ACCENTURE BUSINESS CASE

PIOTR SZEWCZUL

Conclusions after file analysis:

- Each file has unique value that identifies object/person
- Unique identifier from each file can't be set as PRIMARY KEY for desired table – it has to be combined with REPORTING_DATE field
- For REPORTING_DATE and BUCKET combination we can generate another table. If implemented this will reduce data redundancy and prevent problem with inconsistent date format.
- Each file except CLIENT has reference field from another table, that we can set as FOREIGN KEY with REPORTING_DATE to prevent e.g. combining household with client who is no longer repaying a loan.
- Based on required reports we can assume that data will most often be manipulated/filtered based on REPORTING_DATE. With this in mind we can set up our indexes (auto generated based on defined PK).
- Before adding data to selected table, we need to clean it up. For this purpose, I decided to create RAW Tables, which will hold all data in TEXT format. This approach can also help in future data updates – we use SQL code that only inserts records that do not exist already in target table (records in target table can be deleted to force data update).
- Error reports will we stored in database as views. Before data is processed to target table user can analyze view and correct the data if necessary (only rows with no errors are processed to target table).

1. Physical Design of Piggybank database.

Based on above points I created Database model showing in Fig.1

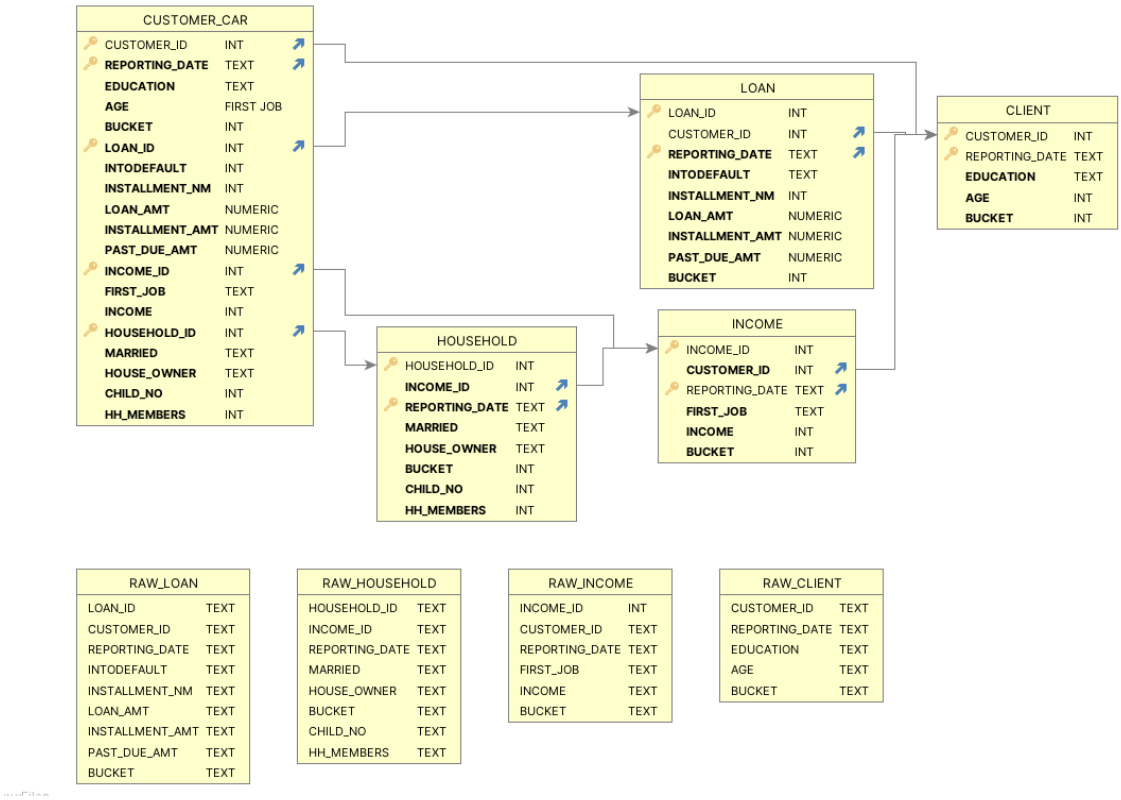

Figure 1 ERD schema piggybank database

Tables in details:

1. LOAN
    a. PRIMARY KEY: (LOAN_ID, REPORTING_DATE)
    b. FOREIGN_KEY: (CUSTOMER_ID, REPORTING_DATE) referencing CLIENT(CUSTOMER_ID, REPORTING_DATE)
    c. INDEXES: auto generated based on PK
2. CLIENT

a. PRIMARY KEY: (CUSTOMER_ID, REPORTING_DATE)
b. FOREIGN KEY: None
c. INDEXES: auto generated based on PK

3. INCOME
    a. PRIMARY KEY: (INCOME_ID, REPORTING_DATE)
    b. FOREIGN KEY: (CUSTOMER_ID, REPORTING_DATE) referencing CLIENT(CUSTOMER_ID, REPORTING_DATE)
    c. INDEXES: auto generated based on PK
4. HOUSEHOLD
    a. PRIMARY KEY: (HOUSEHOLD_ID, REPORTING_DATE)
    b. FOREIGN KEY: (INCOME_ID, REPORTING_DATE) referencing INCOME(INCOME_ID_ID, REPORTING_DATE)
    c. INDEXES: auto generated based on PK
5. CUSTOMER_CAR
    a. PRIMARY_KEY: (CUSTOMER_ID, LOAN_ID, HOUSEHOLD_ID, INCOME_ID, REPORTING_DATE)
    b. FOREIGN_KEYS:
        i. (CUSTOMER_ID, REPORTING_DATE) referencing CLIENT (CUSTOMER_ID, REPORTING_DATE)
        ii. (INCOME_ID, REPORTING_DATE) referencing INCOME (INCOME_ID_ID, REPORTING_DATE)
        iii. (LOAN_ID, REPORTING_DATE) referencing LOAN (LOAN_ID, REPORTING_DATE)
        iv. (HOUSEHOLD_ID, REPORTING_DATE) referencing HOUSEHOLD (HOUSEHOLD_ID, REPORTING_DATE)
    c. INDEXES: auto generated based on PK

Foreign Keys is CUSTOMER_CAR are set to prevent inserting incorrect data into table by hand or any other insert statement. IF data injection will be handled 100% by prepared scripts we can skip this step.

All tables also have defined CHECK CONSTRAINTS based on predefined structure (for details please relate to ddl_database.sql script) to prevent inserting incorrect data.

2. Data cleaning, quality analyses

Data quality analyses are applied to each table by a SQL script, which later is used to create error reporting view (see err_tables_creation.sql script for details). Sample code for RAW_INCOME table analysis below:

```
DROP VIEW IF EXISTS RAW_INCOME_CHECK;
CREATE VIEW RAW_INCOME_CHECK
AS
WITH PRIMARY_CHECK AS (
    SELECT INCOME_ID, REPORTING_DATE, COUNT(*) HOW_MANY
    FROM RAW_INCOME
    GROUP BY INCOME_ID, REPORTING_DATE
    HAVING COUNT(*) > 1
)
, FOREIGN_KEY_CHECK AS (
    SELECT A.CUSTOMER_ID CUST_MAIN ,A.REPORTING_DATE REPORT_MAIN
    , B.CUSTOMER_ID CUST_REF, B.REPORTING_DATE REPORT_REF
    FROM RAW_INCOME A
    LEFT JOIN CLIENT B
        ON A.CUSTOMER_ID = B.CUSTOMER_ID
        AND A.REPORTING_DATE = B.REPORTING_DATE
)

SELECT BB.INCOME_ID
, BB.CUSTOMER_ID
, BB.REPORTING_DATE
, BB.FIRST_JOB
, BB.INCOME
, BB.BUCKET
, CASE WHEN BB.NULL_CHECK != 'GOOD' THEN 1 ELSE 0 END NULL_CHECK
, CASE WHEN  BB.PRIMARY_KEY_CHECK != 'GOOD' THEN 1 ELSE 0 END PRIMARY_KEY_CHECK
, CASE WHEN BB.FOREIGN_KEY_CHECK != 'GOOD' THEN 1 ELSE 0 END FOREIGN_KEY_CHECK
, CASE WHEN  BB.DATE_CHECK != 'GOOD' THEN 1 ELSE 0 END DATE_CHECK
, CASE WHEN BB.PRIMARY_KEY_CHECK != 'GOOD'
    OR BB.FOREIGN_KEY_CHECK != 'GOOD'
    OR BB.INTEGER_POSITIVE_CHECK != 'GOOD'
    OR  BB.NULL_CHECK != 'GOOD'
    OR  BB.INCOME_CHECK != 'GOOD'
    OR  BB.FIRST_JOB_CHECK != 'GOOD'
    THEN 1 ELSE 0 END ERR_CHECK
, INTEGER_POSITIVE_CHECK || char(09) || NULL_CHECK || char(09) || INCOME_CHECK || char(09) || FIRST_JOB_CHECK || char(09) || PRIMARY_KEY_CHECK || char(09)
|| FOREIGN_KEY_CHECK || char(09) || DATE_CHECK ERR_MSG
FROM (
SELECT A.*
, CASE WHEN B.INCOME_ID IS NOT NULL THEN 'PRIMARY KEY ERROR' ELSE 'GOOD' END PRIMARY_KEY_CHECK
, CASE WHEN C.CUST_REF IS NULL THEN 'FOREIGN KEY ERROR' ELSE 'GOOD' END FOREIGN_KEY_CHECK
, CASE WHEN cast(cast(A.INCOME_ID AS INTEGER) AS TEXT) != A.INCOME_ID or CAST(A.INCOME_ID AS INT) <= 0 THEN 'NOT INTEGER OR BELOW 0 IN ROW'
    WHEN cast(cast(A.CUSTOMER_ID AS INTEGER) AS TEXT) != A.CUSTOMER_ID or CAST(A.CUSTOMER_ID AS INT) <= 0 THEN 'NOT INTEGER OR BELOW 0 IN ROW'
    WHEN cast(cast(A.BUCKET AS INTEGER) AS TEXT) != A.BUCKET or CAST(A.BUCKET AS INT) <= 0 THEN 'NOT INTEGER OR BELOW 0 IN ROW'
    ELSE 'GOOD' END INTEGER_POSITIVE_CHECK
, CASE WHEN A.INCOME_ID IS NULL OR A.REPORTING_DATE IS NULL OR A.CUSTOMER_ID IS NULL
    OR A.FIRST_JOB IS NULL OR A.INCOME IS NULL OR A.BUCKET IS NULL THEN 'NULL IN ROW'
    ELSE 'GOOD' END NULL_CHECK
, CASE WHEN STRFTIME('%d',A.REPORTING_DATE) IS NULL THEN 'NOT PROPER DATE FORMAT' ELSE 'GOOD' END DATE_CHECK
, CASE WHEN CAST(A.INCOME AS INT) NOT BETWEEN 1000 AND 30000 THEN 'INCOME NOT IN RANGE IN ROW' ELSE 'GOOD' END INCOME_CHECK
, CASE WHEN A.FIRST_JOB NOT IN ('Y','N') THEN 'UNSUPPORTED VALUE IN FIRST_JOB' ELSE 'GOOD'
    END FIRST_JOB_CHECK
FROM RAW_INCOME A
LEFT JOIN PRIMARY_CHECK B
    ON A.INCOME_ID = B.INCOME_ID
    AND A.REPORTING_DATE = B.REPORTING_DATE
LEFT JOIN FOREIGN_KEY_CHECK C
    ON A.CUSTOMER_ID = C.CUST_MAIN
    AND A.REPORTING_DATE = C.REPORT_MAIN
) BB;
```

Script for each table has default set of checks which is:

- NULL_CHECK: checks if IS NULL condition is met on any column in each row
- PRIMARY_KEY_CHECK: checks for duplicates on primary key field/fields
- FOREIGN_KEY_CHECK (if exists): checks if foreign key exists in specific table (not raw_table). Note: data for each raw_table are processed after check are performed.
- INTEGER_POSITIVE, DECIMAL_POSITIVE: checks are performed based on cast function which converts data which converts data from TEXT to INT/DECIMAL and again to text. Final result will differ from original one if converting can't be used correctly on a specific field.

Other checks used in the scripts verify the range of data / whether the data is in the specified set. Finally, we generate an error message and fields with 0/1 values for future metrics calculations.

Metrics used to define data quality are:

- NULL % - what percentage of rows contains nulls
- DUPLICATE % - what percentage of rows contains duplicate PK
- REFERENCE % – what percentage of rows contain nonexistent FK
- DATE ERR % - what percentage of rows contains date in wrong format
- ERR % - how many rows contains any errors

Script used to calculate above metrics is presented on Fig.2.

```
DROP VIEW IF EXISTS METRICS_RPT;
CREATE VIEW METRICS_RPT
AS
SELECT 'HOUSEHOLD' AS TAB_NAME, (SUM(CAST(NULL_CHECK AS INTEGER))/ COUNT(*))*100 '%_NULL'
, (SUM(CAST(PRIMARY_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_DUPLICATES'
, (SUM(CAST(FOREIGN_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_REFERENCES_ERR'
, (SUM(CAST(DATE_CHECK AS INTEGER))/ COUNT(*))*100 '%_DATE_ERR'
, (SUM(CAST(ERR_CHECK AS INTEGER))/ COUNT(*))*100 '%_ERR'
FROM RAW_HOUSEHOLD_CHECK
UNION ALL
SELECT 'CLIENT' AS TAB_NAME, (SUM(CAST(NULL_CHECK AS INTEGER))/ COUNT(*))*100 '%_NULL'
, (SUM(CAST(PRIMARY_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_DUPLICATES'
, NULL
, (SUM(CAST(DATE_CHECK AS INTEGER))/ COUNT(*))*100 '%_DATE_ERR'
, (SUM(CAST(ERR_CHECK AS INTEGER))/ COUNT(*))*100 '%_ERR'
FROM RAW_CLIENT_CHECK
UNION ALL
SELECT 'INCOME' AS TAB_NAME, (SUM(CAST(NULL_CHECK AS INTEGER))/ COUNT(*))*100 '%_NULL'
, (SUM(CAST(PRIMARY_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_DUPLICATES'
, (SUM(CAST(FOREIGN_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_REFERENCES_ERR'
, (SUM(CAST(DATE_CHECK AS INTEGER))/ COUNT(*))*100 '%_DATE_ERR'
, (SUM(CAST(ERR_CHECK AS INTEGER))/ COUNT(*))*100 '%_ERR'
FROM RAW_INCOME_CHECK
UNION ALL
SELECT 'LOAN' AS TAB_NAME, (SUM(CAST(NULL_CHECK AS INTEGER))/ COUNT(*))*100 '%_NULL'
, (SUM(CAST(PRIMARY_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_DUPLICATES'
, (SUM(CAST(FOREIGN_KEY_CHECK AS INTEGER))/ COUNT(*))*100 '%_REFERENCES_ERR'
, (SUM(CAST(DATE_CHECK AS INTEGER))/ COUNT(*))*100 '%_DATE_ERR'
, (SUM(CAST(ERR_CHECK AS INTEGER))/ COUNT(*))*100 '%_ERR'
FROM RAW_LOAN_CHECK
```

Figure 2 Script used to calculate metrics

3. Data pipeline process

Data injection to FACTS table (CUSTOMER_CAR) are performed in few steps and could be run in single script (for better readability specific operations are separated between different script files). Whole process is described on Fig.3.

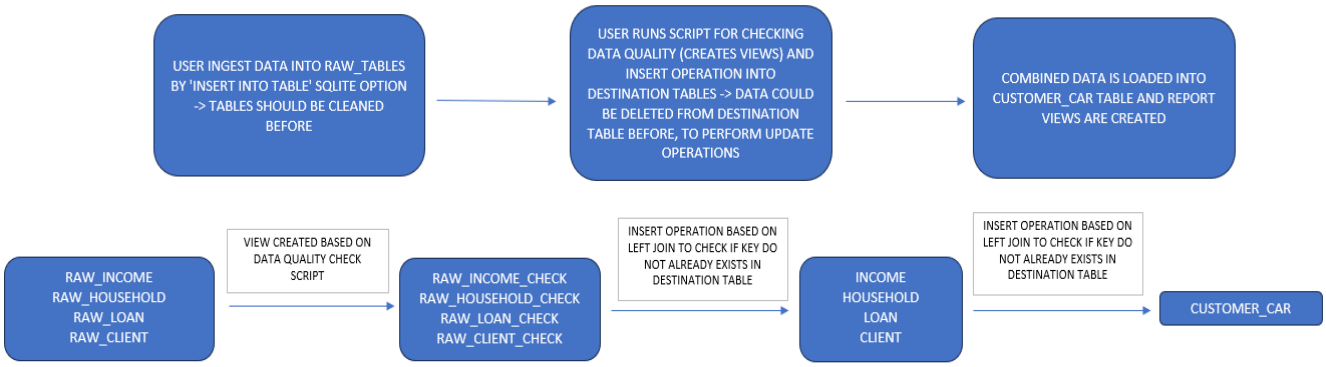

Figure 3 data pipeline piggybank database

Based on the files data it's safe to assume that data is collected on the last day of each month. We can perform above operations on the same day or a day later to keep the data updated.

Data Correction can be easily handled by inserting data into raw_table and performing updates statements in place of insert statement. Whole problem with insert/update statement would be solved if SQLite had MERGE statement.

4. Report

All reports definition is stored in reports.sql file.

5. Recommendations

The SQlite database has a limited set of features and capabilities compared to other RDBMS databases. Particularly troublesome is the support for updating/inserting data without a MERGE statement. Another problem with the SQLite database is the need to store a local copy of the database, which drastically limits the number of users who can perform operations on the database. This inconvenience is required to ensure data consistency. If the amount of data continues to grow, the process outlined in this document may not meet the customer's expectations and force a move to another RDBMS database.

All of the above issues aside, moving to an RDBMS is the right decision. Each table has an established set of attributes and rules. The bank can now easily perform data operations, create reports and analyze current/historical data.