University of Science
and Technology of Hanoi

Bachelor Thesis

Development of Data Lake Core

Author
Nguyễn Gia Phong

Supervisor
Trần Giang Sơn, PhD

July 15, 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 all errors, if occur, replace the normal response.

1. A content is sent to the core microservice.

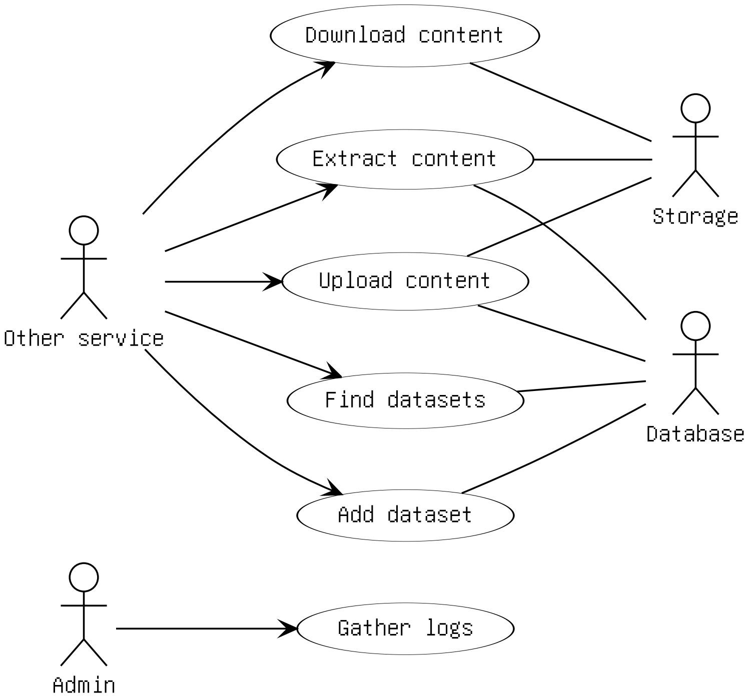

Figure 2.1: Use-case diagram of the core microservice

2. Core adds the content to the underlying storage and register it to the database (DB).
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 GNU/Linux, 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 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

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

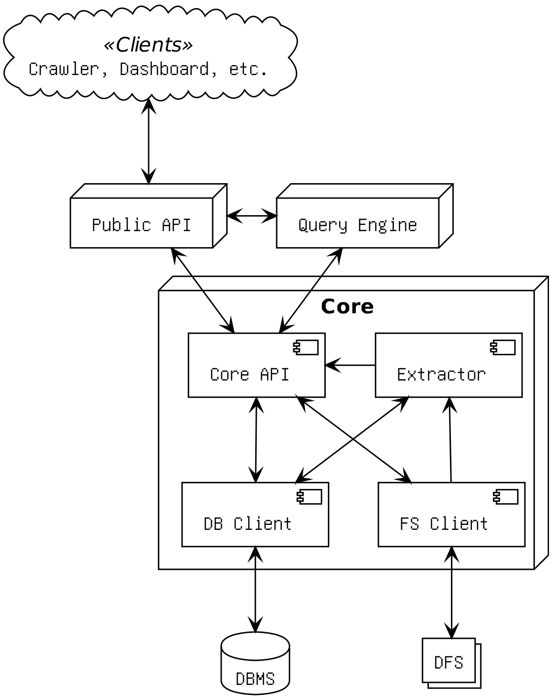

Figure 2.2: Data lake overall architecture with focus on core's components

distributions, journald does this for every systemd service and can be configured to upload to any remote endpoint [9] for debugging convenience.

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

[2] https://oracle.com/java

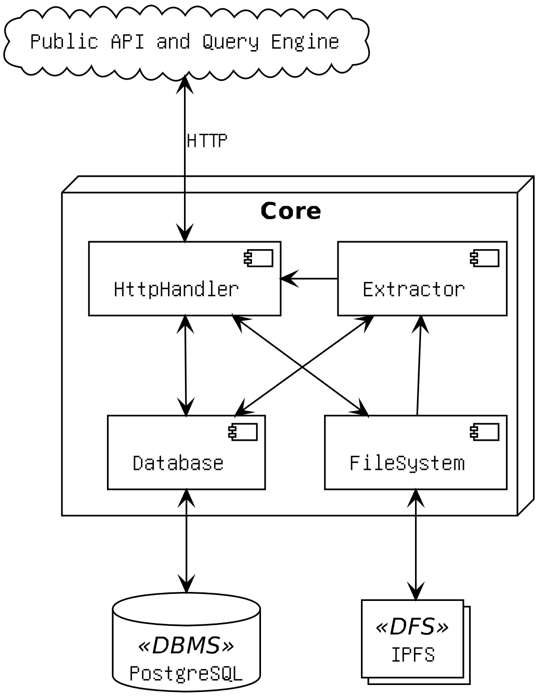

Figure 2.3: Data lake core high-level architecture

also offers great interoperability with other languages. For interacting with dynamic data types, Clojure[3] was used to avoid performing Java reflection.

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 makes it an append-only storage with bidirectional mapping between content and CID while maintaining data integrity [13].

[3] https://clojure.org

[4] https://ipfs.io

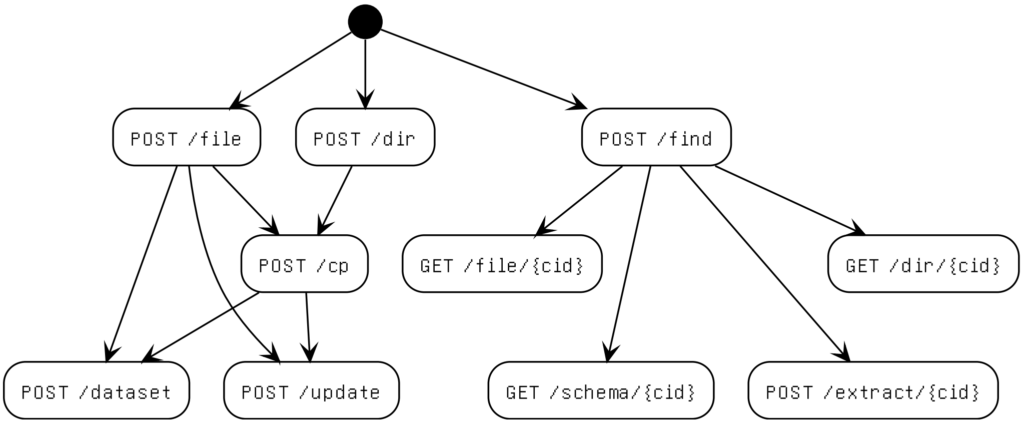

Figure 2.4: Core HTTP API endpoints in a common order of access

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, 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 persistency 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.
- `POST /update`: Add the updated dataset to the lake.

[5] `https://rethinkdb.com`

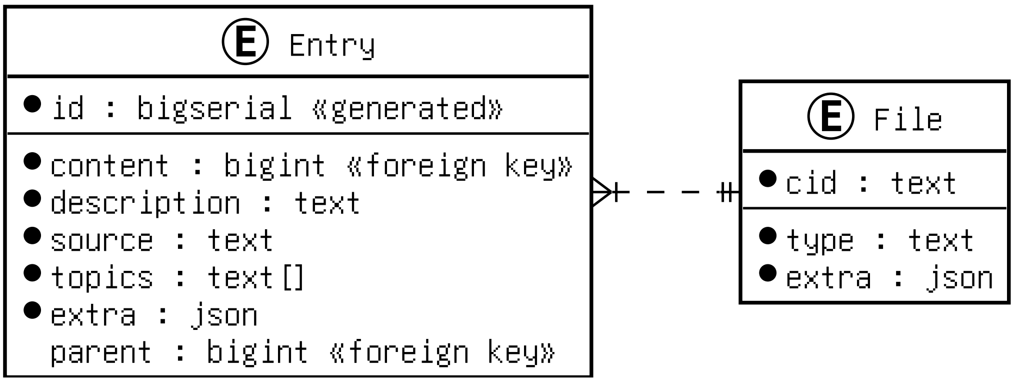

Figure 2.5: Database schema

- `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 2.3.4.

2.2.4 Database Schema

2.2.5 Query Abstract Syntax Tree

2.3 Implementation

2.3.1 Error Handling

2.3.2 Input/Output Handling

2.3.3 Query Transformations

2.3.4 Concurrency

2.3.5 Configuration Parsing

2.4 Quality Assurance

3 Evaluation

3.1 Results

3.2 Discussion

4
Conclusion

A 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.

B

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.

C

Terms and Acronyms

Glossary

content a file or a directory. 2–5, 21

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

Acronyms

API application programming interface. 1, 2, 6, 9

CID content identifier. 2, 4, 5, 8
CRDT conflict-free replicated data type. 8
CSV comma-separated values. 10

DB database. 4, 6
DBMS database management system. 6, 9
DFS distributed file system. 6, 9

FS file system. 6

GNU GNU's not Unix. 5, 21
gRPC Google Remote Procedure Calls. 7

HDFS Hadoop Distributed File System. 8
HTTP Hypertext Transfer Protocol. 7

I/O input/output. 6
ICT information and communication technology. 7
ID identifier. 2, 4

IPFS InterPlanetary File System. 8

JDBC Java Database Connectivity. 9
JSON JavaScript Object Notation. 10
JVM Java virtual machine. 7, 9

OS operating system. 5, 6, 8

USTH University of Science and Technology of Hanoi. 1, 2, 5, 6, 19