Analyzing Transactional Data for Fraud Detection and Anomaly Identification

Yahya Abdulmohsin for Washginton State University

Nov 23rd, 2024

Version: 1.1
Last updated: 2024-12-04

1 My Data

General Information

The dataset used in this study consists of 2,512 samples of transaction data, focusing on various financial activities and customer behaviors. It was collected for the purpose of exploring fraud detection and anomaly identification in financial transactions. Each entry in the dataset captures detailed transaction information, such as transaction amounts, timestamps, account balances, customer demographics, and other related attributes.

Key fields

- TransactionID: A unique alphanumeric identifier for each transaction.
- AccountID: Identifier for each account, allowing for the tracking of multiple transactions per account.
- TransactionAmount: The value of each transaction, varying from small to large transactions.
- TransactionDate: Timestamp of when the transaction occurred.
- TransactionType: Categorical field indicating whether the transaction was a 'Credit' or 'Debit'.
- Location: Geographic location of the transaction, based on U.S. cities.
- DeviceID: Identifier for the devices used during the transaction.
- IP Address: The IP address from which the transaction was initiated.
- MerchantID: Identifier for the merchant associated with the transaction.
- AccountBalance: The account balance after the transaction.
- PreviousTransactionDate: Timestamp of the last transaction, useful for analyzing transaction frequency.
- Channel: Indicates whether the transaction was performed online, at an ATM, or at a branch.
- CustomerAge: Age of the account holder.
- CustomerOccupation: The occupation of the account holder, which may correlate with income patterns.
- TransactionDuration: Time in seconds that the transaction took.
- LoginAttempts: The number of login attempts made before the transaction.

Why This Dataset

This dataset is well-suited for machine learning tasks focused on financial fraud detection, anomaly detection, and behavior analysis. The variety of features makes it ideal for developing predictive models that can help improve financial security and identify irregular transactional behaviors.

2 Data Structures

Chosen Data Structures

- Linked List
  - A linked list was chosen to store the transaction records in a sequential manner. Each transaction entry is stored as a node in the linked list, where each node contains details such as TransactionID, TransactionAmount, TransactionDate, and other relevant attributes.
- Hash Map
  - A hash map is used to store and retrieve data based on unique keys such as AccountID, TransactionID, and MerchantID. This allows for efficient lookups of data related to a specific account or transaction, making it easier to perform operations like searching for specific transactions or accounts.

Justifications for Choosing the Data Structures

- Linked List
  - Why: A linked list is ideal for representing sequential data, where the number of transactions can vary and may not be known in advance. This allows for dynamic memory allocation and efficient insertions and deletions of transaction records. For example, when analyzing transactions in real-time or processing new transaction entries, the linked list offers flexibility without requiring reallocation or resizing of memory.
  - Efficiency: The linked list structure is optimal when frequent insertions and deletions of data elements are required. This is particularly useful when processing transactions sequentially or when handling large datasets where the number of transactions is not fixed. Moreover, the linked list provides constant-time complexity for insertions at the head or tail of the list.
- Hash Map
  - Why: The hash map (or dictionary in Python) is an essential data structure for mapping unique identifiers, such as AccountID and TransactionID, to their corresponding data. This is crucial when we need to quickly access the transactions associated with a specific account or retrieve transaction details based on a unique identifier.

  - Efficiency: Hash maps provide constant-time average complexity for lookups, insertions, and deletions, making them highly efficient for operations that require fast access to data based on a key. In the context of this dataset, this ensures that any transaction or account can be retrieved in minimal time, even with a large dataset.

3 Question and Preparations

Chosen Question

The central question in this analysis is: Can we identify potential fraudulent transactions based on unusual patterns or anomalies? To address this question, we focus on two specific indicators that may suggest fraudulent activity:

- A daily transaction total of 2000 US Dollars or more per user, which could signify abnormal financial behavior.
- Four or more login attempts before a transaction, potentially indicating account compromise or unauthorized access.

By investigating these patterns, we aim to identify transactions that deviate from normal user behavior, offering insights into possible fraud.

Cleaning the Data

- Handle Missing Values
  - Check for Missing Values: Inspect the dataset for any missing or null values in important fields (such as TransactionAmount, AccountID, etc.).
  - Fill Missing Values: If any missing values are found, appropriate methods will be applied to fill them (e.g., using mean or median values for numerical fields, or mode for categorical fields).
  - Drop Rows/Columns if Necessary: If certain rows or columns contain too many missing values and cannot be reliably filled, they will be removed from the dataset to maintain data integrity.
- Validate Data Types
  - Ensure Numeric Fields are Correctly Typed: Confirm that all numeric fields (TransactionAmount, AccountBalance, etc.) are stored as numerical data types.
  - Ensure Date Fields are DateTime Objects: Ensure fields like TransactionDate and PreviousTransactionDate are correctly parsed as datetime objects for accurate analysis and comparison.
- Check for Duplicates

  - Scan for and remove any duplicate rows or redundant entries that may distort analysis, ensuring that each transaction record is unique.

- Validate Logical Consistency
  - Transaction Amounts: Ensure all Credit transactions have positive amounts.
- Check for Anomalies
  - IP Address Format: Ensure that all IP addresses are in valid IPv4 format.

These cleaning steps ensure that the dataset is ready for further analysis and that the results are not skewed by data quality issues.

4 Execution

Operations

Linked List Operations: A Linked List is a linear data structure where elements are stored in nodes. Each node contains the data and a pointer/reference to the next node.

- Insertions/Deletions: Inserting at the beginning or middle of the list is efficient (O(1)), but inserting at the end requires traversing the list (O(n)).
- Deleting an element requires O(n) to find the element, but once located, deletion is O(1).
- Accessing Data: Accessing elements by index is O(n) because you must traverse the list from the beginning to the desired position.
- Searching for an element is also O(n), as you need to traverse the entire list.
- Iterating: Traversing a Linked List is O(n) where n is the number of elements, as each element must be visited once.

Hash Table Operations: A Hash Table is a key-value pair data structure that allows for fast lookups using a hash function to index the keys.

- Insertions/Deletions: Insertions and deletions can be O(1) under average circumstances, assuming a good hash function and low collision rate. However, the worst-case scenario (due to collisions) is O(n).
- Accessing Data: Hash table lookups are O(1) on average, because the hash function computes the index directly, which gives access to the corresponding value.
- Searches: Searching by key is O(1) on average, but again, the worst case is O(n) if there are many collisions.
- Collisions: Hash tables handle collisions using methods like separate chaining (linked lists at each index) or open addressing (probing). If there are many collisions, performance can degrade.

Operation	Linked List	Hash Table
Search (by key)	O(n)	O(1) (average)
Insert at the beginning	O(1)	O(1)
Insert at the end	O(n)	O(1) (average)
Delete	O(n)	O(1) (average)
Traversal	O(n)	O(n) (due to key iteration)
Access by index	O(n)	O(1) (if key known)
Space Complexity	O(n)	O(n)

Implementation

- Linked List Implementation
  - Structure
    - The linked list stores elements sequentially, where each element points to the next.
    - It requires additional memory for each node to store the pointer to the next node.
  - Code Flow
    - You traverse the linked list from the beginning to detect fraud by checking each transaction's properties.
    - Searching and accumulating transaction amounts for the same account and date requires scanning each element sequentially, leading to a higher time complexity for fraud detection operations.
- Hash Table Implementation
  - Structure
    - The hash table stores key-value pairs, with each key hashed into a specific index.
    - It uses a hash function to quickly access the corresponding value.
  - Code Flow
    - Fraud detection leverages the key-value pairs in the hash table to efficiently group transactions by account and date, allowing for O(1) lookups and insertions in most cases.
    - By using a hash set for tracking flagged account-date pairs, it avoids redundant checks and ensures that we only flag high-value day transactions once.

Performance Analysis

Space Complexity:

- **Linked List**:

  - Space complexity is O(n), where `n` is the number of transactions, because each node requires space for both the data and a pointer to the next node.

- **Hash Table**:
  - Space complexity is also O(n) because each transaction is stored as a key-value pair in the hash table.
  - In case of collisions (depending on the resolution strategy), additional space might be used (e.g., linked lists for separate chaining).

Time Complexity:

- **Linked List**:
  - **Detecting Fraudulent Transactions**:
    - **Excessive Login Attempts**: O(n) for each transaction (since we must check each transaction).
    - **High-Value Day Transactions**: O(n) to check and sum the amounts for the same account and date.
  - **Overall**: O(n) for each transaction, leading to an O(n^2) complexity if we have to process all transactions multiple times.

- **Hash Table**:
  - **Detecting Fraudulent Transactions**:
    - **Excessive Login Attempts**: O(1) for each transaction, as we can directly access the data in constant time.
    - **High-Value Day Transactions**: O(1) for adding transaction amounts, O(1) for checking and flagging accounts if amounts exceed 2000.
  - **Overall**: O(n) for detecting fraud, which is more efficient than the Linked List approach.

Scalability:

- **Linked List**:
  - Linked lists tend to degrade in performance as the number of transactions increases because each operation (insertion, deletion, traversal) requires more time as the list grows.

- **Hash Table**:
  - Hash tables perform much better at scale because they allow for constant-time lookups, insertions, and deletions (on average), even as the number of transactions grows.
  - The performance degradation happens only if there are too many collisions, but modern hash tables (with good hash functions) handle collisions well.

Summary

- Linked List is simpler to implement but has poorer performance, especially for large datasets, due to its linear nature for searches, inserts, and deletions.
-
- Hash Table is more efficient for large datasets, with O(1) average time complexity for searching and inserting. However, hash table performance is subject to the quality of the hash function and the handling of collisions.

When to Use Each:

- Linked List: Suitable for applications with smaller datasets or when frequent insertions/deletions are required at both ends of the structure.
-
- Hash Table: Best suited for applications where fast lookups are crucial, and the dataset can grow large (e.g., fraud detection in large transaction systems).

Conclusion

The Hash Table provides a significant performance advantage for fraud detection due to its constant-time average operations, making it the more scalable and efficient choice compared to the Linked List. Therefore, for large-scale transaction data, the hash table is the better data structure, while the linked list could be considered for smaller-scale applications or cases where linked traversal is needed.