first chosen for its high performance, but was later replaced by Hypertext Transfer Protocol (HTTP) due to the lack of multi-parameter streaming methods. This is essential for transporting data together with their metadata and was done by treating metadata as HTTP headers. As gRPC uses HTTP under the hood, the change in protocol would not add any overhead.

2.2.2.3 Programming languages

Java[2] was picked among languages included in any course from the university's information and communication technology (ICT) major, for Java virtual machine (JVM) implementations' performance (comparing to other general-purpose languages' runtime with garbage-collection [10]). The JVM also offers great interoperability with other languages. For interacting with dynamic data types, Clojure[3] was used to avoid performing Java reflection.

[2] `https://oracle.com/java`
[3] `https://clojure.org`

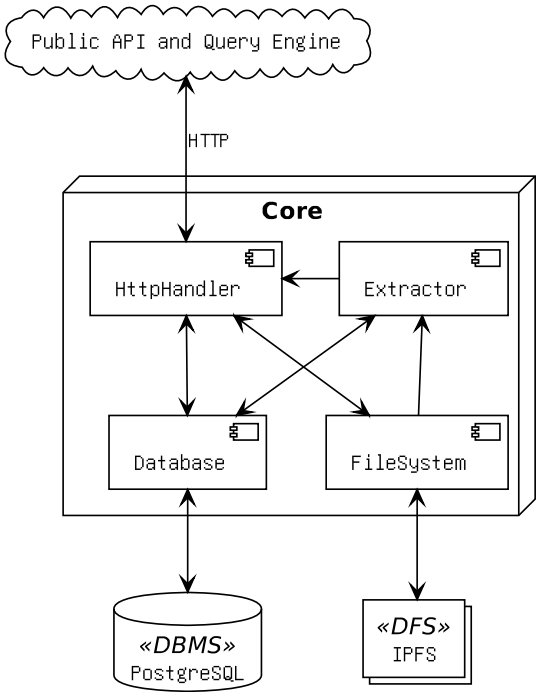

Figure 2.3: Data lake core high-level architecture

2.2.2.4 Distributed file system

Hadoop Distributed File System (HDFS) was initially considered because of Hadoop's popularity in state-of-the-art data lakes [2], however its lack of presence in every Unix-like OS's repository [11] (and the reasons behind it) is an deployment obstacle.

After analyzing several alternatives, InterPlanetary File System (IPFS)[4] was chosen for its cluster's ability to organically grow or shrink nodes without any performance interruption thanks to conflict-free replicated data type (CRDT) consensus [12]. In addition, the use of Merkle directed acyclic graphs (DAG) makes it an append-only storage with bidirectional mapping between content and CID while maintaining data integrity and eliminating content duplication [13].

[4] `https://ipfs.io`

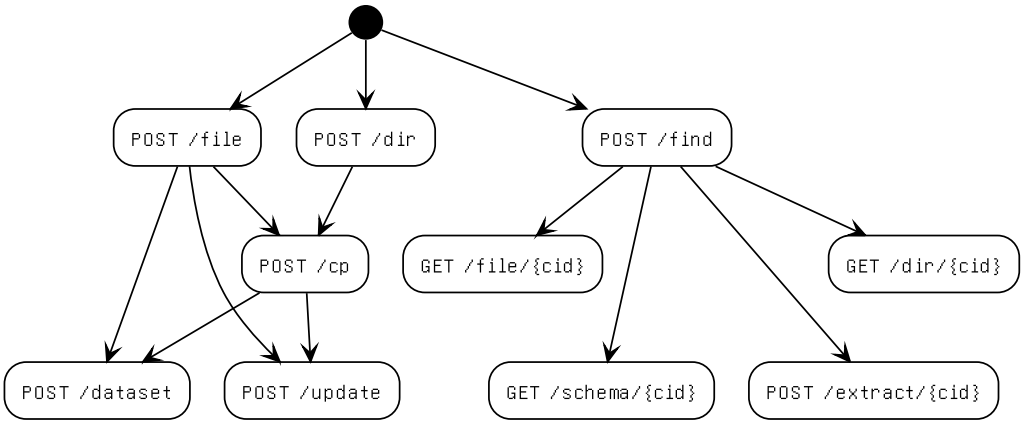

Figure 2.4: Core HTTP API endpoints in a common order of access

2.2.2.5 Database management system

Early prototypes used RethinkDB[5] for its embedded query language and dynamic data support. Due to the lack of a client with connection pool for JVM, though, we had to switched to PostgreSQL[6], which also natively support semi-structured data, to use its Java Database Connectivity (JDBC) driver and improve the performance.

It is worth noting that the connection to both DFS and DBMS must be abstracted for modularity. It is entirely possible that in the future the choices for those will no longer be the most suitable, and the transition to a fitter technology should be as frictionless as possible. The interfaces for the clients of these persistence systems were named following Java convention as shown in figure 2.3.

2.2.3 Interface

The API was derived from the use cases quite straightforwardly. Figure 2.4 sums up the available endpoints in a logical order. All appending operations are arranged on the left and the endpoints on the right are for retrieving the added data.

- `POST /dir`: Create an empty directory.
- `POST /file`: Add the file to the underlying file system.
- `POST /cp`: Copy file or directory inside a directory.
- `POST /dataset`: Add the dataset to the lake.

[5] `https://rethinkdb.com`
[6] `https://postgresql.org`

- `POST /update`: Add the updated dataset to the lake.
- `POST /find`: Find the data according to the given predicate.
- `GET /dir/{cid}`: List content of a file system directory.
- `GET /file/{cid}`: Stream content from underlying file system.
- `GET /schema/{cid}`: Fetch the JSON schema of a (semi-)structured content (a file in JavaScript Object Notation (JSON) or comma-separated values (CSV) format).
- `POST /extract/{cid}`: Extract rows from a (semi-)structured content.

One notable difference to the Use-Case Model is that operations directly on contents are separated for files and folders. Another is the endpoint for JSON schema: this shall be explained in subsection 2.3.2.

Except for file upload requests and file download responses, all communication is done in JSON.

2.2.4 Database Schema

The external DB is used for storing metadata. For contents, other than the CID, only the Multipurpose Internet Mail Extensions (MIME) type is required, while *extra* fields can be stored under a `jsonb` column—as a relational database, PostgreSQL cannot provide optional columns. Each content can then be represented by multiple datasets under different metadata, or it can be a child of a represented one.

On the other hand, a dataset must be backed by exactly one content, so its CID could be used as a foreign key. A few generic properties were believed to be common and useful for indexing, and thus were required: *description*, *source* and *topics*. Whilst the first two might be helpful in a full-text search, *topics* is an array of keywords which can be used to arrange datasets in a hierarchical view.

To encourage derivation of datasets and support reproducibility, a new entry is created upon `POST /update` instead of in-place modification. It is linked to the *parent* through the ID and create a tree of datasets that can be reconstructed from a query output.

The described relations were visualized in figure 2.5.

2.2.5 Query Language

Both `POST /find` and `POST /extract/{cid}` require a predicate for searching. Since the core API acts as a bridge between storages and higher-level services, it would be best to use a language as simple as possible to express the condition. In compilers, it is common to use an abstract syntax tree (AST) as an intermediate language.

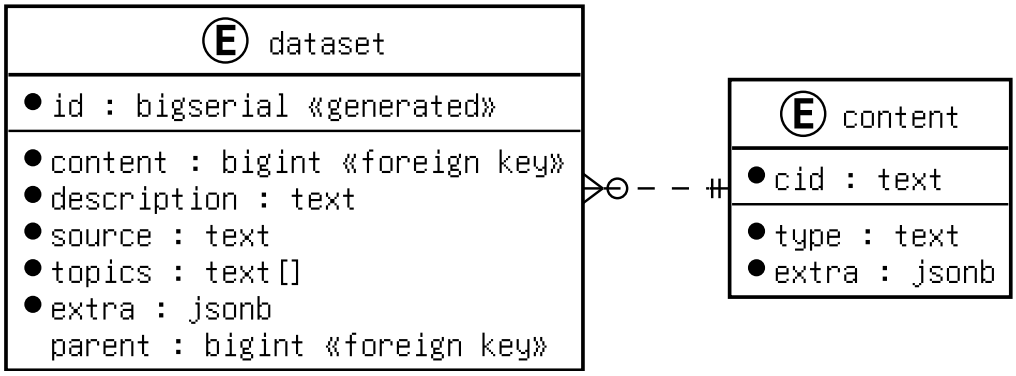

Figure 2.5: PostgreSQL schema for the data lake core

More precisely, an AST is not a language, but any tree can be unambiguously represented by a nested list with a prefix notation. Its most well-known form is the S-expression, which has been popularized by the Lisp family of programming languages [16]. However, it is rarely supported by mainstream technologies, while JSON has been growing in popularity. Therefore, the query abstract syntax tree (QAST) would be expressed in JSON nested arrays instead of parenthesized lists. To comply with JSON syntax, operators must be quoted as a string, for example the addition operator is `"+"`. The following are supported:

- Arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/) and modulo (%)
- Logical operators: *and* (&), *or* (|) and *not* (!)
- Comparison: ==, !=, >, >=, <, <=, ~(regular expression matching) and && (intersection)
- Field access: current row ($) and field getter (.)

An expression may support more than two operands. This helps flatten the query and applies to most logical and arithmetic operators (except for modulo) and the field getter, for example a predicate of *the field* `bar` *of the* `foo` *field of the current row inclusively matching 'branch' and 'batch'* could be written as `["~", [".", ["$"], "foo", "bar"], "b.*ch"]`.

2.3 Implementation

The implementation is hosted on GitHub under the ComLake namespace[7]. For the most part, it was derived directly from the design and use-case se-

[7] `https://github.com/ComLake/comlake.core`

quences. This section is dedicated for low-level designs and implementation details of less obvious cases.

2.3.1 Error Handling

In Java, errors are propagated through exceptions. While they are beneficial in certain ways, lexical scoping in `try/catch` blocks makes it impossible to put just a variable declaration with initialization in a `try` block. This means errors from subsequent statements using the variable are implicitly caught in the same handling, which can be unintentional[8].

Moreover, information from exceptions are usually meant for debugging, not to used by the program. Instead of defining wrapping exceptions to re-raise everywhere, we created an `Outcome` generic class whose method's signatures are shown in listing 2.1.

An `Outcome<Result, Error>` should be used as a return type, where `Result` is the type of the original return value, and `Error` can be anything the caller can analyze, including compound data like sets and hash tables.

Listing 2.1: Generic outcome type for replacing exceptions

```
public class Outcome<Result, Error> {
    public boolean ok;
    public Result result;
    public Error error;
    public static <Result, Error>
      Outcome<Result, Error> pass(Result value);
    public static <Result, Error>
      Outcome<Result, Error> fail(Error value);
}
```

2.3.2 Input/Output and Concurrency

As the data lake often work with large datasets, it would not be possible to load an entire request body into memory[9]. Instead, an asynchronous I/O model was adopted. In the outermost layer, Aleph[10] and Netty[11] are used to turn HTTP requests and responds into and from `java.io.InputStream`.

[8] One may alternatively declare the variable before the assignment, but uninitialized variables are subject to a different class of error.

[9] IPFS client for Java needed a patch to enable request streaming but upstream was irresponsive so a fork has been maintained ever since.

[10] `https://aleph.io`

[11] `https://netty.io`

These streams are only read or immediately written when ready, allowing us to apply back-pressure from connected storages.

In addition, instead of blocking operations by filling I/O buffer, this concurrency model switches to the remaining tasks. Multiple tasks are executed simultaneously in one thread so fewer threads need to be created, resulting in lower overhead and better performance.

Such thread pools are also utilized for metadata discovery, such as schema inferring. When a content is uploaded, its metadata extractor job is added to the pool and run in parallel with the HTTP handler. While this does not matches the definition of a data lake (schema on read), it allows such metadata to present in the dataset search (`POST /find`) without the client explicitly requesting for each entry. That being said, in order to extract *data*, the client needs to be aware of the schema and should not be pooling on the search endpoint. Hence, `GET /schema/{cid}` exists as a synchronization point.

Moreover, as it is possible for the schema to be requested before the scheduled inferring job finishes and duplicate the task, such race condition is minimized by memoizing the future. In Clojure, this was done as shown in listing 2.2.

Listing 2.2: Constructing a metadata extractor

```
(defn metadata-extractor [fs db]
  (memoize (fn [cid mime] (future ...))))
```

2.3.3 Metadata Extraction

From uploaded content, certain metadata can be extracted. For multimedia, these are directly embeded in the files, such as Exif for images and audio recordings (in JPEG, TIFF, WAV or PNG formats [14]). These metadata can later be used for dataset indexing and searching.

For strutured or semi-structured data, a schema could be inferred from the content, which can later be used to extract specific rows or join with other data, as seen with `qri sql` [15]. We stored these schemata as JSON schema for interopability, and effectively treats structured formats like CSV as semi-structured.

CSV schema were then inferred by pattern matching each value. In the core service this was done by passing the data to the higher-order function `reduce`, along with a function `and`'ing the previously detected type with the result of (regular expression) pattern matching, falling back to `string`.

Conceptually it would be similar for JSON, but we used a library[12].

One other semi-structured file format is Extensible Markup Language (XML). However, as its elements' attributes and contents cannot be directly mapped to JSON key-value objects, it is not possible to generate a JSON schema for a XML file.

2.3.4 Query Transformations

As the QAST is used for both `POST /extract/{cid}` and `POST /find`, it is respectively translated into JVM and structured query language (SQL) for processing. First, the original JSON request body is converted to a Clojure `vector`. Its first element is then looked up in the operator table and the rest are recursively applied to the found function `op` if the number of arguments is valid.

When translate to SQL, in `op` is simply string concatenation with the operator strings. Since these expressions are in-fix, it is necessary to wrap parentheses around them. Strings and arrays also need special formatting to be recognized by PostgreSQL when the result is used as a `WHERE` clause.

To extract content, it is slightly more complex: the result has to be a Clojure function `pred` to be passed to the higher-order `filter` along with a collection of Clojure `map`. Let `row` be a `map`, `pred` would have the signature `(fn [row] ...)`. Because `pred` is created from `op`, the latter must be of the form `(fn [args] (fn [row] ...))`. As operators are applied recursively, `args` is a collections of `pred`, or each one in `args` must be called with row: `(map #(% row) args)`. Combining all these, a operator mapping for normal functions could be implemented as seen in listing 2.3[13].

Listing 2.3: Constructing a function returning another one lazily applying given the given one

```
(defn mkfn [func]
  (fn [args]
    (fn [row] (apply func (map #(% row) args)))))
```

For example, `(mkfn +)` is an `op` for addition. Macros, however, cannot be passed to `mkfn` so their more generic functional equivalent was used instead, like `every? identity` replacing `apply and`. After some preprocessing, supported (semi-)structured data (JSON and CSV) are filtered by the same `pred`.

[12]Clojure JSON Schema Validator & Generator: `https://github.com/luposlip/json-schema`

[13]The final implementation involved some additional null-checking.

With the same QAST being used for PostgreSQL, JSON and CSV, query polymorphism has been achieved, which simplifies and unifies the API.

2.3.5 Configuration Parsing

For the ease of deployment, settings for connection to PostgreSQL and IPFS are read from configuration files instead of being hard-coded. TOML was chosen as the format and configurations are fallen back user to system configuration file and finally hard-coded defaults. Their locations on different OS's are determined by the library AppDirs[14].

2.4 Quality Assurance

To assure the correctness of the implementations, multiple testing methods are used. The automated test suite were written as a mixture of both integration and unit tests, aiming for at least 90 % line coverage. They were run locally as well as on SourceHut continuous integration (CI) service[15] to test the deployment of depending services as well. An instance was also set up on ICTLab with the help and manual testing of Lê Như Chu Hiệp and Nguyễn Phương Thảo.

[14] `https://github.com/harawata/appdirs`
[15] `https://builds.sr.ht/~cnx/comlake.core`

3 Evaluation

3.1 Results

Quantity-wise, the internship has delivered all expected outcomes and functional requirements. In summary, requirement analysis, architecture, design and implementation of ComLake core were completed. The API was sufficiently documented[1] for other components to integrate with.

At the end of the internship period, the codebase had a modest size of under 1200 lines of code (excluding all tests and documentation), over a half which was in Java and the remaining was in Clojure. The test suite on the other hand was written entirely in Clojure. It contained 28 assertions in total and covered over 98 % *of the Clojure code*.

Since the first release, eight tags have been published in a course of two months, alternating between adding new features and delivering bug fixes. Despite intensive testing, we were still lacking the confidence to bump to a non-zero major version number and warrant a stable interface: the core API changed still slightly every other commit.

Speaking of performance, the number of connection a core instance could serve has been in the order of thousands since the transitioning from RethinkDB to PostgreSQL, as shown in table 3.1. Note that the drop in the base (control) operation in comlake.core 0.2.0 is likely due to the introduction of logging.

[1] `https://comlake.github.io/comlake.core/api.html`

Table 3.1: Performance in requests per second between comlake.core 0.1.1 and 0.2.0 using RethinkDB and PostgreSQL as DB back-end respectively, benchmarked using wrk on a Intel® Core™ i5-8250U CPU and 8 GB of memory

Operation	comlake.core 0.1.1	comlake.core 0.2.0
File upload *and* DB insert	325.40	357.28
DB look-up	121.01	5575.89
Download file	8217.39	6238.30
No-operation (control)	80994.41	29788.50

3.2 Discussion

3.2.1 Polyglot Development

One notable aspect of the development process is the interoperation of Java and Clojure. While it was seamless to call from one language to another, some tooling of one often encountered troubles working with the other, for instance hot loading plugins for Java and Clojure failed to work together, thus Clojure code had to be manually reloaded and the HTTP server has to be restarted frequently. One other drawback was the lack of a coverage measuring tool for both languages, one for Clojure[2] was used because it was used to implement most of the non-trivial data transformation, plus the HTTP routing for Aleph. This made the reported figure less meaningful; we had to manually check if Java methods are tested, especially on branching points.

Originally, Java was dictated as the sole language for implementation for future maintenance by other USTH students and staff. Clojure was added for first-class functions which were used extensively in constructing JVM-native predicates for data extraction. Closures were also really handy in this case such as in listing 2.3 and for providing FS and DB clients explicitly to a memoized function in listing 2.2 (in contrast to singletons or global variables). While homoiconicity was not utilized directly, Clojure's builtin types allowed organizing target functions in query transformations directly as key-value maps. Together, Clojure helped improve the codebase by making it more concise, intuitive and data-oriented, and thus more maintainable.

[2]Cloverage: `https://github.com/cloverage/cloverage`

3.2.2 Concurrency

The majority of concurrency handling is leveraged by the Aleph framework, which efficiently utilize thread-pool to maximize central processing unit (CPU) utilization and task switching for best I/O performance. Whilst running alone it can serve up to tens of thousands or requests every second on a personal laptop, it is not the only I/O-intensize spot. The way connections are made to to persistence layers is also important. As also shown in table 3.1, using a DB client with connection pool can greatly enhance the performance. The same could be said to the IPFS client, which currently does not implement any connection pool and seem to have a relatively high latency compared to the DB counterpart.

Other concurrency mechanisms were also utilized, such as memoized `future` as a synchronization point to minimize re-execution of metadata extraction. This was not a perfect solution though, while it take almost negligible time to create the `future` in listing 2.2, theoretically there still exists a race condition. More importantly, `memoize` memoizes all results, which occupy memory over time; a cache invalidation mechanism should be added.

3.2.3 Query Polymorphism

The use of a same query language for both dataset lookup and content extraction has both advantages and downsides. On the upside, an unified interface reduces the complexity and is arguably easier to adopt. Even in the development of the core itself, the QAST allowed test cases to share queries and regressions could be spotted from different results from a same query.

On the other hand, in a sense some flexibility was lost. Some data formats like XML cannot have their structure expressed in a JSON schema. Consequently, the data cannot be extracted without a workaround transforming them to a JSON-like format. Such workaround is, though not pretty, possible in theory.

3.2.4 Use of Free Software

Free/libre software is used exclusively as dependencies and in the development process. The benefits of this went beyond making the resulting software copyleft semantically. For instance, the schema inference was made after studying Qri's source code[3] and several patches were made to a fork of the IPFS client for Java and redistributed specifically for ComLake[4]. That

[3] Although the implementation is not directly based on the inspiration.

[4] `https://github.com/ComLake/java-ipfs-http-client`

being said, some library such as JSON-java had to be avoided in order to comply with the GNU Affero General Public License.

4 Conclusion

4.1 Future Works

Acknowledgment

This work would not have been possible without open access, Sci-Hub and Library Genesis. In addition, Wikipedia was a great index and summary for books and articles of certain topics.

Moreover, I would like to thank my colleagues and supervisors for their feedbacks, collaborations and encouragements. My family and friends (both online and in-real-life) were also incredibly supportive.

Bibliography

[1] *Mission and Vision*. University of Science and Technology of Hanoi. Retrieved 2021-07-07. `https://usth.edu.vn/en/abouts/Mission-et-Vision.html`.

[2] Huang Fang. "Managing Data Lakes in Big Data Era: What's a data lake and why has it became popular in data management ecosystem". *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems*, pp. 820–824. IEEE, 2015. doi:10.1109/CYBER.2015.7288049

[3] Chris Richardson. "1.4.1 Scale cube and microservices". *Microservice Patterns*. Manning Publications, 2018. ISBN 9781617294549.

[4] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer and David Holmes. "3.4. Immutability". *Java Concurrency in Practice*. Addison Wesley Professional, 2006. ISBN 9780321349606.

[5] Alan Mark Davis. "Operational Prototyping: A new Development Approach". *IEEE Software*, vol. 9, no. 5, pp. 70–78. IEEE, 1992. doi:10.1109/52.156899.

[6] Richard Stallman. "Why Schools Should Exclusively Use Free Software". *Free Software and Education*. GNU Project. Retrieved 2021-07-14. `https://www.gnu.org/education/edu-schools.html`.

[7] *What is Qri?* Qri.io. Retrieved 2021-07-15. `https://qri.io/docs/getting-started/what-is-qri`.

[8] "Review Dependencies". *Kylo 0.10.0 documentation*. Kylo. Retrieved 2021-07-15. https://kylo.rtfd.io/en/v0.10.0/installation/Dependencies.html

[9] "systemd-journal-upload.service(8)". *systemd manual pages*. Retrieved 2021-07-15. https://www.freedesktop.org/software/systemd/man/systemd-journal-upload.service.html.

[10] Isaac Gouy. *Which programming language is fastest?* The Computer Language Benchmarks Game. Retrieved 2021-07-14. https://benchmarksgame-team.pages.debian.net/benchmarksgame.

[11] *Search Results for HDFS*. Packages Search for Linux and Unix. Retrieved 2021-07-14. https://pkgs.org/search/?q=HDFS.

[12] "Consensus components". *Pinset orchestration for IPFS*. Protocol Labs. Retrieved 2021-07-15. https://cluster.ipfs.io/documentation/guides/consensus.

[13] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System*. CoRR, 2014. arXiv:1407.3561.

[14] Glenn Randers-Pehrson and Tom Lane. "3.7. eXIf Exchangeable Image File (Exif) Profile". *Extensions to the PNG 1.2 Specification, Version 1.5.0*. Retrieved 2021-07-18. https://ftp-osl.osuosl.org/pub/libpng/documents/pngext-1.5.0.html#C.eXIf.

[15] *Qri CLI command line reference*. Qri.io. Retrieved 2021-07-18. https://qri.io/docs/reference/cli_commands.

[16] John McCarthy. "The LISP Programming System". *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. Communications of the ACM, 1960.

C

Terms and Acronyms

Glossary

closure a function having access to variables in the scope it was created in. 18

content a file or a directory. 2–4, 10, 13, 27

homoiconicity a property of programs' primary representation being a data structure in their language's primitive type. 18

predicate an expression evaluating to either true or false. 3–5, 10

tree a connected, acyclic graph. 10, 11, 27, 28

Acronyms

API application programming interface. 1, 2, 6, 9, 10, 15, 17
AST abstract syntax tree. 10, 11

CI continuous integration. 15
CID content identifier. 2, 4, 8, 10
CPU central processing unit. 18, 19
CRDT conflict-free replicated data type. 8
CSV comma-separated values. 10, 13–15

DAG directed acyclic graphs. 8
DB database. 3, 4, 6, 10, 18, 19
DBMS database management system. 6, 9
DFS distributed file system. 6, 9

FS file system. 6, 18

GNU GNU's not Unix. 5, 28
gRPC Google Remote Procedure Calls. 7

HDFS Hadoop Distributed File System. 8
HTTP Hypertext Transfer Protocol. 7, 12, 13, 18

I/O input/output. 6, 12, 13, 19
ICT information and communication technology. 7
ID identifier. 2, 4, 10
IPFS InterPlanetary File System. 8, 12, 15, 19

JDBC Java Database Connectivity. 9
JSON JavaScript Object Notation. 10, 11, 14, 15, 19
JVM Java virtual machine. 7, 9, 14, 18

MIME Multipurpose Internet Mail Extensions. 10

OS operating system. 5, 6, 8, 15

QAST query abstract syntax tree. 11, 14, 15, 19

SQL structured query language. 14

USTH University of Science and Technology of Hanoi. 1, 2, 5, 18, 25

XML Extensible Markup Language. 14, 19