Design Document

This document outlines the process of migrating legacy databases to the AWS cloud. It presents a cost-effective data pipeline for extracting, transforming, and loading the data, the tools to be used in the process, and alternative solutions.

Design Considerations:

Assumptions: The following assumptions have been made to perform the ETL process:

1. The columns “created_date” and “updated_date” has a default value of the current date-time when the table was created
2. The columns “created_by” and “updated_by” have a default value of “admin”
3. The type of email_id depends on the domain name. For example “@gmail.com” is of the type “com” and “@cm.edu” is of the type “edu”
4. The type of phone depends on if it’s a home phone or a cell phone. The values of the respective columns(phone_home, phone_cell) have been populated with their types

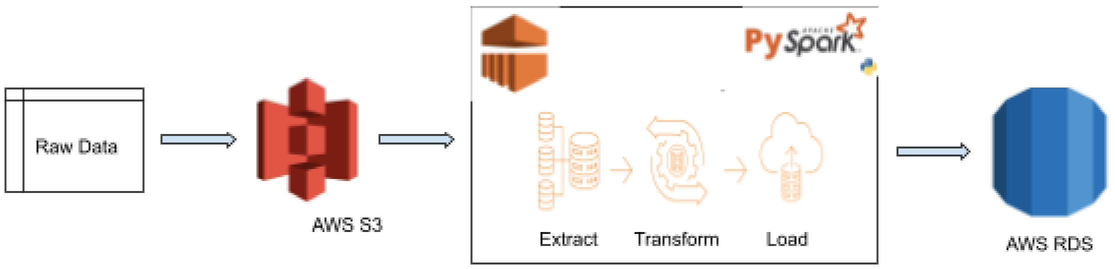
Constraints: The system is constrained by the following:

1. Cost: Considering the large amount of data that would flow in it is important to design a system that is cost-effective. Given that we have finalized AWS as the preferred cloud provider, the decision needs to be made on what technologies we would use for ETL
2. Scalability: The system needs to be scalable in the sense it needs to handle large amounts of data
3. Maintainability: The system needs to handle the changes in the schema easily

System Environment: The system would be deployed on AWS with raw data stored onto AWS S3. The ETL pipeline would be performed on AWS EMR, a self-managed service, with a flexible design and customization costing us around $0.046 per hour with m5g.xlarge instances. An alternative is AWS GLUE, a fully managed ETL service with automatic scaling, and easy integration that costs us $0.44 per DPU-Hour. To start with, I chose EMR as I was familiar with it, even though it wasn't included in the free tier. Although GLUE was included in the free tier, it allowed the scheduling of just one spark job per day and hence was harder to experiment with when compared to EMR. Moving forward we can either stick to EMR as it scales well for large data or switch to AWS GLUE if there’s no need for cluster customization.

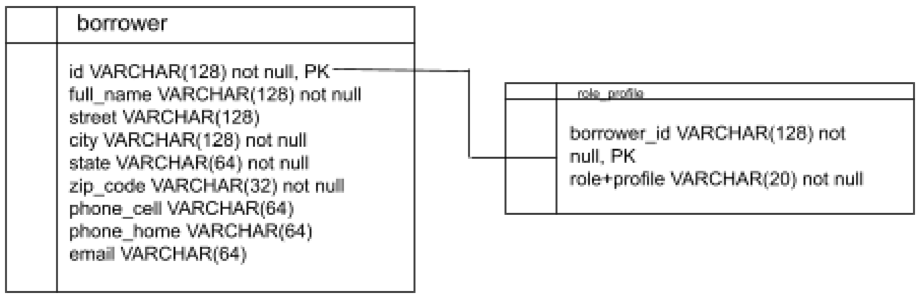
Design Methodology:

System architecture: The system processes raw data stored in an ‘xlsx’ file at AWS S3, transforms the data to the required schema through Spark Jobs, and loads it to a relational database, AWS RDS. The raw data is loaded onto AWS S3 using AWS CLI. This data is used by AWS EMR to perform the ETL process. The data is extracted into a spark data frame using pySpark and transformed into the schema described in Fig 2 using Dataframe APIs. The processed data is then loaded into Amazon RDS for persistent storage. A diagram of the components involved in this process is shown below:



1. Component Diagram

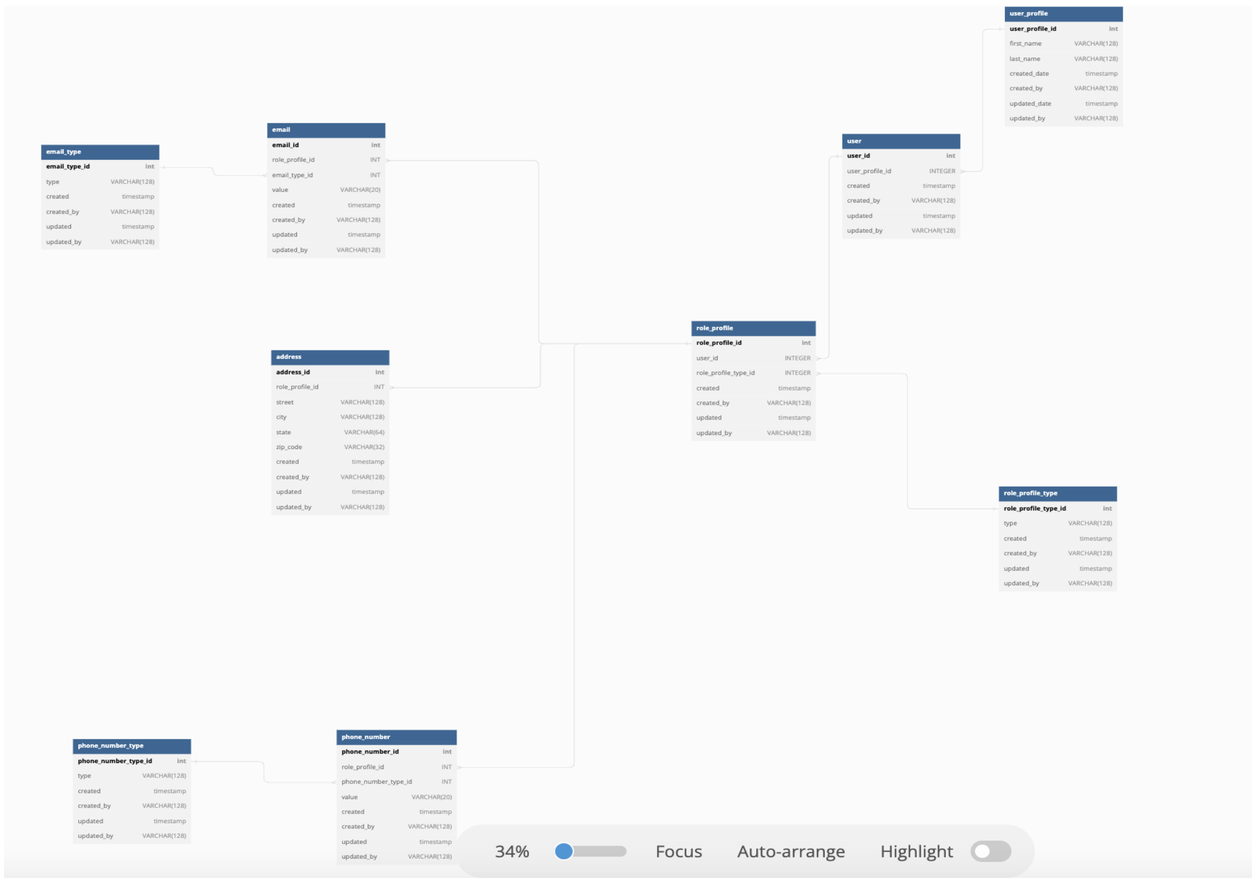
Data: The raw data is denormalized and consists of a highly coupled schema as shown below:



borrower
id VARCHAR(128) not null, PK
full_name VARCHAR(128) not null
street VARCHAR(128)
city VARCHAR(128) not null
state VARCHAR(64) not null
zip_code VARCHAR(32) not null
phone_cell VARCHAR(64)
phone_home VARCHAR(64)
email VARCHAR(64)

role_profile
borrower_id VARCHAR(128) not null, PK
role+profile VARCHAR(20) not null

Thus, in order to have a normalized schema for better efficiency, we transform it as below:



email_type	
email_type_id	int
type	VARCHAR(128)
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

email	
email_id	int
role_profile_id	INT
email_type_id	INT
value	VARCHAR(20)
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

user_profile	
user_profile_id	int
first_name	VARCHAR(128)
last_name	VARCHAR(128)
created_date	timestamp
created_by	VARCHAR(128)
updated_date	timestamp
updated_by	VARCHAR(128)

user	
user_id	int
user_profile_id	INTEGER
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

address	
address_id	int
role_profile_id	INT
street	VARCHAR(128)
city	VARCHAR(128)
state	VARCHAR(64)
zip_code	VARCHAR(32)
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

role_profile	
role_profile_id	int
user_id	INTEGER
role_profile_type_id	INTEGER
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

role_profile_type	
role_profile_type_id	int
type	VARCHAR(128)
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

phone_number_type	
phone_number_type_id	int
type	VARCHAR(128)
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

phone_number	
phone_number_id	int
role_profile_id	INT
phone_number_type_id	INT
value	VARCHAR(20)
created	timestamp
created_by	VARCHAR(128)
updated	timestamp
updated_by	VARCHAR(128)

The mapping from the old schema to the new is as follows:

New Schema	Old Schema

Table name	Field Name	Field name	Table Name
role_profile	role_profile_id	borrower_id	role_profile
role_profile_type	type	role_profile	role_profile
user_profile	user_profile_id	id	borrower
user_profile	first_name, last_name	full_name	borrower
address	street	street	borrower
address	city	city	borrower
address	state	state	borrower
address	zipcode	zipcode	borrower
phone_number	value	phone_cell	borrower
phone_number	value	phone_home	borrower
email	value	email	borrower

Please note that for all the tables, values of created_by, updated_by, created_date, and updated_date are set to default values as stated in the assumptions sections.

ETL Algorithm: Tables in the new schema have been populated using the values in the raw data based on the mapping described above. The additional details for each table with the columns that involve extra processing have been described below.

1. user_profile: The full name column has been split into two columns, first and last names based on the separating char “ “. For Example, “John Doe” would be split as John (First name) and Doe(Last name).
2. user: Links to the user_profile using the user_profile_id. Has a primary key “user_id” which is a monotonically increasing value starting from 1.
3. role_profile: Links to the user using the user_id. Has a primary key “role_profile_id” that corresponds to the original “id” in the raw data
4. role_profile_type: Consists of two types: borrower and co-borrower that has a corresponding id of 1 and 2 respectively. Links to role_profile using role_profile_id

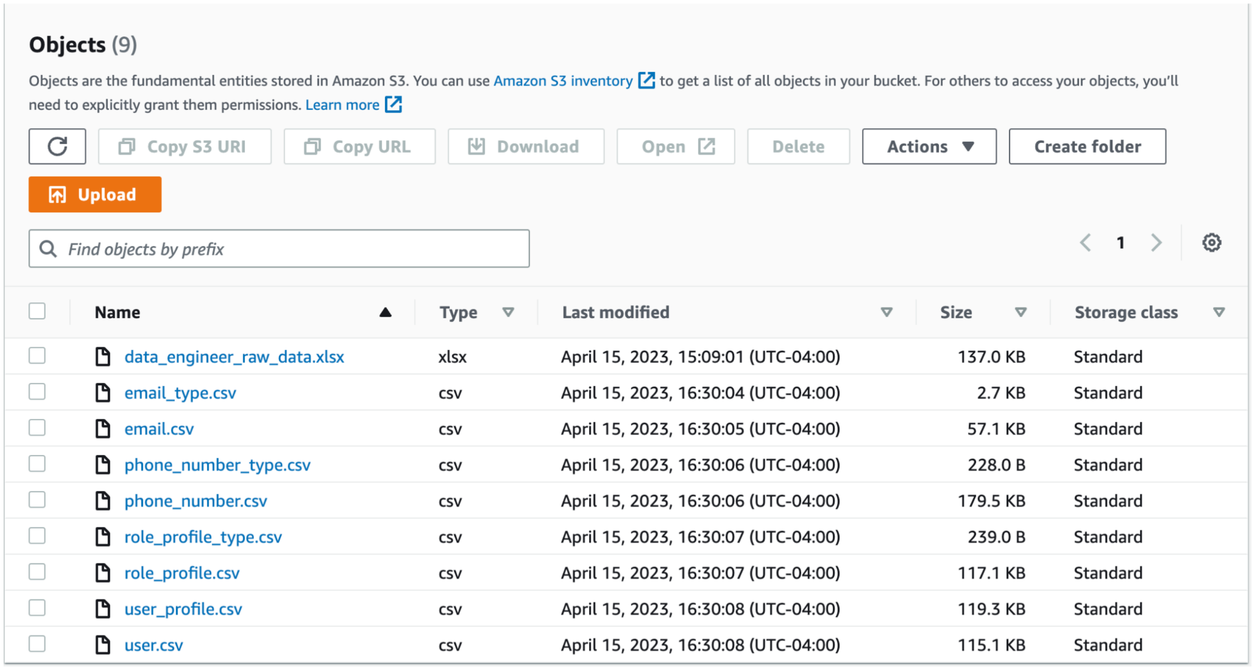
5. address: All the details are directly obtained from the raw data. Has a primary key “address_id” which is a monotonically increasing value starting from 1. Links to role_profile using “role_profile_id”.
6. phone_number: The columns “phone_home” and “phone_cell” are combined into a single column and the value stored depends on the type of phone which can be obtained from phone_number_type. Has a primary key “phone_number_id” which is a monotonically increasing value starting from 1. Links to role_profile using “role_profile_id”.
7. phone_number_type: Has two types: home and cell, with values 1 and 2 respectively. Links to phone_number using “phone_number_id”.
8. email_type: The type is determined by the domain name. Has a primary key “email_type_id” which is a monotonically increasing value starting from 1.
9. email: The value is obtained from the raw data and maps to one of the types stored in email_type. Has a primary key “email_id” which is a monotonically increasing value starting from 1. Links to role_profile using “role_profile_id”

Tools and Technologies: The following tools have been used in designing the system:
1. AWS: For deployment the chosen cloud provider was AWS. This is based on the company’s needs stated in problem statement
2. AWS EMR: As mentioned earlier, AWS EMR is very helpful in building scalable applications that are cost-effective
3. AWS RDS: Since the target schema is organized into tables, AWS RDS could be a great choice. It has support for MySQL, Athena, and other DB for future usage
4. PySpark: Spark is one of the most preferred languages for distributed processing. It can scale well with large amounts of data and supports Python, scala, R, and Java. Based on my prior experience, I have chosen pySpark to complete this assignment. Additionally, pySpark has a lot of community support with packaged library functions that avoid the overhead of coding everything from scratch

Testing: It is good to have a testing strategy defined for ETL early since re-doing these actions might be very time-consuming and costly.
1. Unit testing: Tests can be defined with custom cases after each transformation using pytest. The advantage is that we could also check the coverage of the test cases to get an estimate of our code.
2. Integration testing: We need to verify the system after the integration with various components. Check if the data is loaded onto RDS correctly. This could be done by writing a simple API to select and test it through Postman

Deployment: The steps involved in deploying the system to AWS have been listed below:
1. Load data to S3: Considering the current data size, the best way is to use AWS CLI to copy the data into S3. The command used is: *aws s3 cp data.csv s3://my-bucket/*
2. AWS EMR for ETL: Launch the EMR cluster and connect to Zepplin using the link given in the ‘Applications’ section of the cluster summary. Upload the ipynb notebook into

Zeppelin and run cell by cell to load and infer the schema, transform the data, and output the data back to S3.
Once this process is complete, the s3 bucket would have the following files:



Objects (9)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
data_engineer_raw_data.xlsx	xlsx	April 15, 2023, 15:09:01 (UTC-04:00)	137.0 KB	Standard
email_type.csv	csv	April 15, 2023, 16:30:04 (UTC-04:00)	2.7 KB	Standard
email.csv	csv	April 15, 2023, 16:30:05 (UTC-04:00)	57.1 KB	Standard
phone_number_type.csv	csv	April 15, 2023, 16:30:06 (UTC-04:00)	228.0 B	Standard
phone_number.csv	csv	April 15, 2023, 16:30:06 (UTC-04:00)	179.5 KB	Standard
role_profile_type.csv	csv	April 15, 2023, 16:30:07 (UTC-04:00)	239.0 B	Standard
role_profile.csv	csv	April 15, 2023, 16:30:07 (UTC-04:00)	117.1 KB	Standard
user_profile.csv	csv	April 15, 2023, 16:30:08 (UTC-04:00)	119.3 KB	Standard
user.csv	csv	April 15, 2023, 16:30:08 (UTC-04:00)	115.1 KB	Standard

3. Storing Data to RDS: (I am using MySQL here since the data is structured) If I were to use AWS GLUE, I could have directly scheduled a spark job with it. But trading off for cost and prior experience, I chose EMR, hence this was a complex step. Considering the costs of running EMR for data loading given the free tier restrictions, I decided to store it in S3. However, here's one way you could load the data to RDS through EMR.
    a. Create an RDS instance using the AWS Management Console. Be sure to create it in the same region as the EMR
    b. For MySQL, you can download the JDBC driver and connect to use the 'write' dataframe API to store the data into RDS
    Ex:

```
df.write \
.format("jdbc") \
.option("url","jdbc:mysql://<my-rds-instance>:3306/my_database") \
.option("driver", "com.mysql.jdbc.Driver") \
.option("dbtable", "my_table") \
.option("user", "my_user") \
 .option("password", "my_password") \
.mode("write") \
.save()
```

Alternative Approach: The easiest alternative is to use AWS GLUE which is a self-managed cluster to avoid configuring the cluster. However, an interesting thing to explore would be the use of Apache Kafka if we were to process a stream of data in real time. Here are the steps that briefly describe how Kafka could be used in this case:

1. Install Kafka and configure the Kafka broker, producer, and consumer properties

2. Create a Kafka topic to store the input CSV data using Kafka and specify the number of partitions and replication factor.
3. Write a Kafka producer application to read the data from S3 and produce it to the Kafka topic
4. Create a Kafka Streams application to transform the input CSV data into the desired output schema. The stream processing logic, such as filtering, aggregating, etc. can be specified using DSL
5. Consume transformed data from the Kafka topic through a consumer application that reads the transformed data from the Kafka topic and writes it to S3/RDS