Status Completed

Product requirements

Product manager Person
Engineer Person
UX designer Person
UX researcher Person

Introduction

Purpose

The goal of this development is to provide a feature that allows customers to access their transaction history through a mobile application, enabling them to review up to two years of transactions, while adhering to specific limitations on the number of records returned.

Problem statement

Customers require easy access to their transaction history. However, for performance reasons, transaction histories must be limited: if the total number of transactions exceeds 3500, the API should only return the latest 3500 transactions in descending order, while ensuring that all transactions are displayed if the count is fewer than 3500.

Target audience

Primary Users: Bank customers who wish to access their transaction history for personal financial management, auditing, or review purposes.
Pain Points: Difficulty accessing transaction history due to limitations in current banking applications.
Needs: A reliable, fast, and user-friendly way to query and display past transaction data for up to two years.

Objectives

Goals

Enable customers to retrieve a transaction history spanning up to two years.

Improve user experience by limiting the transaction data based on set thresholds (3500 transactions).

Optimize the API performance to handle millions of transactions efficiently.

Non-goals

The API will not include features for modifying or adding transactions to the database.

The API will not handle any transactions beyond a two-year history.

Metrics

Quantitative

Transaction Retrieval Time: Measure how quickly the API responds to a request, aiming for a response time of under 2 seconds.

System Load: Track server resource utilization and ensure the system can handle requests for millions of users, especially during peak traffic times.

Transaction Count Accuracy: Ensure that the total number of transactions returned is correct and adheres to the 3500 transaction limit when necessary.

Qualitative

1. **User Satisfaction:** Collect feedback through surveys or app reviews to gauge customer satisfaction with the transaction history feature.
2. **Error Rate:** Monitor how often users experience issues when querying transactions (e.g., data discrepancies or failure to load transaction history).

Development Details:

Technologies Used:

- **Backend Framework:** Python, FastAPI
- **Database:** MongoDB
- **Big Data Processing:** Apache Spark, Hadoop, PySpark

Architecture & Design:

1. **Database Selection & Architecture:** Initially, Hadoop Datalake was considered, but we chose to separate the load and set up a dedicated server to handle this application. This approach ensured that we could allocate dedicated resources for managing customer data and transaction history efficiently. The system handled **36 million customer accounts** and **16 billion+ transactions**, which required scalable and efficient data handling.

2. **ETL Development:**

    - **Accounts ETL:** For customer accounts, a **daily backload** strategy was implemented using Spark in Python. Given the nature of the account data, we opted to overwrite the data daily to ensure fresh and accurate information.

    - **Transactions ETL:** The transactions required an incremental ETL process. Initially, we performed a **one-time backload** of historical transactions using Spark with PySpark, reading data from Hadoop directories. The system then transitioned to an **incremental ETL** process to update the transaction history as new data came in.
3. **Reason for Backload Choice:**
   Although we had the option to use Hadoop's **murmur_hash** for stage-based data handling and incremental data processing, we opted for the backload method due to the complexity of the transaction data and the need for an efficient, easily manageable solution.

4. **Data Storage:** After processing the data with Spark, we converted it into **JSON format** and loaded it into **MongoDB**. This format was chosen for its flexibility and compatibility with FastAPI, which would allow for smooth API integration.

API Design:

Framework:
The API was built using **FastAPI**, a modern, fast web framework for Python. This was chosen for its performance, asynchronous support, and ease of integration with the backend database (MongoDB).

API Endpoint:

- **Request Parameters:**
    - `from_date`: The start date for the transaction history query.
    - `to_date`: The end date for the transaction history query (maximum 2 years difference from the `from_date`).
- **Logic:**
    - If the total number of transactions between the given date range is **less than or equal to 3500**, the API returns **all transactions**.
    - If the total number of transactions exceeds **3500**, the API limits the result to **3500 transactions**, sorted by **descending order** (most recent first).

API Workflow:

1. The user sends a request with a specified date range (from_date to to_date).

2. The system calculates the number of transactions in the specified range.
3. If the total transaction count is ≤ 3500, all transactions are returned.
4. If the total transaction count exceeds 3500, the API returns only the most recent 3500 transactions in descending order.

Summary of Key Choices and Benefits:

- **MongoDB** was used as the database for its flexibility in handling large, semi-structured data, which allowed for efficient storage and retrieval of transaction records.
- The **Spark-based ETL** solution ensured efficient processing of vast amounts of data (16B+ transactions), with the ability to scale for future growth.
- **FastAPI** provided a fast, asynchronous interface for querying the transaction data, optimizing performance for mobile banking app users.
- By opting for a **backload strategy** for account data and an **incremental ETL** process for transactions, we ensured that the system remained efficient, scalable, and maintainable over time.

This solution met the requirement to deliver accurate transaction histories within the constraints of the bank's data security policies, while also providing a scalable and efficient architecture to handle large volumes of data and transactions.