PicDB: A Tag-Based Feedback Community System

Cheng-An Feng
National Taiwan University
Taipei, Taiwan
b06705056@ntu.edu.tw

Ying-Chiao Chen
National Taiwan University
Taipei, Taiwan
b07901184@ntu.edu.tw

Chang-Ting Tsai
National Taiwan University
Taipei, Taiwan
r09942038@ntu.edu.tw

Dao-Jan Chang
National Taiwan University
Taipei, Taiwan
b06902045@ntu.edu.tw

Abstract

In this report, we propose a tag-based management system that can automatically update and guarantee the high quality data. Based on MongoDB, our system now focuses on image data. When uploading image data, users need to give some description for it. Once the image is uploaded to database, users can insert tag to the image for what they consider it to be. For the images that are tagged, users can give feedback to the tag no matter it is good or bad. Until the feedback of tag is greater than a threshold, our system will put the image to high quality data pool. The data which are in high quality pool will be preferentially selected if users want to download data from our system. With this management system, users do not need to worry about correctness of data and can enjoy the high quality data from this system.

CCS Concepts: • **Software and its engineering → Search-based software engineering**.

Keywords: Database Management System, Data Catalog, Data Quality, Image Database, Tag-Based Query

ACM Reference Format:
Cheng-An Feng, Ying-Chiao Chen, Chang-Ting Tsai, and Dao-Jan Chang. 2021. PicDB: A Tag-Based Feedback Community System. In *DBMS '21: Database Management System Final Project, June 29, 2021, Taipei, Taiwan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Due to the fast-paced nature of technology, data becomes an important factor for every company. As AI becomes more popular, company requires more data to help them train model or carry out market analysis. While the data becomes larger than before, the data might be messy if there is not anyone to manage the database. Therefore, a good data management system can make company efficiently find the data they need.

In this work, we design a data management system that can automatically select high quality data through the tagging mechanism. Our system now aims to image data. For upload function, users can choose to upload one image or a folder of images. To record the source of data, users are asked to enter some description of the data. System will save the information of the image and the person who upload data. While images are uploaded to database, users can insert tag to them. Since the image may have a lot of descriptions for it, users are allowed to insert tag to image that already owns the tag. Our system also provides database summary for the user who access the database first time. Feedback mechanism is applied in our management system. Users can give positive or negative response to any tag. Once the positive feedback of the tag achieves the threshold, we consider the tag of data to be a high quality tag which represents high accuracy of the data. When users download the image, our system select data with high quality tag first. It prevents users from getting wrong data. Moreover, we adopt cache function on client. Cache is created at client and records the **Objectid** of the image after users first download the image with specific tag. If users lose some images that are downloaded before, our system use cache to help users retrieve the same images.

For next section, we share about some research and motivation of this project. Next, we introduce the detail of each API and our technical idea. Then, we talk about some future work to improve our system. At last, there is a conclusion of our project.

2 Motivation

Data are generated in an increasing rate today. Therefore, how to process data in an effective and efficient way has coming to everyone's sight.

To solve the problem, many companies use data warehouse for data versioning coupled with powerful analytic

DBMS '21, June 29, 2021, Taipei, Taiwan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$00.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

tools. Several years later, the semi-structured and unstructured data become more and more important for gaining more insights. To store these type of data without modifying the current architecture, many adopt data lake, placed between the data source and the business analytic tools, serving as a place to store data regardless of their type and origin.

The data lake architecture gains its popularity because its separate the "create" and "use" phase of data, which makes it really convenient for data provider to generate many data as they want. Nevertheless, this architecture seems not be the final solution. An additional layer between the source and the analysis add more complexity for data processing. Data engineers have to do more work to provide a seamless data flow while maintaining the consistency and replication between many storage places.

Thus some want to combine these two architecture into a single one, which is proposed as Lakehouse[8]. They introduced this new structure to cope with the problems mention above. They provide their own solution, called Delta Lake[7]. However, this still cannot fully combine the benefits of two.

The core of out project is to implement an data lake system that can have some features to boost ML data processing. A common approach is to implement a feature store [1] that stored curated features for an organization. To extend the feature store, we have to build some related systems, as mentioned in this article[2].

After weeks of survey, we have an interesting idea in mind. The main concept comes from the Google's GOODS[6] system. They propose a post-hoc data catalog system with some data quality enhancement mechanism while implementing several useful functionalities such as searching tools. We apply this idea on our project as well.

In the GOODS system, they use quite a few techniques to construct the relationships among tables to achieve data catalog. However, for unstructured data like images, audios, or just text strings, many of them become invalid.

The same problem can also be found in several data catalog system, like Marquez[5] or Amundsen[3]. They provide a wealth of features, but still not suitable for an unstructured data type.

Another inconvenience arises from the data source for ML training. Kaggle[4] is a popular machine learning and data science community. This community provides a bunch of images for data science use. However, for an user to get images he wants, he have to do many tedious search by comparing which dataset is what he really wants. In most of the situation, the user just want some "type" of images to use rather than having some many irrelevant features.

Based on the above two problems, that is why we want to implement a tag-based images sharing community that can benefit a wide range of people. By simplifying the data retrieving process, the developer can focus on find the insight behind the data instead of the trivial works.

3 Features

3.1 Upload APIs

Users can upload their image or a file of images that produced from their projects. They only need to specify three fields of the image information, "Description", "Uploader" and "Tags". The messages will be created and then recorded in the MongoDB for further usage.

3.2 Download APIs

We provide 2 download APIs for users. One is **get_images()**, which handles the images fetching process. The other one is **move_images()**, allowing users to move images from internal image pool to given user's folder. Notice that we have an image pool for managing images data. The user should not touch the image pool directly, which is managed automatically by the program, but only through our APIs to load images for later use.

The **get_images()** API let users get images directly from database or use local cache images instead. Users can set some parameters that specify the properties to choose from, and the function will construct related queries to fetch data from remote database or local cache.

Users are also allowed to label a cache version. Then users can leverage the labeled cache to identify different version of caches, then to do experiments multiple times with the same dataset in the local cache rather than to query database each time to save database system's workloads and network bandwidth.

The **move_images()** allows users to move those images in some cache version to another directory. This action is considered a better practice. One advantage is to prevent data from corruption by separating download part and user's use part. Another is to make the implementation more simpler and clean.

3.3 Tagging APIs

In tagging part, there are four APIs that are provided for users, including **insert_tag()**, **show_summary()**, **show_information()** and **show_image()**.

The **insert_tag()** lets users tag the image with specific ID. Besides, it also records the information that who tag the data. If users consider many images to be the same tag, they can just split the ID with space to tag more than one image.

For the users who need to have the summary of database, the API **show_summary()** gives them the top 3 tags and users who frequently give the tags.With this brief information, users can quickly understand the overview in the database.

For the users who want to get the information of the image in database, we provide two APIs **show_information()** and **show_image()**. The **show_information()** API shows information of the image with specific ID. Users can see all tags in an image and also the log that who add the tag. The

show_image() API can visualize the image in database. It can help user check the data content before inserting the tag.

3.4 Feedback APIs

Users can give their feedback about the tag of pictures. Every picture records the number of times it is tagged by every kinds of tag. Once the number of some tag of a picture exceeds the threshold, the picture is added to the corresponding high quality pool.

4 Tagging Concept

We use tagging for our system to perform the APIs mentioned in the previous section. We rely on **feed_back()** and **modify()** function to make our system perform better, but there are still things to discuss.

4.1 Freedom

Since we allows users to add or modify tags, we maximize the flexibility of tagging. But, it may come up with some situations.

First, many different words to describe the same thing or same tag. For example, "Elevator" and "Lift", "Garbage" and "Rubbish", and so forth may complicate the system and lower the performance which is not getting the expected data. Second, plural and singular tag for the same thing can cause the same issue.

This may be solved by creating pools for tags and stemming. The user or project needs to choose tags in a pool to add or modify or add tags into a specify pool. For example, assume that there is a pool called "Cats", then "Small Cat" tag, "White Cat" tag and so on, can be stored in the pool based on users query for future usage. Thus, users can find all data and don't need to worry about missing something that is expected.

Another solution may be using machine learning to create connection between tags. Two similar images may have similar traits and then tags may has close relationship. If the system use CNN or other machine learning models to compare new image with the highest-credits image for every tag, the relationship between tags may be created.

4.2 Spamming

Since the tagging system is open and rely on every users, they may add inappropriate number of tags or add misleading tags. Beside using negative feed back via our API, human detecting or spamming algorithm may com in handy.

These solutions are not the most efficient or the best, but may intrigue others for practicable ideas.

5 Implementation

5.1 Download APIs Parameters

Below shows some important parameters in our download APIs.

For **get_images()**:

- **tags**: The tags combination used for fetching data.
- **use_cache**: An option to use local cache or get images directly from the database. The default value is True.
- **cache_version**: If used cache, the version of cache to get images from.
- **next_cache_name**: If to create a new cache, the name of the new cache.

Only the **tags** argument is required, others are optional but have default values.

For **move_images()**:

- **tags**: The tags combination used for fetching data.
- **dst_path**: The destination directory to move images to.
- **cache_version**: The cache version to use. The default value is 0 (latest).

The **tags** and **dst_path** arguments are required, others are optional but have default values.

5.2 Images Management

All images are saved under a specific system-defined path, which is invisible to users. In the image pool, each image is named after the **ObjectId** in the MongoDB database, preventing filenames collision. The images pool contains all images from all tags combination.

5.3 Images Queries

A query can be divided into 3 stages.

In the first stage, we either fetch images from the database or use the local cache. If the user chooses to query from the database, the API will first check if the index, which is the high quality pool of images, of that tags combination exists. If the index does not exist or the index has no enough images, the program then try to get other images from the default image pool in the database.

In the second stage, we compare the query images list by performing intersection set operations to filter out those had already been downloaded. Then we get a list that we should download to satisfy user's request.

In the last stage, we will construct a query based on the images id list we obtain in the previous stage to get images content directly from the database.

5.4 Cache Management

Cache is extremely significant in our system, which not only saves the resources from repetitive downloading, but also facilitate the further use.

Once a user gives a tags combination, the program creates a cache folder whose name is a string that is concatenated with "-" in the sorted lexicographic order. For example, with a tags combination ["meme", "cat", "orange"], the cache folder name is "cat-meme-orange". This naming methods ensure a unique name for each tags combination. We assign each tags combination a folder under a root cache folder for easier management.

Under each tag cache folder, we maintain several cache files, which contain the images id list and some other metadata. Each cache file is named in the "version-name" format. To make matters concrete, there may by some cache file like "1-exp1", "2-test2". For each query operation, the user is able to choose whether to label the cache. The labeled cache will be saved under the tag cache folder for further use. A special one is "0-latest". It is the most recent cache that user queried. This file serves as a temporary file if user forget to label the cache.

5.5 Move Images

We split apart the download part and the use part. Thus, users have to call the **move_images()** API to use the images. This function is just to move given images to the user's destination directory.

Users have to specify a cache version to use, where each cache version is assigned a unique integer number. As we implemented in the downloading part, the move image API will check if specified images had already been moved or not. However, we will not delete those images that already exists even if it does not in the cache version. We choose to implement in this way because we assume that users only want some type of images rather than a totally correct number. The user have to automatically delete the whole directory if he want to perform a completely different operation that use the same folder as the data source.

6 Discussion

In this section we discuss some features that are important but not yet implemented.

6.1 Implementation Improvement

In out current download implementation, we save images and cache in folders, which can cause several problems.

First, we save images in a single image pool, and the program compares the images to decide which images to download. This encounters a severe scalability problem because each time the user construct a query, we have to walk through the folder to get the whole images list. Second, in our assumption, the image database located at a remote server rather then local machine.

The better implementation is to insert the images into a local database, MongoDB in our project. In this way, we have to maintain two connection handler, one for remote image server, the other one for local cache database. We can insert the exact same document including the image content into the local database because the _id is unique, which will not conflict each other. We still can implement cache versioning, but to map the current tag cache directory to a collection, the cache files to documents in the local database. This alternative implementation becomes really simple and reliable, what we do is to provide a middleware that hides complex operation behind the scene.

6.2 Tags Improvement

We assume that tags are all independent in our project. Nonetheless, some tags may have relationships. For instance, a image tagged as "cat" may not be tagged as "dog", but one can be independently tagged as "orange".

A possible solution is to construct a relationship graph that holds relationships between tags. Note that we maintain the relationship between "tags" instead of "images" themselves. It it because each image can be categorized to some of the tag groups, therefore the problem is reduced to find the relationships between tags.

For each tag creation or destruction, we examine the existing graph and try to insert or remove the tag by adjusting the structure of the relationship graph. When users want to insert a new tag to a image with an assigned tags already, we must resolve the relationship to decide whether this new tag can be added. This operation depends on the tags credit and other factors.

Another way for building this graph is to use some existing language database that have semantics for many words. The difficulty of this approach is to convert the semantics to the relationships we want.

6.3 Tags Hierarchy

The goal of our project is how to build a model to retrieve images given only information of tags combination. One difficulty we face is to construct an efficient model resolve the problem of tags hierarchy.

Although we can identify the relationship of a tag, it is not entirely the case in a data query. For example, given a tags combination ["cat", "meme", "orange"], it may be easy to see that "orange" is an auxiliary tag, but "cat" and "meme" is hard to tell which one is more important. To make things worse, the case differ in different situation, in some cases "cat" may be more important, but in other cases the "meme" is what user really want.

This difficulty affect the model of hierarchy caching. We can not guarantee the ordering of a tag combination, the only possible solution is to use a flat structure, that is why we implement the database with only one image pool and only one-level cache. Or we will encounter serious data replication and consistency issues.

6.4 Security

Another difficulty in designing this system is to prevent adversaries from corrupting the whole database.

The approach we take is to introduce the feedback mechanism that reach consensus among users. This idea comes from the Google GOODS system. By users' cooperation, the images with high quality tags can bubbled up to the surface. If someone add a lot of trash tags, the tags will automatically be removed by the feedback system.

We also have to do source control. By limiting the feedback times of each user, we can contain this bad insertion. This constraint should be put on the server to prevent from further client-side program exploitation.

An alternative approach is to set up several authority nodes responsible for tags manipulation. However, this will violate the principle of the community because it make it harder for an user to directly send feedback. This approach nevertheless need further research to figure out a possible solution that can strike a balance between each side.

6.5 Scalability

Though our data model seems not quite efficient, we should not be so pessimistic. The simple model usually can leverage the power of parallel processing to improve the performance.

The set operation model does not require a global consistent view among all users. What users want is to retrieve some images they want. The database can apply replication and sharding in a large scale. Then when a user fire a query with several tags, we run the query for each tag in parallel, then to combine then with set intersection.

In this way, we not only can add randomness to the query process, but also make break the bottleneck for the most time-consuming part, the data retrieving process. The set intersection can be perform very efficiently in the modern programming language even the scale is large. The time complexity of set intersection is O(n).

6.6 Images Priority

As the time passed, the database will be populated with more and more images, but many of them will even never be exposed to people's eye on the account of the index mechanism we say before.

It is not a good thing for it will strangle the potential of the community. A possible solution is to labeled the image as "newbie" in the first month it is uploaded to our system. For those "newbie" images, the system will increase the priority of these images to become more easily to be chosen.

By introducing the priority, the system can manipulate images in a more flexible way. This idea can be generalized to not only the new images, but those images that has less accessiblity. You can think of this operation as "stiring" the database. In the long run, all images in the database will be treated what it deserved.

6.7 File Types

We focus only on the image field right now, but it is possible to expand the data types to audio, video, csv and others. Since we are using tags to query and implement, the file name, the file type and file content will not be the first consideration compare to tags. So, the goal of containing many different file types is feasible and can make the data base more useful.

6.8 Additional Data Process

In additional to plain image retrieving process, we can apply some advanced image processing functionalities that directly built in our system. For example, we can construct a data pipeline system that integrates image uploading, resizing, retrieving, with machine learning functions included as plugins. Further more, an user-defined automatic script can also be inserted into data processing to make the system become even more convenient.

7 Conclusion

In this report, we present an automatically data management system that provides high quality data. Our technical idea focus on the tag of the data with the feedback mechanism. By this mechanism, system can automatically identify high accuracy tag of the image. Users do not need to worry about the data quality from this system. As more users access the database, we believe that our system will be more powerful, allowing people easily get the data with precise tag.

8 Work Contribution

- **B06705056 Cheng-An Feng**: Demo video, Upload API, Report
- **B07901184 Ying-Chiao Chen**: Download APIs, Report, Survey Paper
- **R09942038 Chang-Ting Tsai**: Tagging API, Report
- **B06902045 Dao-Jan Chang**: Feedback API, Merge APIs

References

[1] 2018. Hopswork Feature Store. https://www.logicalclocks.com/blog/feature-store-the-missing-data-layer-in-ml-pipelines.
[2] 2020. Feature Store as a Foundation for Machine Learning. https://towardsdatascience.com/feature-store-as-a-foundation-for-machine-learning-d010fc6eb2f3.
[3] 2021. Amundsen. https://github.com/amundsen-io/amundsen.
[4] 2021. Kaggle. https://www.kaggle.com/.
[5] 2021. Marques. https://marquezproject.github.io/marquez/.
[6] Natalya F. Noy Christopher Olston Neoklis Polyzotis Sudip Roy Steven Euijong Whang1 Alon Halevy, Flip Korn. 2016. Goods: Organizing Google's Datasets. In *SIGMOD/PODS*.
[7] Armbrust et al. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. In *Proceedings of the VLDB Endowment (PVLDB)*. 3411–3424.
[8] Reynold Xin Michael Armbrust, Ali Ghodsi and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Annual Conference on Innovative Data Systems Research (CIDR)*. 11–15.