Report 5

Detail of each logical block

Le Nhu Chu Hiep

1. Introduction

This report is a part of the design task in the hivilake project. The complete system design mentioned in report (4), but it is still too abstract for the direct implementation. Therefore, this document is formed to provide a clearify vision of each logical block in system. In this way, the implement task will be more straight forward.

2. Objective

In this report, the logical block is the biggest target. The doc will try to split the block into smaller logical block, interface and more detail explanation about the role, purpose and suggestion of each block. Moreover, the suggestion, recommend and a good idea during design detail could be mention also. And because each block has their own properties and attribute, the reader should not expect to see the same sub-section in each block section.

3. Methodology

3.1 Storage Wrapper

For the convenient purpose, from now, the storage wrapper will be called: SW.

3.1.1 Brief Description

The idea of framework is a self-describe storage system with all logical component built on top of it. And for the logical component communicate to the storage, a gateway is defined called SW. It is expected to provide enough tool chain for storage access purpose.

3.1.2 Motivation

During research time, I recognize there are plenty of option for storage. Each option come with their own strength and attribute. And my point is that the system should be flexible enough to work with many type of storage. But it is too complicate for design, maintain and extent system for many storage.

Then the solution is group all storage accessing to a unique logical component. Then that componenet will provide interface to another block to use. This design grand all access handler to the one block only so the maintenance and extension become much easier.

3.1.3 Discussion

In this dicession, I will mention to 2 issues, first is the the features will be integrated into SW and the other is the SW design constrain.

3.1.3.1 Feature Integrate

The storage wrapper simplifies the storage access job and hides detail attribute of the storage system. But since SW reduce the complexity of the storage system, it come with the potential risk that can break the system. Those risk can be mention in 2 below question:

- How many is enough ?
- How not to eliminate the strengh of the storage system ?

Now, let me explain them carefully. Since SW try to become the gateway that connect higher layer to the low level storage, it need to provide enough storage access feature. But if it stick too deep with a specific storage, SW will lost the flexibility and hard to be compatible to another storage system. The simple solution will be integrated basic feature only. Keep everything simple will enable high compability for SW. Unfortunately, the simple SW also means eliminating the strengh of the storage system. Like the hadoop system comes with many advance feature as the parrallel computing tool as well as many extra system attribute tuning but not all of them should be included in SW since it stick too deep with the hadoop storage.

So the perfect solution is not exist, instead, it should be "Balance in all things". And in the below diagram, I will propose a balance design.

Features Design

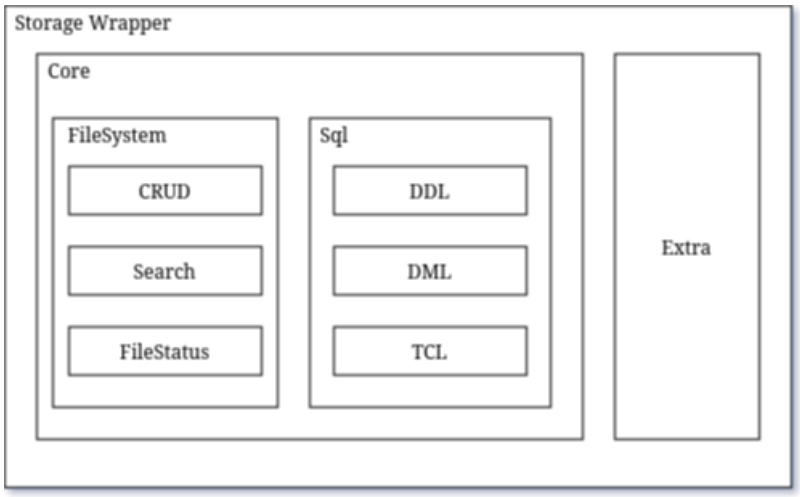

Figure 1: Storage Wrapper Feature Diagram

In this design, the system is splited into 2 package: core and extra. The core should include the basic function of a storage, so that the higher layer can reuse the core feature with many difference storage system as well as they share the same basic attribute. And for leveraging the userful feature of the specific storage system, SW provide the "extra" space. It basically is another core and serves as "user repository". It provide more deeper tuning feature for the specific storage feature. For any higher logic using object provided from extra, they should not expected to be compatible with widely storage system.

The SW core has 2 more smaller part supporting 2 different types storage are filesystem and sql. With the filesystem, CRUD operation should be supported. And the filestatus extraction and search are the importante operations for data discovery so the core also integrate them. On the other, the tabular data is a very unique data type with lots of useful application. So SW expected to have the list of special operation to work with it. And because the sql is the greatest language to handle this data type, SW is expected to provide some useful part of sql include: DDL (Data Definition Language), DML (Data Manipulation Language) and TCL (Transaction Controller Language).

Although NoSql is another popular data type and even more useful than tabular data. The concepture about NoSql is not unified and they vary from vendor to vendor. It cause difficult to SW to implement the concept as well as a unique function for all. Therefore, NoSql will not be handled in the core, instead, it should be mention in "extra"

part. Moreover, many NoSql engine accepts sql language also. Therefore, the NoSql supported is not necessary concerned for now.

In the time this report is writen, while the core has very detail design, the “extra” part is added for supporting future extensible only and does not included in the implementing plan.

3.1.3.2 Design Constraint

During implementation time, the consistency need to be carefully thought of. And for guarantee SW is unique and consistency in developing and maintaining time, there are 3 questions should be answered:

- How should another component use SW ?
- What functionality feature should SW implement ?
- How to extent SW for future demand ?

For first question, the interface mechanism of java is the best choice right now. It provide a list of method to interact with SW. And since one interface can be shared between multiple class, SW allow caller to access into many similar storage type without changing calling method.

The interface mechanism also can be leveraged to solve the second and third questions. Combining interface mechanism and inherit property with several constraints rule is the key solution for 2 final question. Below is list of constraint rule (it should be proven and update through time).

Constraints Rule

1. The class should implement a really needed interface only.
2. The class inherit is prefered than a new class.
3. A interface should only include a limited properties following its name.
4. SM should has its own pre-define set of exception and interface raise them only.
5. A good and detail document is necessary for both class, interface and exception.
6. The number of interfaces should not be obstacle, if the storage system has a new properties can not merge to any exist interfaces, then it deserves a new one.
7. The interface should be the only road to communicate with SW.

Constraint Diagram

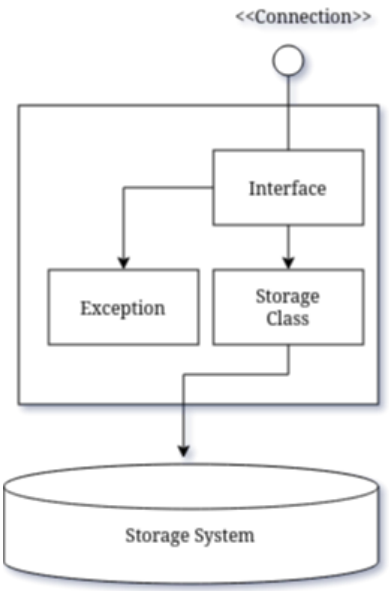

Figure 2: System Wrapper Constraint Diagram

The constraint diagram help simplify the design job a lo. From the low-level developer point of view, if they want to integrate new type of storage system into the lake and understand what they want, they can following a set of steps below:

1. Choose list of interface that the new storage system is expected to support.
2. If the expected can not be reached with current interfaces list, write new interface and comeback to step 1.
3. Implementing the storage class and handle the exception as well as raise expected exception in interface.
4. If the predefine exception can not captured all error in new storage system, provide new exception (do not forget provide/update also the interface to throw that exception). Then come back to step 3.
5. Write factory class to initialize the storage class.
6. Test code
7. Pushing code into lib file for realeas as well as providing a good doc.
8. Providing extra interface package for new utility developer.

Any end user can check the SW library is compatible to their system or not through the SW release doc and list of its supporting interface. If everything go well, they can integrate/replace new SW into their system with very little effort. For the utility developer team, they can build or update their utility for new SW with fully support about lib and doc.

3.2 Utility Features

The hivilake system come with 3 default utilities: Storage Manager, File Query, System Log. The reason is because the hivilake begin with the concept to provide a light file manager system built on top of the distributed system, and I feel that a basic file manager should support only 3 features which are "store file", "search file" and a "log mechanism" to keep track activity of whole system. So the below section will talk into detail of each utility.

3.2.1 Storage Manager

For the convenient writing, from now on, the concept "Storage Manager" will mention as SM.

SM is consider to be a important part in the utility layer. Its job enable user to ingest data into lake and make sure everything go right. Since this utility can modify the storage, any misunderstand or wrong manipulation can cause the crash for the data pool. Therefore, SM design task should be very careful not only in ingest logic (actually it is very simple) but in the **exception and corrupt handle logic**.

But before talking about the most headache part (exception and corrupt handling). Let following the flow and defining all SM concept first.

Hivi Repo

For very first time of this project, the data type target is medical image and voice viet file. They are binary file and rarely change during time. So that, while was developing the hivilake system, I had also thought about a storage design orienting metadata to store and manage binary file called "Hivi Repo". This is the metadata oriented design. Therefore, the first SM version will support "Hivi Repo" design only. Futher data type storing like sql or nosql as well as another binary storing method will be mention in another verison.

The hivi repo is defined as a self-describe repository/folder. It is used to store any kind of binary file having similar attribute plus some extra metadata. The future program can use this metadata to perform the audit or discover task. Specially, the all file metadata will be centralized into 1 file. And because the metadata has the same schema, the metadata file will be considerd to as a tabular data file and be access as a sql file. That mean the advance sql technology can be applied to increase extracting information process from metadata. Therefore, the struct of a hivi repo will like:

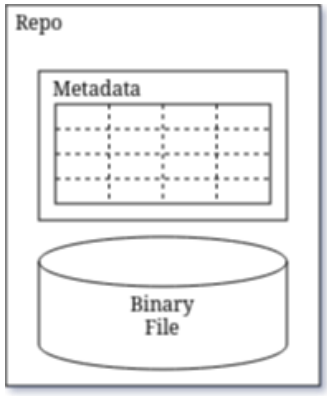

Figure 3: Hivi Repo Represent

Moreover, as git mechanism, the hivi repo store all metadata and any extra information in a sub-folder called ".hivilake". So any directory contain ".hivilake" will be considered as a repo and one hivi repo must contain the ".hivilake" sub-folder.

For futher discussion on this storage style will be in other report. The hivi repo mention here for better designing SM and showing enough information for that purpose only.

SM Functionality

SM has the role to ingest and guarantee the quality of data. Therefore, its pipeline includes: initialize storage, update storage and audit storage. The first 2 part allow ingest file into lake and provide govenance process, while the final performs audit task to improve data quality. Note that SM was designed with core concept is "hivi repo", and support binary ingest only (any extra developing will be reported in revision)

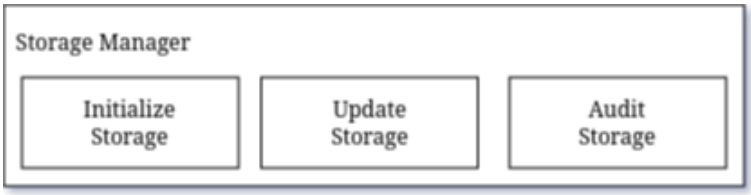

Figure 4: Storage Manager

Initialize Storage

Before the data can be uploaded into lake, the lake need to create a space to store them. And this logical block was born for that simple purpose - initialize a region for storing data. It should include 2 task:

- Verify the target directory empty.
- Generate sub-folder ".hivilake".
- Generate metadata schema and extra file.

Update Storage

This functionality block allow user pushing file into lake which go with its requirement metadata. The requirement metadata should be defined in the "Initialize storage" task. For speed up the ingest time, this task should be duplicated and run concurrently.

Audit Storage

If 2 first functional block is define and insert data into lake, this final block will confirm the reliability of data. It considers the metadata as the true ground and audit binary file following its metadata. Additionally, the block also support the whole repo audit (depend on hivi repo also). Therefore, the block will be present as:

Figure 5: Audit Storage

In the design, there are 2 part, the file audit to guarantee the data quality and detect wrong ingest. The other - repo audit - report profiling information of whole repository. Since SM using “hivi repo”, the audit process can be acclebrated by leveraging the central metadata file and concurrent processing.

SM Exception

Since this is the low-level issues, the report should not be addressed too deep. However, the corruption and exception can cause the crash and data lossing, so it is necessary to be mentioned before lower design happen. Therefore, instead of techonology dicussion, the report will try to provide some solution for avoiding the crash as well as recovery strategy.

First of all, the most of IO error is detected and auto-handled by the storage system it-self. So the system broken mentioning in here relate to the “hivi repo” incomplete task.

NOTE From now-on, the “repo” term will be used as the replacement for “hivi repo” for convenient writting.

Since the repo is not ACID transaction system, there are plenty of potential risk during running time. Like SM block can stop before finishish the transaction or SW is broken in middle of process. In that situation, there are 3 case can happen:

1. Metadata update, the binary file does not update / corrupt
2. Metadata does not update, the binary file update / corrupt
3. Metadata does not update, the binary file does not update

In first 2 circumstance, the information is still recored although it is incomplete. The system can depend on these incomplete information to recovery some data. Since the metadata file is the true ground, if it can be updated as case 1, it will not cause broken system. So the case 1 does not need any futher effort.

But in the case 2, the metadata is not update but the file is still added into lake. In that case, the new data push to system but is not logged into the metadata file. And since the metadata does not keep track information about file, the system does not know about its exists and the file is consider to lost. To avoid that error, the system can storing recovery infomation into the file name. Because any recored file in lake always has a name file. Therefore, the easy solution for file recovery is that leverages the file name (normally supporting up 255 character length) to save the recovery information in case the metadata crashed.

For case 3, since both data and metadata does not be saved into lake, the system consider it as a failure ingest and ignore.

The other problem can happen in concurrent case. Nomally, multi-thread can access and ingest data into lake at the same time as long as they provide different data with different metadata. But if there are 2 thread provide the same kind data with same metadata, the storage which is single recorder will be confused and raise exception. Despite of no damming, it become a bottleneck in the concurrent process. The simple solution can be a locking mechanism and allow 1 thread record at once time. However, a good locking mechanism is hard to implement in different storage system as well as the deplay time will be effect on the performance. So using the key constraint to distinguish each file will be the greatest solution. And because the concurrent process should not be limited by the relation properties as RDBMS, the key mentioned in here should be the value is agreed by anyone at anytime without any communication as time, location, user, ...

3.2.2 File Query

File Query provides the mechanism to search file, exploit file detail and download file. This utility provides the mechanism to discovery the data. And since the data discovery is the final purpose of the lake. It will need to satisfy several constraint (it should be proven and update):

- The search time should be fast enough for a good user experience even with large dataset.
- The search mechanism should have the capability to expoit all data with their detail in the lake with the smaller needed memorable task.
- The search language should be flexible enough to adapt with many different kind of user demand.

NOTE from now on, the File Query will be mentioned as FQ only.

Feature Design

From the key constraint, the report propose below design:

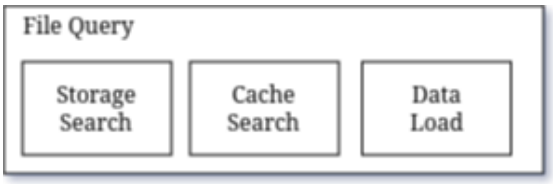

Figure 6: File Query

In this design, the system leverage 2 type of search: “storage search” and “cached search”.

The storage search is most useful search in a large storage. It apply search mechansim into whole storage and retrieve neccessary following a pattern. It is a real-time search with high accuracy. But since the search mechanism work on whole lake system, it is expected to a slow search with a poor search mechanism.

By contrast, the cached search is expected to a fast search with many diverse search types. The idea of cached search is “cache space”. Instead of search in whole lake, the search engine load a sub-set of information into a drum and apply the indexing process. And because the searched information is cached into a drum, the search performance will be improved through multiple searching time. However, because the drum is limited and it should be limited, the search mechanism is evaluate to be low accuracy and out-of-date.

Since a perfect search engine can satisfy all above key constraints is pretty hard to design or even with the current best practice search engine. Therefore, my idea use 2 type of search engine each satisfy a part of key constraint list and combine both into FQ. That is the reason of 2 search features in FQ and their opposite properties.

Beside to the search feature, FQ also enable user to access and get data by the “data load” function. It is only a simple connection into SW to open a data streamming.

NOTE Although this report provides detail of logical block, it is not expected to include the detail technology of each utility. Then the detail mechanism of each sub-block in FQ and any futher utility block will have their own report about the utility detail.

3.2.3 System Log

Until now, the system has enough functionality to become a basic lake system. And these utility was designed to work standalone without needed to know the existence of another utility. It increases the independency for utility layer but also causes the lack of system global vision. Unfortunately, when the system can not provide a global vision, it is hard to kept track and maintained. Therefore, the “System Log” utility is proposed to solve that issue.

NOTE From now on, the word “SL” will be used as alias to “System Log”.

SL is a special utility since it will be the first dependency in hivilake. Although both SM and FQ is designed to work well on their scope, they can not reach of expectation without the global vision, so the system decides to integrate SL into SM and FQ as the dependency.

Feature Design

Since SL was designed to enable a global vision of whole system. It is expected to keep track serveral information:

1. System log information
2. System identify information

If the system log information used to moniter the status of whole system, the identify information is considered as the global data which is accepted by whole system. The report proposes a simple SL design to statisfy the above list:

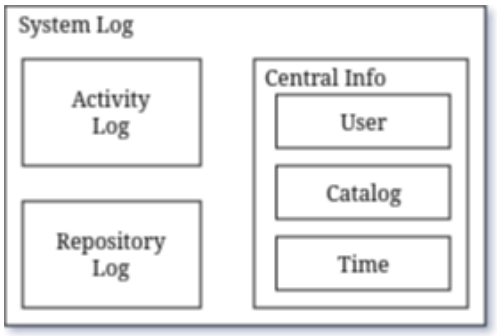

Figure 7: System Log Feature

The SL utility mechanism work as a global note system. On current version "0.1.0", it keep track 3 types of information. The "Activity Log" recods all activity in the system while the "Repository Log" work as the repository healthy status tracking. Finally, the "central info" save the identify information (includes user info, system time, category label).

Schema Design

Although SL act as a global notes, it should be stored in the tabular type because the tabular data is easier to control and monitor than normal text. Another advance is the schema on write property of tabular brings the consistency for SL and the system can use sql to manage them. For a bit unconvenient, the log schema should be design carefully from the beginning and will be hard to change after be deployed.

In this section, the system provides the prototype schema for SL (they can be update or change in another revision):

Log Detail

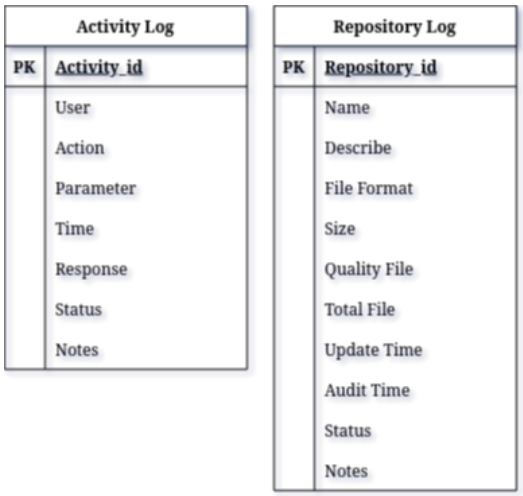

Figure 8: Log Schema

Central Infomation

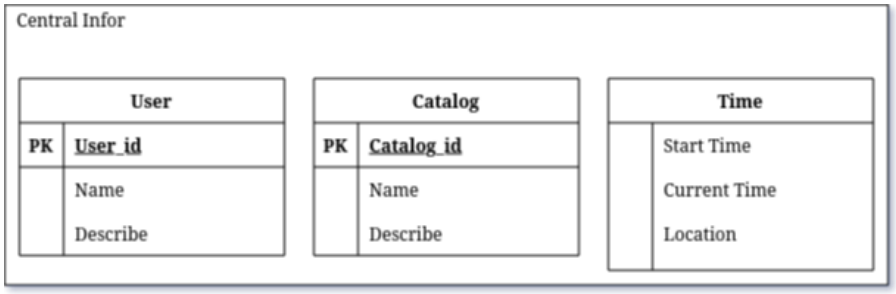

Figure 9: Central Information

In this prototype shema, the log detail provide the monitor data which will be update frequently. On the other hand, the central information part stores the identify information that will be agreed by whole system. These information can be appended and rarely updated through time.

3.3 API

The system provides list of utilities which can be use together to exploit the lake. But the system has not yet had a interface for outside user to access and use those utilities. Therefore, we decide to provide a API layer that allow the user can access into the system utilities. API (application programming interface) will define a set of routes and each route serves a service task. However, each organization has their own business requirement, so the api should be depended on the specific business rule providing by the company.

API Feature

Therefore, the api layout should be structured like:

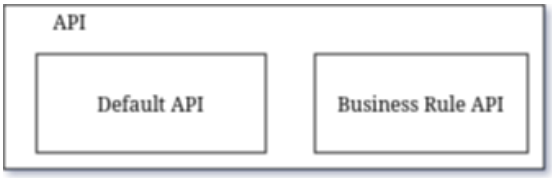

Figure 10: API Feature

The design has 2 blocks, the “default api” will load all api provided by the utilities vendor. It provide directly accessing to the public features of utility. In that way, anyone can load the utility and use them without care too much on the routing configuration. By contrast, the organization who want to customize the api to provide their expected service, they can design their own api rule and load it through the “business rule api” mechanism.

The propose api technology for this project will be HTTP base api like Restful API or RPC API like gRPC. The HTTP base API will allow a friendly interface and integrate into many difference system. On the other hand, RPC API will be harder to integrate into another system, but it will be a fast and light API which will have a better performance comparing to HTTP base API. And for this moment, we prefer the Restful API than gRPC since it is similar with us than gRPC. But the futher discussion will be raised up and the final result will be announced on another report.

4. Synthetic Logical Diagram

After the analysis and design each component, the final detail logical diagram will be proposed below (it maybe update in future revision report)

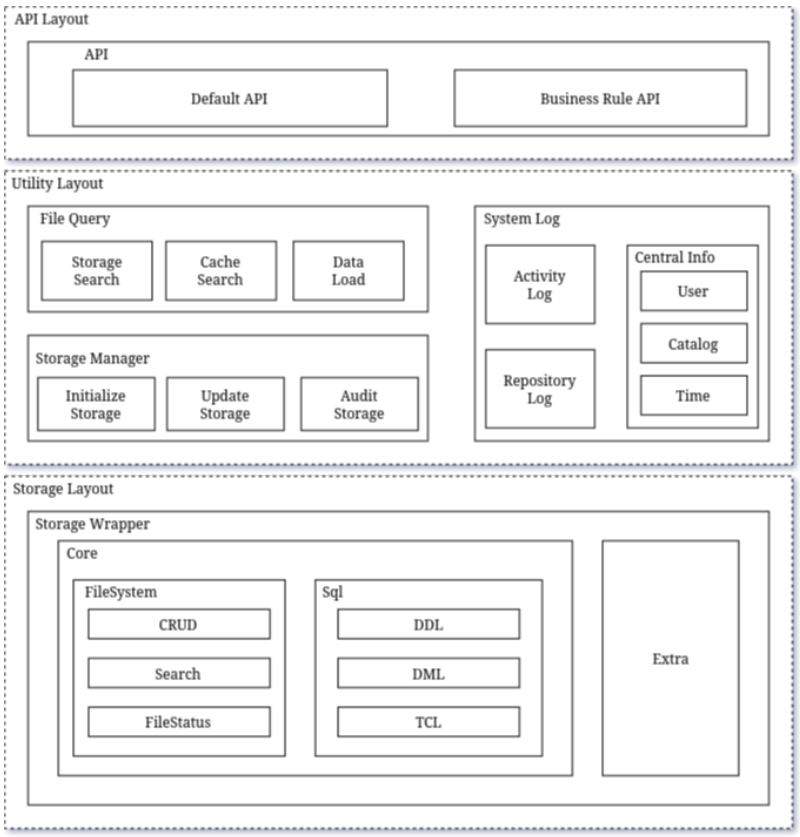

Figure 11: Logical Synthetic Diagram

5. Terminology

This section introduces acronym in the report for a easier reading time.

1. SW: Storage Wrapper - the logical block of Storage Layout
2. SM: Storage Manager - the logical block of Utility Layout
3. FQ: File Query - the logical block of Utility Layout
4. SL: System Log - the logical block of Utility Layout