Hazelcast Certification – Code Challenge

Design & Implementation

# Introduction

This document contains the technical description of the solution’s components.

The TPS scalability and latency, the sizing of the cluster and the

Durability and availability of the system is also discussed in this document.

# Design of distributed data structure

The distributed data structure selected is an IMap with a credit card number as the key of the entry and the list of its associated transactions as the value.



1. Map definition in ‘IMDGCertification’ class

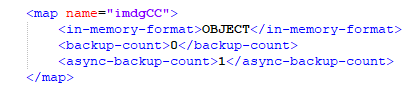
The main reason to use this data structure is that contains, in a single entry, all the historical transactions for a credit card. In consequence we assure that, when the fraud detector makes its calculations, the data needed is in the same member, so operations can be executed without the cost of extra network calls.

As map entries are not related among themselves there's no need to change Hazelcast default's partitions distribution.

As we are going to see in the next chapter of this document, an Entry Processor is used to perform the fraud detection process, this is the reason why it is used OBJECT as the in-memory-format in the Map in order to minimize serialization cost.

In the same way, the synchronous backup is disabled and a single asynchronous backup is configured to improve the speed of processing without losing data safety.

There is no a specific need to persist data so no map loader nor map store is used.

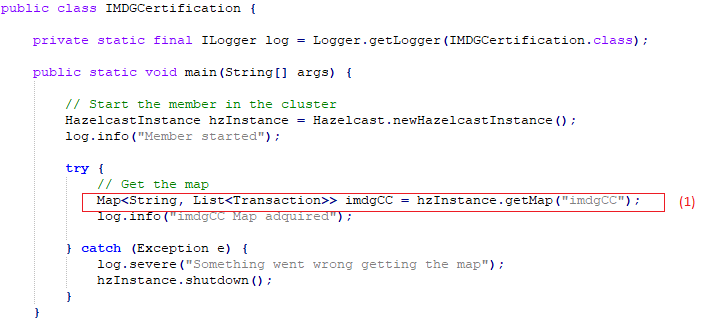


1. Map configuration in ‘hazelcast.xml’ configuration file

# Components used in the solution

## Cluster members

Cluster members are implemented by the **‘IMDGCertification’** class included in the ‘com.hazelcast.certification.cluster’ package.



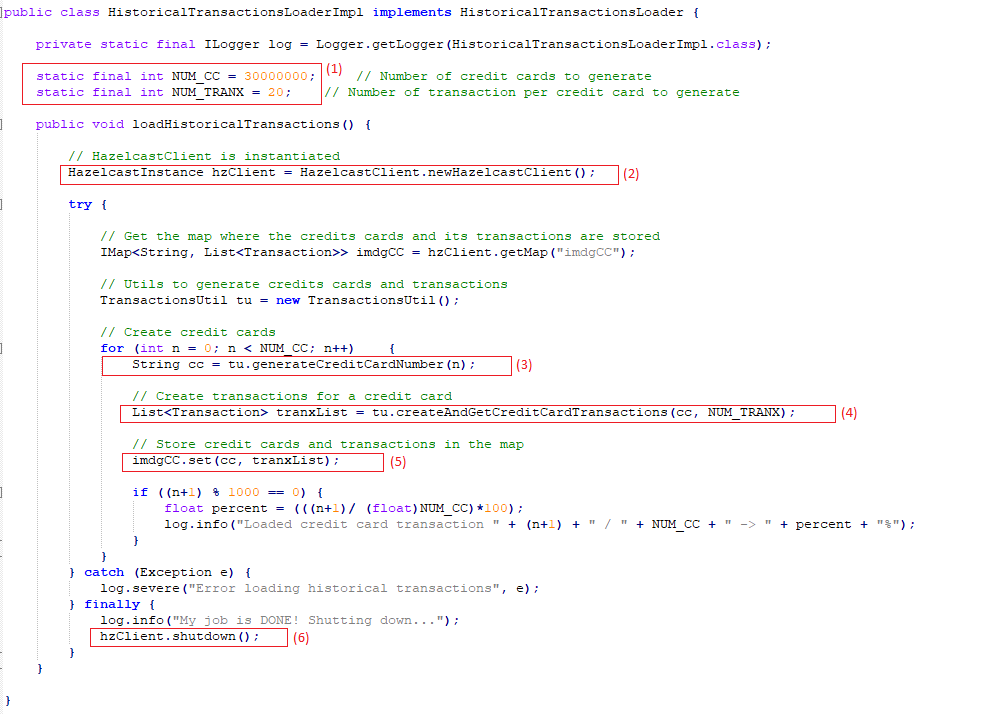
1. ‘IMDGCertification’ class

It’s a Hazelcast instance which its main objectives are: bring support to the IMap data structure described above and perform the fraud detection process using an Entry Processor. (1)

This class can be instantiated as many time as needed to achieve the optimal cluster performance.

## Historical transactions loader

Historical transactions loader process is implemented by the **‘HistoricalTransactionsLoaderImpl’** class included in the ‘com.hazelcast.certification.process’ package.



1. ‘HistoricalTransactionsLoaderImpl’ class

It’s a Hazelcast client that implements the **‘HistoricalTransactionsLoader’** interface, generating the historical data and loading it into the cluster’s Map. (2)

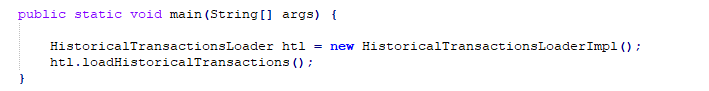
Credit cards numbers and historical transactions are generated by using **'TransactionsUtil'** methods. (3) and (4)

The set method is used instead of put, to store data in the Map to be more efficient because we don't need the stored value to be returned. (5)

Once all the data is stored in the Map the client shutdowns itself. (6)

The amount of data to load is controlled by the constants NUM\_CC and NUM\_TRANX. This is useful for testing purposes. (1)

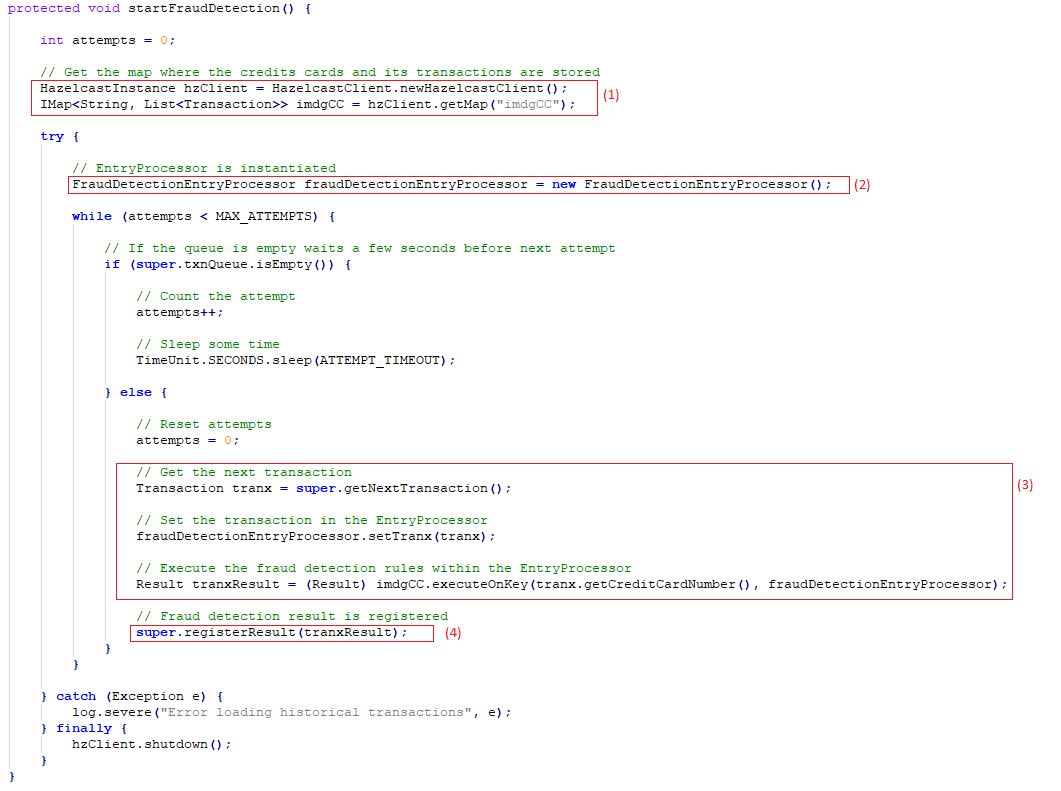
**‘Historical Transaction Runner’** class is in charge of putting the loader process in motion.



1. ‘HistoricalTransactionRunner’ class

## Fraud detection process

The business logic of the fraud detection process is implemented by the **‘FraudDetectionImpl’** class included in the ‘com.hazelcast.certification.process.impl.executorService’ package.



1. ‘FraudDetectionImpl’ class

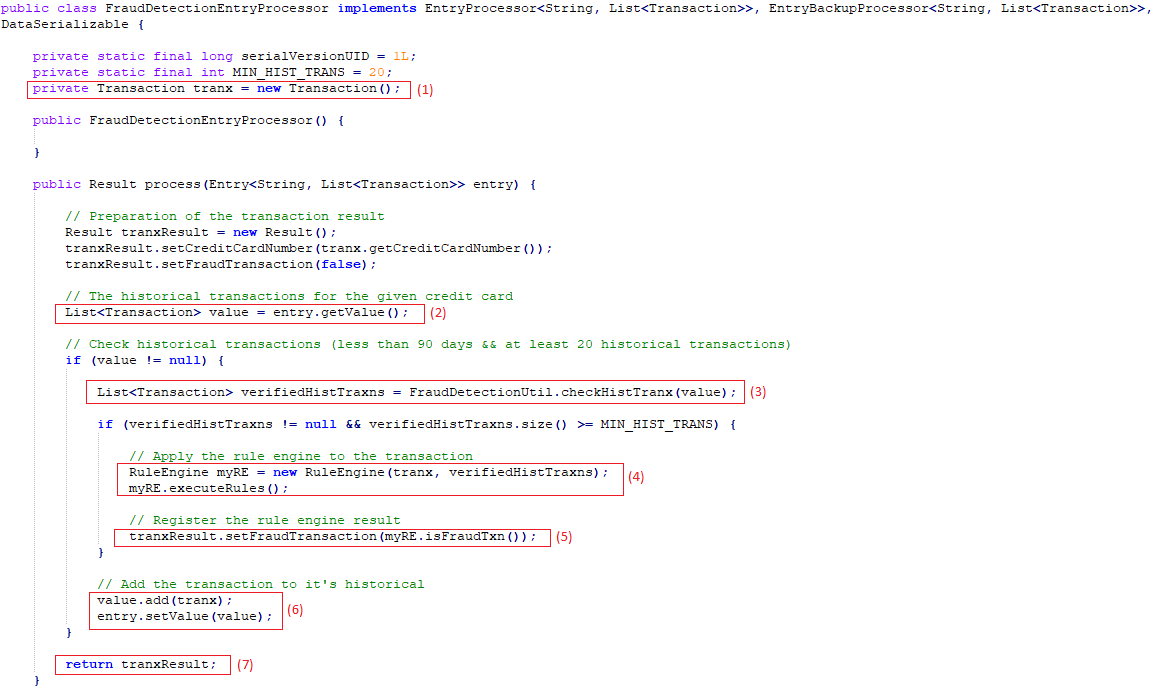
It’s a Hazelcast client that extents the **‘FraudDetection’** class, implementing the startFraudDetection method. (1)

Due that the fraud detection process basically consist on reading from the Map, modify the value, and putting the entry back in the Map an **Entry Processor** is chosen to perform this operations. The entry processor executes a read and updates upon the member where the data resides and this eliminates costly network hops. (2)

For each incoming transaction that is received from the transactions generator, the process sets the transaction in the entry processor and rely on it to execute the fraud detection process of the given transaction’s credit card number. (3)

The result of the fraud detection process is registered. (4)

The entry processor is implemented by **‘FraudDetectionEntryProcessor’** class and it’s applied on the key that contains the credit card’s transaction.



1. ‘FraudDetectionEntryProcessor’ class

As mentioned before, the Entry Processor contains the transaction that is going to be processed. (1)

The process verifies if the transactions of the entry’s credit card meets the requirements of number (at least 20 historical transactions) and age (with less than 90 days). To verify the age of the historical transactions a ‘checkHistTranx’ method is created in the **‘FraudDetectionUtil’** class and it is used by the entry processor. (2) and (3)

Historical transactions that don't meet the requirements don't participate in fraud detection process. The current transaction neither participates.

Rule engine for fraud detection process is provided with validated historical transactions of the credit card. (4)

The result of the fraud detection process is registered and the transaction is added to the collection of historical transactions of a credit card. (5), (6) and (7)

# TPS scalability and latency

Over time the system’s amount of data will grow and the fraud detection process will be more costly. For that reason it’s very important that the system can be easily scalable so it can have enough resources to store new data and perform the fraud detections operation with low latency.

With the proposed solution the system can scale horizontally by adding more members of the cluster. The system can also scale vertically by adding more resources (CPU, RAM) to the members of the cluster.

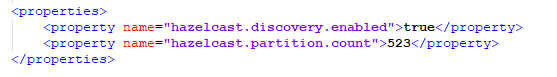
Due to members of the cluster have to store the data and perform the operations of fraud detecting, the best way to scale the system is by adding new members to the cluster.

To maintain a low-latency system the backup configured is asynchronous as explained below in more detail.

# Sizing of the cluster to achieve a particular performance

As we can see in appendix 1, the size of the initial amount of data that the cluster have to store is 51 GB approximately.

We need to increase the partition count to the point where data load per-partition is under 100 MB. That gives us the number of **523 partitions**, which is also a prime number, and that’s mean that the data will be distributed to the members almost evenly.



1. Partition count in ‘hazelcast.xml’ configuration file

However, with 523 as partition count, the partition size is too close to the limit of 100 MB. A forecast the projected data growth would be necessary to calculate to resize the number of partitions.

# Durability and availability of the application

As it was mentioned in the design of distributed data structure section, the map that stores the credits cards data and it’s configured to have one asynchronous backup.

The backup allows to maintain a copy of the data and restore them if any member of the cluster goes down. As we want to prioritize performance over the backing up, the asynchronous backup is choose because don’t block operations.

On the other hand the durability of the data is not ensure because no map store is used for data persistence. If all nodes fails or are shut down, the data will be lost.

# Appendix

## Appendix 1. Partition count calculation

### Historical data size

Number of credits cards: 30 million

Size of one credit cards: 14 chars \* 2 bytes = 28 bytes

**Total size of the credits cards: 28 bytes \* 30 million = 801 MB**

Number of initial transactions for each credit cards: 20 transactions

Size of one transaction: 90 bytes

* creditCardNumber: 28 bytes
* timestamp: 8 bytes
* countryCode: 6 bytes
* responseCode: 4 bytes
* txnAmount: 10 bytes
* txnCurrency: 6 bytes
* txnCode: 10 bytes
* merchantType: 8 bytes
* txnCity: 10 bytes

**Total size of transactions: (90 bytes \* 20 transactions) \* 30 million = 50,29 GB**

**Total size of historical data: 801 MB + 50,29 GB = 51,07 GB**

### Partition count

**Partition count: 51,07 GB / 100 MB = 523 partitions**