THE PIGGYBANK

DATA ENGINEERING CASE

The Piggybank is an American banking start-up. So far, bank has not that many customers, however it expects rapid growth in upcoming fiscal year. Currently all customers data is stored in CSV (Comma Separated Values) files and data is processed in the MS Office – all monthly reporting such as risk monitoring DPD30+, DPD60+, DPD90+ and overall credit portfolio summary.

The IT management has decided to switch from CSV files and use RDBMS (Relational Database Management System) to satisfy current and upcoming business needs.

You, as data engineer were asked to try out open-source RDBMS solution (SQLite), migrate all CSV filed to it and recreate business reports that are running now in MS Office.

Index

I. Overview of used data and presentation of physical model

For the purpose of this business case, four data sets were provided and imported into a database.

a. Client – information about customers

```
CREATE TABLE CLIENT (
    CUSTOMER_ID    INTEGER             PRIMARY KEY AUTOINCREMENT,
    REPORTING_DATE DATE                NOT NULL,
    AGE            INTEGER (18, 100) NOT NULL,
    EDUCATION      VARCHAR (20)        NOT NULL,
    BUCKET         INTEGER             NOT NULL
);
```

In the CLIENT table, CUSTOMER_ID is used as a primary key since it allows to identify each row in the table with a unique value. An **index** is created for the column CUSTOMER_ID to return customer information for a defined customer ID.

b. Income – information about customer income

```
CREATE TABLE INCOME (
    INCOME_ID      INTEGER                   PRIMARY KEY AUTOINCREMENT,
    CUSTOMER_ID    INTEGER                   REFERENCES CLIENT (CUSTOMER_ID),
    REPORTING_DATE DATE                      NOT NULL,
    FIRST_JOB      CHAR                      NOT NULL,
    INCOME         INTEGER (1000, 30000) NOT NULL,
    BUCKET         INTEGER                   NOT NULL
);
```

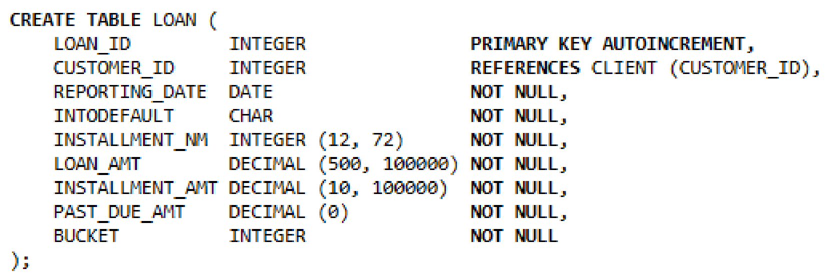
In the INCOME table, INCOME_ID is used as a primary key since it’s a unique number attributed to customer income. A foreign key is set for CUSTOMER_ID referring to CLIENT (CUSTOMER_ID) to link both tables together based on the customer ID. An **index** is created for the columns CUSTOMER_ID and INCOME_ID to return customer income information based either on the customer or income ID.

c. Household – information regarding customer family

```
CREATE TABLE HOUSEHOLD (
    HOUSEHOLD_ID   INTEGER          PRIMARY KEY AUTOINCREMENT,
    INCOME_ID      INTEGER          REFERENCES INCOME (INCOME_ID),
    REPORTING_DATE DATE             NOT NULL,
    MARRIED        CHAR             NOT NULL,
    HOUSE_OWNER    CHAR             NOT NULL,
    CHILD_NO       INTEGER (0, 10) NOT NULL,
    HH_MEMBERS     INTEGER (1, 10) NOT NULL,
    BUCKET         INTEGER          NOT NULL
);
```

In the HOUSEHOLD table, HOUSEHOLD_ID is used as a primary key since the column represents a unique number that identifies the household. A foreign key is set for INCOME_ID referring to INCOME (INCOME_ID) to link both tables together based on the income ID. An **index** is created for the columns HOUSEHOLD_ID and INCOME_ID to return information regarding customer family based either on the household or income ID.

d. Loan – information about loan status and characteristics



```
CREATE TABLE LOAN (
    LOAN_ID         INTEGER                PRIMARY KEY AUTOINCREMENT,
    CUSTOMER_ID     INTEGER                REFERENCES CLIENT (CUSTOMER_ID),
    REPORTING_DATE  DATE                   NOT NULL,
    INTODEFAULT     CHAR                   NOT NULL,
    INSTALLMENT_NM  INTEGER (12, 72)       NOT NULL,
    LOAN_AMT        DECIMAL (500, 100000)  NOT NULL,
    INSTALLMENT_AMT DECIMAL (10, 100000)   NOT NULL,
    PAST_DUE_AMT    DECIMAL (0)            NOT NULL,
    BUCKET          INTEGER                NOT NULL
);
```

In the LOAN table, LOAN_ID is used as a primary key since the column represents a unique number that identifies the loan. A foreign key is set for CUSTOMER_ID referring to CLIENT (CUSTOMER_ID) to link both tables together based on a customer's ID. An **index** is created for the columns LOAN_ID and CUSTOMER_ID to return information about loan characteristics based either on the loan or customer's ID.

II. Data quality checking

In order to make accurate informed business decision, good quality of data is important. Poor data can lead to biased analysis and wrong recommendation. In this case study we will check data quality through 3 main metrics: **completeness**, **consistency**, and **accuracy**.

- Data completeness metric measures whether all the necessary data is present in the dataset. To do so, we can check the presence of null values in our tables.
- Data validity metric measures how well data conforms to required value attributes. To do so, we can check if the primary keys used are unique as they should.
- Data accuracy metric measures how accurately the data reflects reality. To do so, we can check if there are any outliers in the datasets. Inconsistencies in the data could occur during data entry and can lead to wrong results if not taken care of.

a. CLIENT table

Checking the presence of null values in the table:

```
CREATE VIEW [client null assert] AS
    SELECT *
      FROM client
     WHERE CUSTOMER_ID IS NULL OR
           REPORTING_DATE IS NULL OR
           AGE IS NULL OR
           EDUCATION IS NULL OR
           BUCKET IS NULL;
```

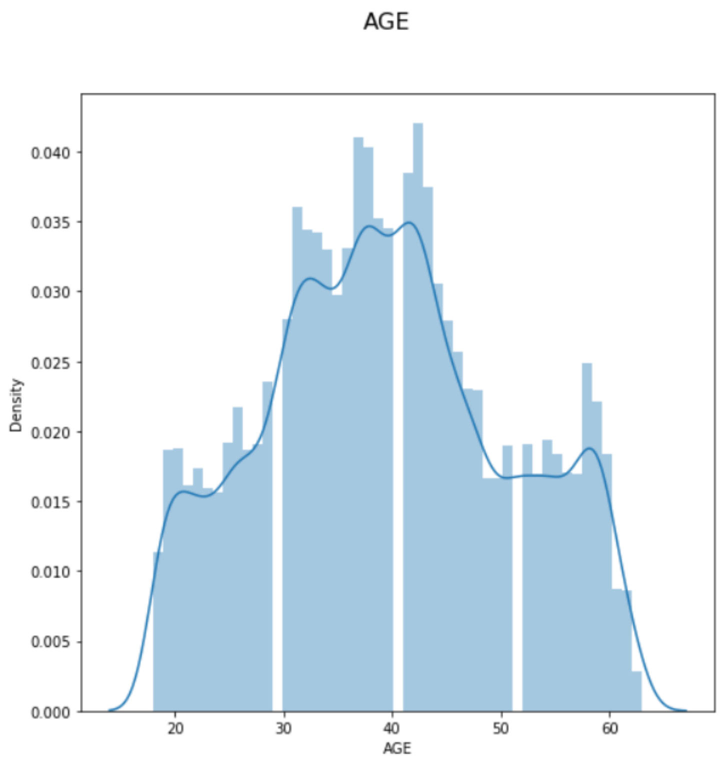
The query does not return any row meaning there is no null value in the table.

Verifying if the primary key (CUSTOMER_ID field) is unique:

```
CREATE VIEW [client unique assert] AS
    SELECT CUSTOMER_ID,
           SUM(1) AS count
      FROM CLIENT
     GROUP BY 1
    HAVING count > 1;
```

The query does not return any row meaning there is no duplicate CUSTOMER_ID values.

Trying to spot outliers in "AGE" column using plots:



The visual does not show any data inconsistency in the "AGE" column.

We then iterate to the other tables.

b. HOUSEHOLD table

Checking the presence of null values in the table:

```
CREATE VIEW [household null assert] AS
    SELECT *
      FROM household
     WHERE HOUSEHOLD_ID IS NULL OR
           INCOME_ID IS NULL OR
           REPORTING_DATE IS NULL OR
           MARRIED IS NULL OR
           HOUSE_OWNER IS NULL OR
           CHILD_NO IS NULL OR
           HH_MEMBERS IS NULL OR
           BUCKET IS NULL;
```

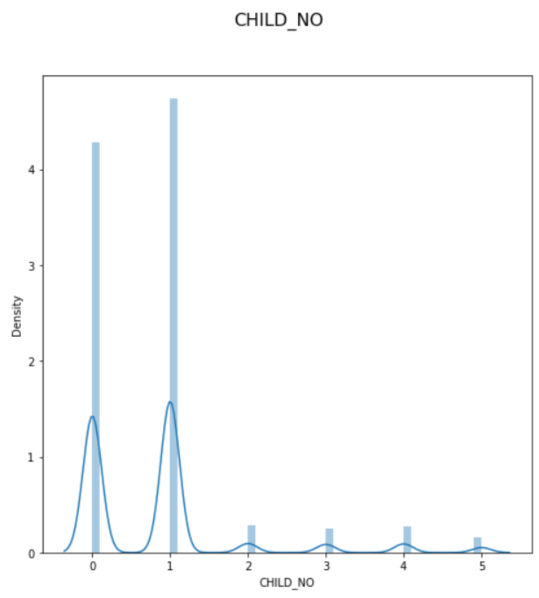
The query does not return any row meaning there is no null value in the table.

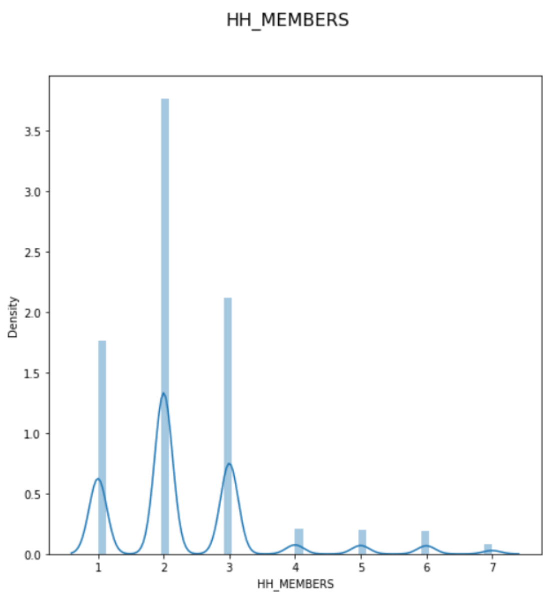
Verifying if the primary key (HOUSEHOLD_ID field) is unique:

```
CREATE VIEW [household unique assert] AS
    SELECT HOUSEHOLD_ID,
           SUM(1) AS count
      FROM HOUSEHOLD
     GROUP BY 1
    HAVING count > 1;
```

The query does not return any row meaning there is no duplicate CUSTOMER_ID values.

Trying to spot outliers in “CHILD_NO” and “HH_MEMBERS” columns using plots:





The visual does not show any data inconsistency in “CHILD_NO” and “HH_MEMBERS” columns.

c. INCOME table

Checking the presence of null values in the table:

```
CREATE VIEW [income null assert] AS
    SELECT *
      FROM income
     WHERE INCOME_ID IS NULL OR
           CUSTOMER_ID IS NULL OR
           REPORTING_DATE IS NULL OR
           FIRST_JOB IS NULL OR
           INCOME IS NULL OR
           BUCKET IS NULL;
```

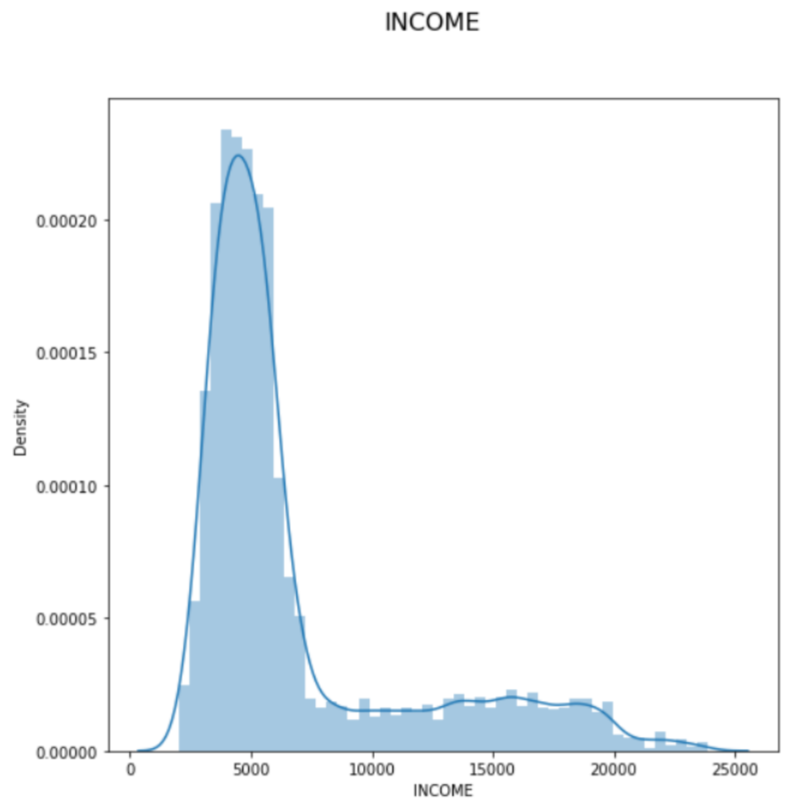
The query does not return any row meaning there is no null value in the table.

Verifying if the primary key (INCOME_ID field) is unique:

```
CREATE VIEW [income unique assert] AS
    SELECT INCOME_ID,
           SUM(1) AS count
      FROM INCOME
     GROUP BY 1
    HAVING count > 1;
```

The query does not return any row meaning there is no duplicate INCOME _ID values.

Trying to spot outliers in the "INCOME" column using plots:



The visual does not show any data inconsistency in the "INCOME _NO" column.

d. LOAN table

Checking the presence of null values in the table:

```
CREATE VIEW [loan null assert] AS
    SELECT *
      FROM loan
     WHERE LOAN_ID IS NULL OR
           CUSTOMER_ID IS NULL OR
           REPORTING_DATE IS NULL OR
           INTODEFAULT IS NULL OR
           INSTALLMENT_NM IS NULL OR
           LOAN_AMT IS NULL OR
           INSTALLMENT_AMT IS NULL OR
           PAST_DUE_AMT IS NULL OR
           BUCKET IS NULL;
```

The query does not return any row meaning there is no null value in the table.

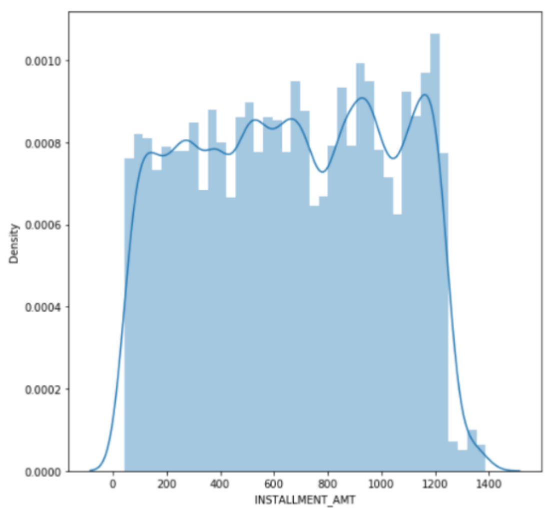
Verifying if the primary key (LOAN_ID field) is unique:

```
CREATE VIEW [loan unique assert] AS
    SELECT LOAN_ID,
           SUM(1) AS count
      FROM LOAN
     GROUP BY 1
    HAVING count > 1;
```

The query does not return any row meaning there is no duplicate LOAN _ID values.

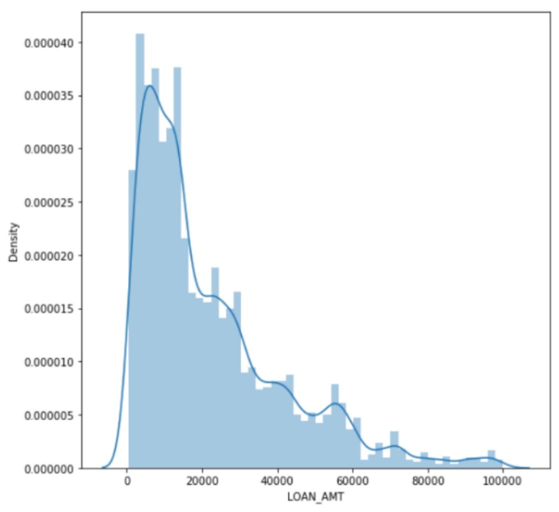
Trying to spot outliers in “INSTALLMENT_AMT” and “LOAN_AMT” columns using plots:

INSTALLMENT_AMT



PAST_DUE_AMT

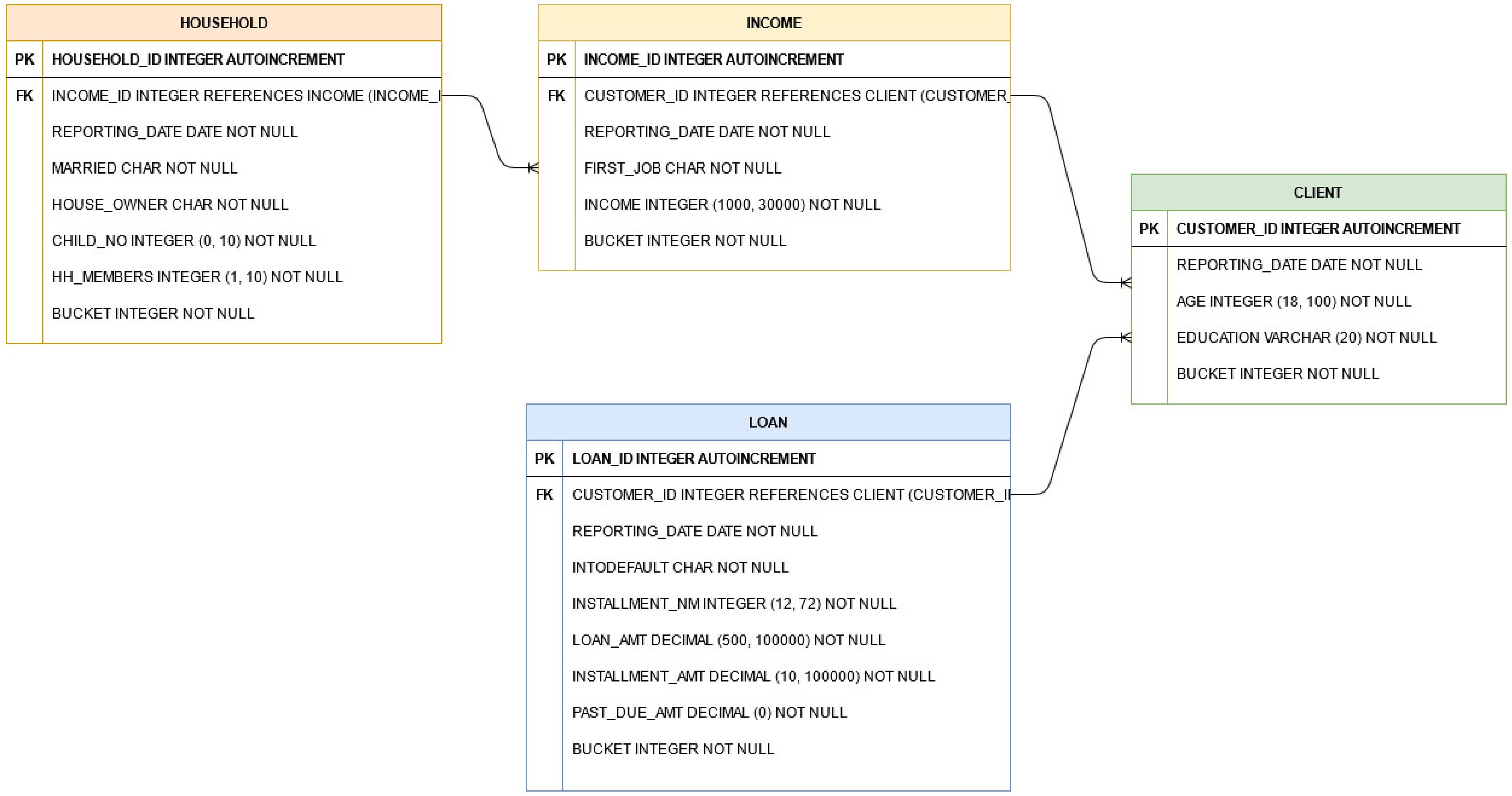
LOAN_AMT



The visual does not show any data inconsistency in “INSTALLMENT_AMT” and “LOAN_AMT” columns.

The provided datasets do not require any cleaning since the data is complete, consistent, and accurate.

III. Entity relation diagram (ERD) for source tables


HOUSEHOLD
PK HOUSEHOLD_ID INTEGER AUTOINCREMENT
FK INCOME_ID INTEGER REFERENCES INCOME (INCOME_I
REPORTING_DATE DATE NOT NULL
MARRIED CHAR NOT NULL
HOUSE_OWNER CHAR NOT NULL
CHILD_NO INTEGER (0, 10) NOT NULL
HH_MEMBERS INTEGER (1, 10) NOT NULL
BUCKET INTEGER NOT NULL
INCOME
PK INCOME_ID INTEGER AUTOINCREMENT
FK CUSTOMER_ID INTEGER REFERENCES CLIENT (CUSTOMER_
REPORTING_DATE DATE NOT NULL
FIRST_JOB CHAR NOT NULL
INCOME INTEGER (1000, 30000) NOT NULL
BUCKET INTEGER NOT NULL
CLIENT
PK CUSTOMER_ID INTEGER AUTOINCREMENT
REPORTING_DATE DATE NOT NULL
AGE INTEGER (18, 100) NOT NULL
EDUCATION VARCHAR (20) NOT NULL
BUCKET INTEGER NOT NULL
LOAN
PK LOAN_ID INTEGER AUTOINCREMENT
FK CUSTOMER_ID INTEGER REFERENCES CLIENT (CUSTOMER_I
REPORTING_DATE DATE NOT NULL
INTODEFAULT CHAR NOT NULL
INSTALLMENT_NM INTEGER (12, 72) NOT NULL
LOAN_AMT DECIMAL (500, 100000) NOT NULL
INSTALLMENT_AMT DECIMAL (10, 100000) NOT NULL
PAST_DUE_AMT DECIMAL (0) NOT NULL
BUCKET INTEGER NOT NULL

IV. Data Pipeline

The client expects an increasing amount of data for upcoming fiscal year. So, the pipeline must be scalable, ensure data quality and be able to collect and process data from different sources by matching and merging the data into a single record, the CUSTOMER_CAR table (destination).

For a start-up in banking industry, we opt for a **Data Quality Pipeline** as it contains functions such as standardization of all new customer names at regular intervals and validation of customer's information like addresses that are useful for the loan application approval.

The data comes from the client's platform for loan application. As the clients are awaiting approval for their loan, and customer's data must be ready for analysis, the pipeline moves data from the source at regularly scheduled intervals of 20 minutes. The information collected is composed of data about the client, its household, its income dans its loan situation.

The Pipeline performs standardization, matching and merging of the data to ensure the information matches the physical model of CUSTOMER_CAR (format, schema).

In case of the need to handle data correction, it is possible to automate the correction in the "Transform" part of the ETL pipeline. For instance, if the date is entered in a different format, we can implement a replace tool. Same solution goes if a pattern is found in typos being present in the loaded data.

The clean and formatted data is then appended to CUSTOMER_CAR.

V. RDBMS recommendation

Since Piggybank is expecting a rapid growth of customer, the choice of RDBMS to store data is relevant. They are specifically designed to store and retrieve large amount of information, are consistent, stable, and reliable.

It is important to consider the business requirements before recommending a database system. The banking industry has its proper features and needs that must be taken into account. The ability to process daily volumes, to support high-rate transaction processing and data protection are obvious major concerns for a growing banking start-up.

In addition, Piggybank is looking for a RDBMS that satisfies other key requirements: the database must be open source, cost effective and offers flexibility, scalability, and reporting capabilities.

To fulfil those requirements, and based on the nature of the business, I would recommend two open-source RDBMS that are widely used in the banking sector: **MariaDB** and **PostgreSQL**.

PostgreSQL is a solution of choice for businesses of all sizes. By far the most mentioned advantage of PostgreSQL is the efficiency of its core algorithm, which means that it outperforms many databases supposed to be more advanced. This is especially useful if you are working with large datasets. It is also one of the most flexible open-source databases on the market. You can write functions in many languages: Python, Perl, Java, Ruby, C and R. Finally, as one of the most commonly used open-source databases, PostgreSQL's community support is one of the best on the market.

On the other hand, MariaDB is an open-source distribution of MySQL and was created after the acquisition of MySQL by Oracle. Many users choose MariaDB over MySQL because MariaDB has frequent security updates. While this does not necessarily mean that MariaDB is more secure, it does indicate that the development community takes security seriously, which represents an important point for a banking start-up. Other advantages of MariaDB are that it will almost certainly remain open source and highly compatible with MySQL. Because of this compatibility, MariaDB also works well with many other languages commonly used with MySQL.