Design Document: Legacy Database Migration to AWS Data Lake

1. Introduction

This document outlines the design for migrating legacy databases to AWS data lake. The goal is to develop a cost-effective, efficient, and scalable pipeline that syncs data to AWS data lake and ETLs it into a datastore to serve as a data source for new applications being built. The migration needs to happen in a way that ensures data consistency between AWS and on-prem data sources.

2. Requirements

- Migrate data from on-prem legacy databases to AWS data lake.
- ETL data from data lake into a datastore (e.g. RDS, Athena) for new applications.
- Develop a pipeline that ensures data consistency between AWS and on-prem data sources.
- Design a cost-effective and scalable solution that can be operated by a small ops team.
- Identify an appropriate delay for data syncing between on-prem and AWS data sources.

3. Assumptions

During the migration of the database, the following assumptions were made from observation, listed from major to minor ones:

- The .xlsx file is the only source of the data.
- The current .xlsx file serves as the initial point of the database, and there will be more updates to that file for synchronization.
- In production settings, the file may be significantly larger, and so PySpark is chosen over vanilla python for the ETL purpose.
- The use case for the database is more of an OLTP (online transaction processing) scenario than OLAP (online analytical processing), so RDS is preferred over Athena.
- The 'borrower_id' column on the role_profile sheet is identical to the 'id' column on the borrower sheet.
- Email type refers to the domain of an email address.

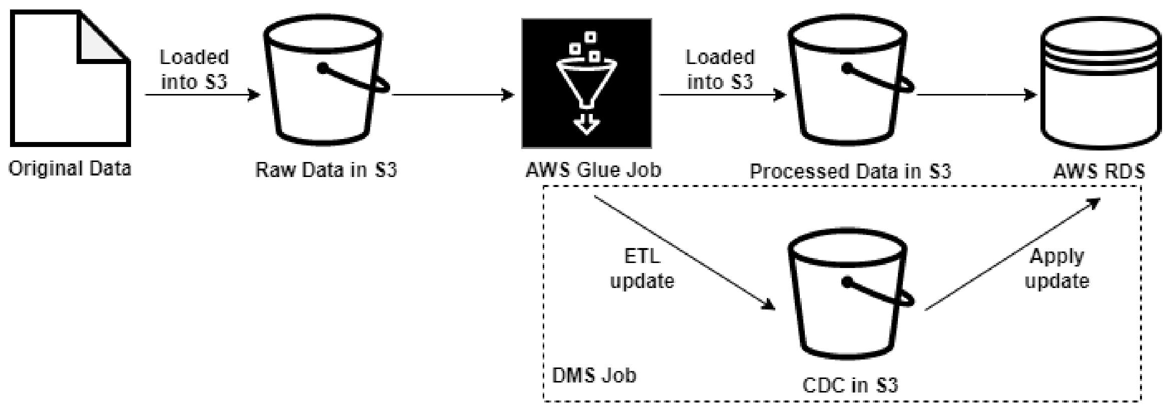

Fig 1. Architecture Diagram for Data Migration Job

4. Proposed Solution

To achieve the above requirements, we propose the following design and the architecture diagram is also included in Fig 1 above:

- Data Extraction:
  We will first extract data from on-prem databases to an existing AWS S3 bucket for further processing.
- Data Transformation:
  Once the data is in S3 buckets, we will use AWS Glue ETL to transform and clean the data, specifically transform the .xlsx file into several .csv files, where each is a table in the schema provided. AWS Glue ETL is a fully managed ETL service that provides an easy way to transform data in S3 buckets to a format that can be easily queried by a datastore.
- Data Loading:
  Next, we will load the transformed data into a datastore. We here use RDS Mysql as a datastore. Trade-offs for choosing databases are documented in the Trade-off section.
- Data Consistency: (To-Do)
  To ensure data consistency between on-prem and AWS data sources, we propose using AWS DMS change data capture (CDC) functionality. CDC will enable us to capture and replicate changes made to on-prem data sources to AWS data sources in real-time. Currently, this feature is not practically implemented on AWS, as there is no update to the original files.
- Cost-Effective and Scalable Solution
  To keep low costs and high scalability, we propose using AWS managed services as much as possible, which provide a cost-effective and scalable solution that requires minimal maintenance and configuration. We have used Glue ETL ($0.44/hr), RDS Mysql (free tier) and a t3.micro EC2 instance ($0.0104/hr) as the database client so far. But Glue ETL cost is only incurred on use. So the total operational cost would be minimal to host the EC2 client. If we need to deploy DMS in the future, the cost would still be small, based on the required workload instance.
- Delay for Data Syncing:
  The delay occurs both during ETL and loading into Mysql. Since we used PySpark for ETL and disabled primary key check during data loading, the delay time is minimal and acceptable. Notice that in this case, we update the database on a regular basis, i.e., every week. But the delay for real-time streams is not the same. If the update stream is real-time, then every update will go through the above pipeline and will at least incur minutes of delay, which is quite large for streaming scenarios. So, it is suggested we do a batch update instead of real-time update.

5. Trade-Offs

While developing the proposed solution, we made the following trade-offs:

- Pyspark vs python: We choose Pyspark to process the raw data instead of python, though the raw data is hardly a large file. We expect the real world scenario would have much larger data and consider the longer development time of Pyspark than python, so we still develop the ETL script based on Pyspark.

- RDS vs Athena: From the observation of the data, we assume this is more of a transactional system than an analytical system. We can easily insert, delete, query and update in RDS, while we can have complex analysis using Athena. Based on the nature of the data, we choose RDS Mysql as a proper database.
- Type of user_profile_id: In the raw data, a UUID is given as the user_profile_id. It is hard to generate a suitable (uniquely identifiable) and storable (smaller than 64 bit) integer from such UUID. So, we keep it as a 128-bit string, though it would be harder to create an index based on such a primary key in the user_profile table.

6. Alternatives

- **Alternative 1:** If we are aiming for real-time updates, we may set up a Kafka stream of the updates, and let the DMS read from the stream and apply the update. This requires a stable Kafka service and is more costly than the proposed solution.
- **Alternative 2**: It is hard to process the change of data into S3 bucket for DMS to synchronize, as the CDC files themselves are complex to compute. We can set up an on-prem database and ETL the raw data locally, instead of processing on AWS Glue. This makes synchronization of updates much easier. But it needs to expose the endpoint to DMS, which may have security issues, and requires reliable networking and high bandwidth.

7. Conclusion

The proposed solution provides a cost-effective, efficient, and scalable pipeline for migrating legacy databases to AWS S3. It ensures data consistency between on-prem and AWS data sources through AWS MDS and can be operated by a small ops team. By using AWS managed services, we can keep costs low and scalability high. However, we acknowledge that trade-offs have been made and alternative solutions may be appropriate depending on the use case.