**Project Approach**

**Dataset creation**

Basically there is a very big file and from this file I have created the required datasets i.e., customers\_data, loans\_data, loan\_repayments and loan\_defaulters.

**1. Customers\_data**

The customers\_data is having information about the borrowers.

Selecting the columns member\_id, emp\_title, emp\_length, home\_ownership, annual\_inc, addr\_state, zip\_code, country, grade, sub\_grade, verification\_status, tot\_hi\_cred\_lim, application\_type, annual\_inc\_joint, verification\_status\_joint of the raw file to create customers\_data.

**Steps for dataset creation**

**Step1-** Using “withColumn”, create a new column “emp\_id”.

As the dataset is having a member\_id column which is having null values, thus we have to create a unique emp\_id column.

Using the “sha2” function to create the hash values for each customer using the information in 9 existing columns of the dataset and creating an emp\_id column. And this hash value will repeat only when the values for all 9 columns are the same.

Finally, concatenated 9 columns those column are as follows: 'emp\_title' ,'emp\_length', 'home\_ownership','annual\_inc', 'zip\_code', 'addr\_state', 'grade', 'sub\_grade', 'verification\_status' .

**Step2 -** Using spark sql query to select the information related to customers and using DataFrame writer API to write the output to the required folder. Using repartition(1) to save all the data in a single file.

**2. Loans\_data**

Loans\_data is having details about the loans given by the institution.

Selecting the columns Loan\_id, member\_id, loan\_amnt, funded\_amnt, term, int\_rate, installment, issue\_d, Loan\_status, purpose, title of the raw file to create Loans\_data.

**Steps for dataset creation**

Using the spark sql query to select the information related to loans and using DataFrame writer API to write the output to the required folder. Using repartition(1) to save all the data in a single file.

**3. Loan\_repayments**

Loan\_repayments is having details about the loans repayment history.

Selecting the columns loan\_id, total\_rec\_prncp, total\_rec\_int, total\_rec\_late\_fee, total\_pymnt, last\_pymnt\_amnt, last\_pymnt\_d, next\_pymnt\_d of the raw file to create Loans\_data.

**Step for dataset creation**

Using spark sql query to select the information related to loans and

using DataFrame writer API to write the output to the required folder. Using

repartition to save all the data in a single file.

**4. Loan\_defaulters**

Loan\_defaulters is the dataset having data about all the defaulters.

Selecting the columns member\_id(defaulter), delinq\_2yrs, delinq\_amnt, pub\_rec, pub\_rec\_bankruptcies, inq\_last\_6mths, total\_rec\_late\_fee, mths\_since\_last\_delinq, mths\_since\_last\_record of the raw file to create Loans\_data.

**Step for dataset creation**

Using spark sql query to select the information related to loans and using DataFrame writer API we have written the output to the required folder. Using repartition to save all the data in a single file.

**Data Cleaning**

**Customers\_data**

During the cleaning process of "customers\_data" modifications were applied to the data initially stored in the raw folder. After processing, the cleaned data was saved in the cleaned folder.

1.To give the correct schema we have explicitly mentioned the schema while creating the customer\_raw dataframe. The defined schema is as below

customer\_schema = 'member\_id string, emp\_title string, emp\_length string, home\_ownership string, annual\_inc float, addr\_state string, zip\_code string, country string, grade string, sub\_grade string, verification\_status string, tot\_hi\_cred\_lim float, application\_type string, annual\_inc\_joint float, verification\_status\_joint string'

2.To ensure accurate column names, we utilized the "withColumnRenamed" function. The column names were modified as follows:

“.withColumnRenamed("annual\_inc", "annual\_income") \

.withColumnRenamed("addr\_state", "address\_state") \

.withColumnRenamed("zip\_code", "address\_zipcode") \

.withColumnRenamed("country", "address\_country") \

.withColumnRenamed("tot\_hi\_cred\_lim", "total\_high\_credit\_limit") \

.withColumnRenamed("annual\_inc\_joint", "join\_annual\_income") “

3.To mention the time when we were processing the data we have created the “ingest\_date” column and used the “current\_timestamp()” function the query used is given below.

”.withColumn("ingest\_date", current\_timestamp())”

4.To get rid of duplicate records using below spark sql query unique records are selected and saved to new dataframe.

customers\_distinct = customers\_df\_ingestd.distinct()

“spark.sql("select \* from customers\_distinct")”

5.As the “emp\_length” column i.e. year of experience is in string format and follows a pattern such as "10 years.". So the non digits values(i.e.(\D)) are replaced using “regexp\_replace” function and replaced with blank space(“”) “.withColumn("emp\_length", regexp\_replace(col("emp\_length"), "(\D)", ""))”

6.To convert “emp\_length” column which was in string format to integer format it was casted into integer as below.

“.withColumn("emp\_length", customer\_income\_cleaned.emp\_length.cast("int"))”

7.To replace the nulls in the “emp\_length” column with the average employment length. First “avg\_emp\_duration” was calculated and then it was replaced in the “emp\_length” column using the below query “.na.fill(avg\_emp\_duration, subset=["emp\_length"])”

8.To clean the “address\_state” column, all values which were having more than 2 characters were replaced with the “NA” and other values remained unchanged, as shown in the following query.. “.withColumn("address\_state", when(length(col("address\_state")) > 2, "NA").otherwise(col("address\_state")))”

9.This cleaned data is stored to the output folder using the DataFrame writer API in csv and parquet format.

**Loans\_data**

During the cleaning process of "loans\_data" modifications were applied to the data initially stored in the raw folder. After processing, the cleaned data was saved in the cleaned folder.

1.To give the correct schema we have explicitly mentioned the schema while creating the customer\_raw dataframe. In a single step only we have changed the column names and mentioned the right datatypes.

The defined schema is as below.

loan\_schema = "loan\_id string, member\_id string, loan\_amount float, funded\_amnt float, loan\_term\_months string, interest\_rate float, monthly\_installment float, issue\_date string, loan\_status string, loan\_purpose string, loan\_title string"

2. To mention the time when we were processing the data we have created the “ingest\_date” column and used the current\_timestamp() function. ”.withColumn("ingest\_date", current\_timestamp())”

3. The records are dropped if in any one of the columns in the below check column has the null value

columns\_to\_check = ["loan\_amount", "funded\_amnt", "loan\_term\_months", "interest\_rate", "monthly\_installment", "issue\_date", "loan\_status", "loan\_purpose"]

Using below query:

loan\_df\_injested.na.drop(subset = columns\_to\_check)

4. Using “regexp\_replace” function “ months” in the loan\_term\_months column was replaced with the blank space(””). Then it was casted to float using the

.cast() function. After this to convert this month into year this value was divided by 12 and at the end casted into integer using the below query.

withColumn("loan\_term\_months", ((regexp\_replace("loan\_term\_months", " months", "")).cast("float")/12).cast("int")) \

.withColumnRenamed("loan\_term\_months", "loan\_term\_years")

5. To rename the unnecessary “loan\_purpose” with others “when clause” was used. For this the purposes beyond the defined list(loan\_purpose\_lookup) were renamed to “others” using the below query.

loan\_purpose\_lookup = ["debt\_consolidation", "credit\_card", "home\_improvement", "other", "major\_purchase", "medical", "small\_business", "car", "vacation",

"moving", "house", "wedding", "renewable\_energy",

"educational"]

.withColumn("loan\_purpose", when(col("loan\_purpose").isin(loan\_purpose\_lookup), col("loan\_purpose")).otherwise("other"))

6. This cleaned data is stored to the output folder using the DataFrame writer API in csv and parquet format.

**Loans\_repayments**

During the cleaning process of "Loan\_repayments," modifications were applied to the data initially stored in the raw folder. After processing, the cleaned data was saved in the cleaned folder.

1. We ensured the accurate schema by explicitly specifying it while creating the customer\_raw dataframe. In one step, we both renamed the columns and assigned the correct data types.

The defined schema is as below.

loan\_repay\_schema = "loan\_id string, total\_principle\_received float, total\_interest\_received float, total\_late\_fee\_received float, total\_payment\_received float, last\_payment\_amount float, last\_payment\_date string, next\_payment\_date string"

2. To mention the time when we were processing the data we have created the “ingest\_date” column and used the current\_timestamp() function. ”.withColumn("ingest\_date", current\_timestamp())”

3. Records are discarded if any of the columns listed below contain null values.

columns\_to\_check = ["total\_principle\_received", "total\_interest\_received", "total\_late\_fee\_received", "total\_payment\_received", "last\_payment\_amount"] Using below query:

df.na.drop(subset = columns\_to\_check)

4. If the total\_payment\_received is 0.0 but total\_principle\_received is not 0.0, indicating that the principle has been paid, the situation arises where the total payment received is inexplicably zero. In such cases, we replace total\_payment\_received with the sum of total\_principle\_received, total\_interest\_received, and total\_late\_fee\_received using a "when" clause.

Using the below query this is achieved.

.withColumn("total\_payment\_received", when( (col("total\_payment\_received") == 0.0) & (col("total\_principle\_received") !=0.0),

col("total\_principle\_received")+col("total\_interest\_received")+col("total\_late\_fe e\_received"))

.otherwise(col("total\_payment\_received")))

5. To remove the columns where total\_payment\_received is zero, we selected all the records where total\_payment\_received is not zero. Below filter condition is used to filter the records and then saved into another dataframe.

..filter("total\_payment\_received != 0.0")

6. Since last\_payment\_date cannot be 0, it must either contain a valid date or a null value. To replace null values, a "when clause" is employed, utilizing the following query.

.withColumn("last\_payment\_date", when( (col("last\_payment\_date") == 0.0),

None).otherwise(col("last\_payment\_date"))

7. Similarly as next\_payment\_date cannot be 0, either it should be some date or null value. Thus to replace null values ‘when clause’ is used and the query used is as below.

.withColumn("next\_payment\_date", when( (col("next\_payment\_date") == 0.0),

None).otherwise(col("next\_payment\_date"))

8. This cleaned data is stored to the output folder using the DataFrame writer API in csv and parquet format.

**Loan\_defaulters**

While cleaning of Loan\_defaulters following changes have been made on the data which was kept in the raw folder and then after processing saved in the cleaned folder.

1. To give the correct schema we have explicitly mentioned the schema while creating the customer\_raw dataframe to mention the right datatypes.

The defined schema is as below.

defaulter\_schema = "member\_id string, delinq\_2yrs float, delinq\_amnt float, pub\_rec float, pub\_rec\_bankruptcies float, inq\_last\_6mths float, total\_rec\_late\_fee float, mths\_since\_last\_delinq float, mths\_since\_last\_record float"

In the above delinq\_2yrs column which was in string initially was converted into the float datatype so any non-float values were converted into nulls.

2. Using below query delinq\_2yrs which was earlier casted to float was converted into integer as time period(i.e.years) should be an integer. And all the nulls were replaced by zero as the time period(i.e.years) cannot be null either it should be some integer or 0. The query used is as follows.

.withColumn("delinq\_2yrs", col("delinq\_2yrs").cast("integer")).fillna(0, subset = ["delinq\_2yrs"])

3.The data was stored into 2 separate files

a.The first dataset contains details of customers who missed or delayed payments, based on the conditions delinq\_2yrs > 0 or mths\_since\_last\_delinq

> 0. This dataset focuses on delinquency and includes the following columns: member\_id, delinq\_2yrs, delinq\_amnt, int(mths\_since\_last\_delinq). The records were filtered using the query:

spark.sql("select member\_id, delinq\_2yrs, delinq\_amnt, int(mths\_since\_last\_delinq) from loan\_defaulter where delinq\_2yrs>0 or mths\_since\_last\_delinq>0")

b.In the second dataset information about only those member\_id (borrowers) is considered for whom either there is public record(pub\_rec) or bankruptcy record (pub\_rec\_bankruptcies) or enquiries in the last 6 months (inq\_last\_6mths). This dataset is basically related to public records and the enquiries.

4.Both these cleaned dataset are stored to the output folder using the DataFrame writer API in csv and parquet format.

**Loan score calculation is dependent on 3 factors –**

1. Loan Payment History (Loan repayments history for previous loans if any)

2. Customer’s Financial Health

3. Loan Defaulters History (Delinq, Public records, Bankruptcies, Enquiries)

In order to calculate the loan score, two important tables are required

- Delinq

- A table with columns consisting of the details of Public records, Bankruptcies, Enquiries

**Cleaning the records and creating a Processed Dataframe**

loans\_def\_processed\_df = loans\_def\_raw\_df.withColumn(“delinq\_2yrs”, col(“delinq\_2yrs”).cast(“integer”).fillna(0, subset=[“delinq\_2yrs”])

loans\_def\_processed\_pub\_rec\_df = loans\_def\_raw\_df.withColumn(“pub\_rec”, col(“pub\_rec”).cast(“integer”).fillna(0, subset=[“pub\_rec”])

loans\_def\_processed\_pub\_rec\_bankruptcies\_df = loans\_def\_processed\_pub\_rec\_df.withColumn(“pub\_rec\_bankruptcies”, col(“pub\_rec\_bankruptcies”).cast(“integer”).fillna(0, subset=[“pub\_rec\_bankruptcies”])

loans\_def\_processed\_inq\_last\_6mths\_df = loans\_def\_processed\_pub\_rec\_bankruptcies\_df.withColumn(“inq\_last\_6mths”

, col(“inq\_last\_6mths”).cast(“integer”).fillna(0, subset=[“inq\_last\_6mths”])

**Creating a temporary table on the above processed data**

loans\_def\_processed\_inq\_last\_6mths\_df. createOrReplaceTempView(“loan\_defaulters”)

**Creating a detailed Dataframe including the above mentioned columns from the loan\_defaulters data used for the calculation of loan score**

loan\_defaulters\_detail\_records\_enq\_df = spark.sql(“select member\_id, pub\_rec, pub\_rec\_bankruptcies, inq\_last\_6mths from loan\_defaulters”)

**Writing back the Processed data to the Cleaned folder**

Writing back in CSV Format

loan\_defaulters\_detail\_records\_enq\_df.write \

.option(“header”, True) \

.format(“csv”) \

.mode(“overwrite”) \

.option(“path”, “/public/trendytech/lendingclubproject/cleaned/Loan\_defaulters\_detail\_records

\_enq\_csv”) \

.save()

**Writing back in PARQUET Format**

loan\_defaulters\_detail\_records\_enq\_df.write \

.format(“parquet”) \

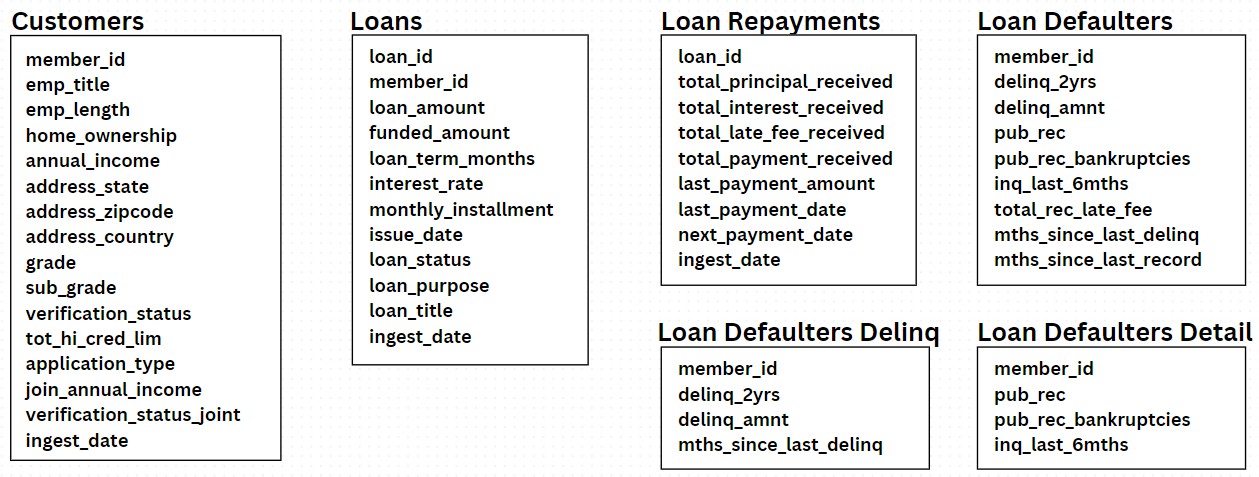
.mode(“overwrite”) \

.option(“path”, “/public/trendytech/lendingclubproject/cleaned/Loan\_defaulters\_detail\_records

\_enq\_csv”) \

.save()

**Final Cleaned and Processed Datasets for future processing**



**Permanent Table Creation on the Cleaned Data –**

Business Requirement 1 : Some of the teams are required to analyse the cleaned data which requires the creation of permanent tables on top of the cleaned data that allows the downstream teams to query the data using simple SQL like queries.

For the above use-case, since multiple teams are accessing the data, it is a best practice to create external tables as it doesn’t affect the actual data even if the table is dropped accidentally.

In the case of Managed Tables, Data is stored in the warehouse directory as mentioned in the configuration options while creating the spark session. Spark uses hive metastore to store the metadata in a persistent way.



**Reading the cleaned data and creating Dataframe for further processing**

**Reading the cleaned data and creating Dataframe for further processing**



**Steps for creating a Permanent Table**

1. **Create a Database with a proper naming convention to recognize the data easily.**



1. **Create an External Table by providing the fully qualified table name (database\_name.table\_name)**

A computer screen shot of a program

AI-generated content may be incorrect.

1. **To view the data**



**Data can be viewed in the Hive terminal as well. Command - describe formatted <table-name>**

(will give all the details of the table, like - owner of the table, type of the table, and so on)

**Business Requirement 2 :** The teams require a single consolidated view of all the datasets with the latest up-to-date data.

**Solution :** The best practice would be to create a view on the cleaned data that refreshes every 24 hrs. So the data that is part of the view will not be older than 24 hrs.

**Creating a View - create or replace view <view-name> as select-query**

A screenshot of a computer

AI-generated content may be incorrect.

Note: Creating a view would be much faster as there is no actual data processing taking place. However, a query to view the data, like the following -



This query will take time to execute as it involves joining multiple tables to generate a view with the desired data.

**Business Requirement 3:** Yet another team wants real quick access to the “view data” without having to wait for the view results to be processed. Since processing the results takes a very long time.

**Solution :** Precalculate the results by executing the join of tables prior. Ex - A weekly job performs the join of the underlying tables and stores the results in another table.

**Disadvantage :** Even though the results are pre-calculated and can be fetched much faster than the previous case, the data is not the latest

up-to-date data but rather a week old data. This approach can be taken if it is okay to consider a little older data for further processing.

**Note**: In this case, a Managed Table is created. The actual data for this table will be stored in the warehouse directory and the metadata is present in the Hive metastore.

# **Loan Score Calculation Criterias**

Higher the Loan Score, better the chances of getting the loan approval and vice-versa.

**3 major factors affecting the Loan Score :**

1. **Loan Repayment History**

This factor based on - last\_payment & total\_payment\_received

1. **Loan Defaulters History**

This factor is based on - delinq\_2years, public\_rec, public\_rec\_bankruptcies & inq\_last\_6mths

1. **Financial Health**

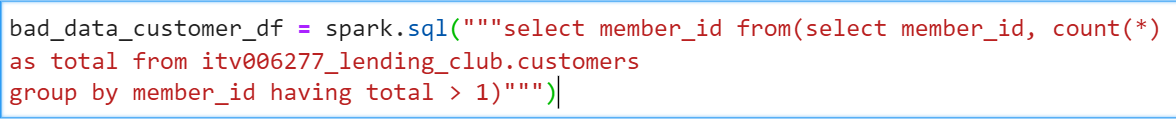
This factor is based on - home\_ownership, loan\_status, funded\_amount, grade\_points

**% contribution of each of the factors for the loan score calculation -**

Loan Repayment History - 20% Loan Defaulters History - 45% Financial Health - 35%

**Identifying the Bad Data and Final Cleaning (Repeating Member IDs)**

The repeating member ids would be bad data as there are multiple records for a single member-id (Ideally, there should be one record associated with a member-id).

* Query to identify the bad data :

Displays all the member-ids which have more than 1 records associated, implying the bad data.

* A computer screen with text

  AI-generated content may be incorrect.Create a consolidated CSV file which consists of the bad data from all the dataframes (Union of all the dataframes and choose the unique member-ids). This file can then be shared with the upstream team for correction of the bad data.
* Create a temporary view of the bad data file.
* Segregate the Good Data and create a final cleaned dataframe by excluding the member-ids present in the bad data temporary view. Store the final cleaned data to a new\_cleaned folder

A screenshot of a computer

AI-generated content may be incorrect.

A close-up of a logo

AI-generated content may be incorrect.

* A close up of text

  AI-generated content may be incorrect.Create External Tables over the final new cleaned data.

Note: Repeat the same for the other datasets

* + Create Spark Session
  + A blue background with text on it

    AI-generated content may be incorrect.A screen shot of a computer code

    AI-generated content may be incorrect.Need to configure user defined variables as shown in the diagram below (Code modifications are easier when the variable values are defined separately as compared to hardcode
  + Factors that contribute to the Loan score calculation

A computer code with red text

AI-generated content may be incorrect.**Loan Repayment History - 20%**

A computer screen shot of a computer code

AI-generated content may be incorrect.**Loan Defaulters History - 45%**

A screenshot of a computer

AI-generated content may be incorrect.**Financial Health - 35%**

**Final Loan Score Calculation :**

Final Loan score results will be stored under the processed folder in HDFS.

A screenshot of a computer

AI-generated content may be incorrect.