Software Engineering: A Generic and Scalable Data Reconciliation Pipeline

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Emily Huo

Fall, 2022

Rich Nguyen, Department of Computer Science

Software Engineering: A Generic and Scalable Data Reconciliation Pipeline

CS4991 Capstone Report, 2022

Emily Huo
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
esh2nne@virginia.edu

ABSTRACT
A California-based technology company's enterprise platform needed a new automatic and generic way to reconcile data discrepancies that would replace its old manual process. In order to solve this problem, I designed a generic and scalable data reconciliation pipeline to process data based on use case to identify data discrepancies and notify online services to reconcile the discrepancies. I implemented this using gradle, spark, Hadoop, scala, kafka, and java. I successfully designed the data pipeline to be generic so it could be used for multiple different use cases and developed and tested the first part of the pipeline. In the future, the rest of the pipeline will need to be implemented and tested as well.

1. INTRODUCTION

The increasing amount of data collected by companies means that there is also an increased chance of data becoming inconsistent, which impacts business. For example, if a client is getting outdated information, this could severely impact decisions made and cost the company a lot of money. One reason this happens is because of the difference between online, nearline, and offline storage. Online data is immediately available, while nearline and offline is not. There is a time fetch difference between online and nearline, as well as between nearline and offline. Thus, online data is the most up-to-date, while nearline contains the second most recent, and offline the most outdated data. The syncing between the three causes inconsistencies.

Another common reason for inconsistent data is business logic. Due to these inconsistencies, there must be some way to compare and verify data. The brute force way is to manually verify by going through the databases and checking each condition by hand and then fixing the inconsistency by making the entries in the database match what should be stored. However, this is extremely time consuming, so I devised a process to replace these manual checks.

2. RELATED WORKS

The work that was most relevant and that heavily shaped the solution was a pipeline already built to check for out-of-sync instances between two different platforms in the company. It was an automated pipeline that would check between two databases because the services were in the process of migrating from one platform to another. The pipeline caught inconsistencies during the migration and then fixed it so that the databases stayed in sync. The difference between the migration scenario and this problem is that the definition of out-of-sync for the migration is well defined and has a set number of types. This makes it easy to hard-code the logic to catch the different instances. However, for the new scenario, the pipeline had to be generic and scalable because it will be expanded upon and used for verification in the future and use cases are unknown. In order to generalize the pipeline, I conducted research into data types that would be generic and selected an AnyRecord defined within the company to achieve the desired generalizability.

Kafka was also chosen to be a crucial part of the pipeline after investigation of related works. In Auradkar, et al, (2012), kafka is defined as a scalable and efficient messaging system for collecting various user activity events and log data. It adopts a messaging API to support both real time and offline consumption of event data.

In Haseeb, et al (2017), kafka is used as the message queue. The paper recommends the use of Hadoop and Spark which will vastly improve the performance of data stream processing pipelines. Review of this literature strongly influenced my own project design.

3. PROJECT DESIGN

The project had very specific requirements in order to serve the enterprise platform team. These specifications shaped the project design and were at the forefront of the all the design decisions made.

The pipeline was designed in three parts: offline data processing, kafka push job, and data validation and reconciliation.
The first part, offline data processing, consisted of a gradle file that contained a Hadoop Spark job that calls a scala file which defines data inconsistencies for that particular use case. It goes through the specified database tables and cross-references the data entries using the logic defined to see what data entries violate the definition. It writes all the data entries that violate the logic to another table. The gradle file then sends a kafka message using the kafka push.

The second part of the pipeline, the kafka push job, has the gradle file push out a kafka message which is designed to be generic using the AnyRecord field. The AnyRecord field is filled out by the sender to contain the information needed to identify what type of data inconsistencies were just caught. As this pipeline gets scaled up to catch more use cases, the kafka push job receiver needs to know which one in order to be able to call the right reconciliation logic. The kafka message gets sent by the kafka push.

The third part of the pipeline starts by receiving the kafka push message. This file then reads the message to find where the use case is defined and calls the corresponding reconciliation file. This reconciliation file is written in java and is made up of two parts. The first part is the validation portion. Validation is crucial in this project because since the data processing is done offline, by the time the validation process is called, the data could have been changed. To ensure that this data entry is still inconsistent and should be reconciled, the data first goes through validation. After validation, the logic that reconciles the data is called.

4. RESULTS

This project was selected to be completed that quarter because of how significant the impact would be on the enterprise team. Before the creation of this pipeline, different projects on the team would need to be verified manually. For example, the first use case that the pipeline was built from is a validation case between the roles and the seats for a certain type of role.

There were instances where the role was defined but the corresponding seat did not exist. This would be a case where the role and seat were out of sync. The brute force way of validating and reconciling this problem would be as follows:

1) The engineer that was working on this project would perform a routine maintenance check of the databases using a script that was written that defined the inconsistencies.
    a. This would be run by hand weekly and the data entries that were out of sync would be identified.
2) Then, the engineer would go in manually and reconcile the difference by adding in the seat for the role.

Now, with the creation of this pipeline, this weekly manual check and reconciliation are no longer needed. The pipeline runs automatically and depending on the specification of how frequently this use case would be run, it would do all of this automatically.

5. CONCLUSIONS

This data reconciliation pipeline was marked as a project of high importance and was to be completed by the end of the quarter. Since this pipeline is designed to be scalable, it will have a significant impact on the entire enterprise platform team by automating work that was otherwise being done by hand. All of these manual checks were done frequently, causing the company valuable working time. With the creation of this generic pipeline that can be customized for each specific use case, a lot of time, effort, and money can be saved and used for other more useful projects.

6. FUTURE WORK

There is one additional phase of the pipeline left to be completed in the future: a reporting feature. This will be added throughout the pipeline in two different parts. The first part will be in the first section of the pipeline, offline processing. An email report will go out after the data inconsistencies are identified. The email will report metrics such as how many instances of each use case were found.

The second part of reporting will be in the last part of the pipeline, data validation and reconciliation. Similarly, an email report will go out after the reconciliation is performed. This email will also contain metrics similar to the first email such as how many instances of each use case was successfully reconciled and fixed. The email report can be used by the enterprise platform to monitor the health of the reconciliation pipeline.

REFERENCES

Auradkar, A. Botev, C. Das, S. Maagd, D. Feinberg, A. Ganti, P. Gao, L. Ghosh, B. Gopalakrishna, K. Harris, B. Koshy, J. Krawexz, K. Kreps, J. Lu, S. Nagaraj, S. Narkhede, N. Pachev, S. Perisic, I. Qiao, L. Quiggle, T. Rao, J. Schulman, B. Sebastian, A. Seeliger, A. Shkolnik, B. Soman, C. Sumbaly, R. Surlaker, K. Topiwala, S. Tran, C. Varadarajan, B. Westerman, J. White, Z. Zhang, D. Zhang, J. 2012. "Data Infrastructure at LinkedIn," 2012 IEEE 28th International Conference on Data Engineering, 2012, pp. 1370-1381, doi: 10.1109/ICDE.2012.147.

Javed, M. Lu, X. Panda, D. 2017. Characterization of Big Data Stream Processing Pipeline: A Case Study using Flink and Kafka. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT '17). Association for Computing Machinery, New York, NY, USA, 1–10. Retrieved November 27, 2022, from https://doi.org/10.1145/3148055.3148068