Hazelcast Certification – Code Challenge

Design & Implementation

# Introduction

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

TPS scalability and latency, sizing of the cluster and durability and availability of the system is also discussed.

# Design of distributed data structure

The distributed data structure selected for the solution 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

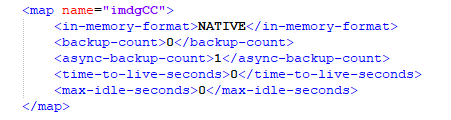
The main reason to use this data structure is that contains, in a single entry, all the historical transactions for a credit card. In consequence we assure that, when the fraud detector makes its calculations, the data needed is in the same member, so operations can be executed without the cost of extra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

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

High-Density Memory Store is used to solve garbage collection limitations so the application can exploit hardware memory more efficiently without the need of oversized clusters[[1]](#footnote-1). This is another reason to choose the IMap as the distributed data structure selected because it supports the HD Memory. In consequence, IMap in memory format is configured as NATIVE.

Additionally, 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.

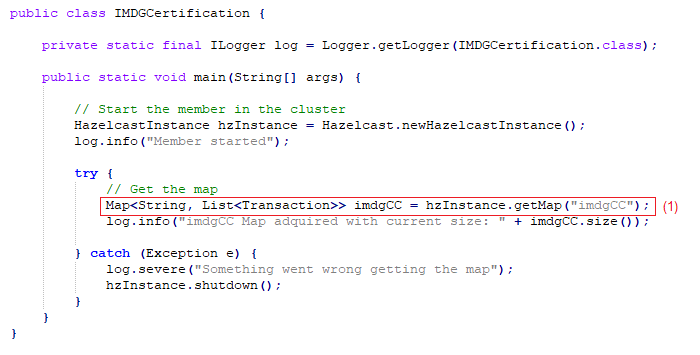


1. Map configuration in ‘hazelcast.xml’ configuration file

# Components used in the solution

## Cluster members

* Cluster members are implemented by the **‘IMDGCertification’** class.



1. ‘IMDGCertification’ class

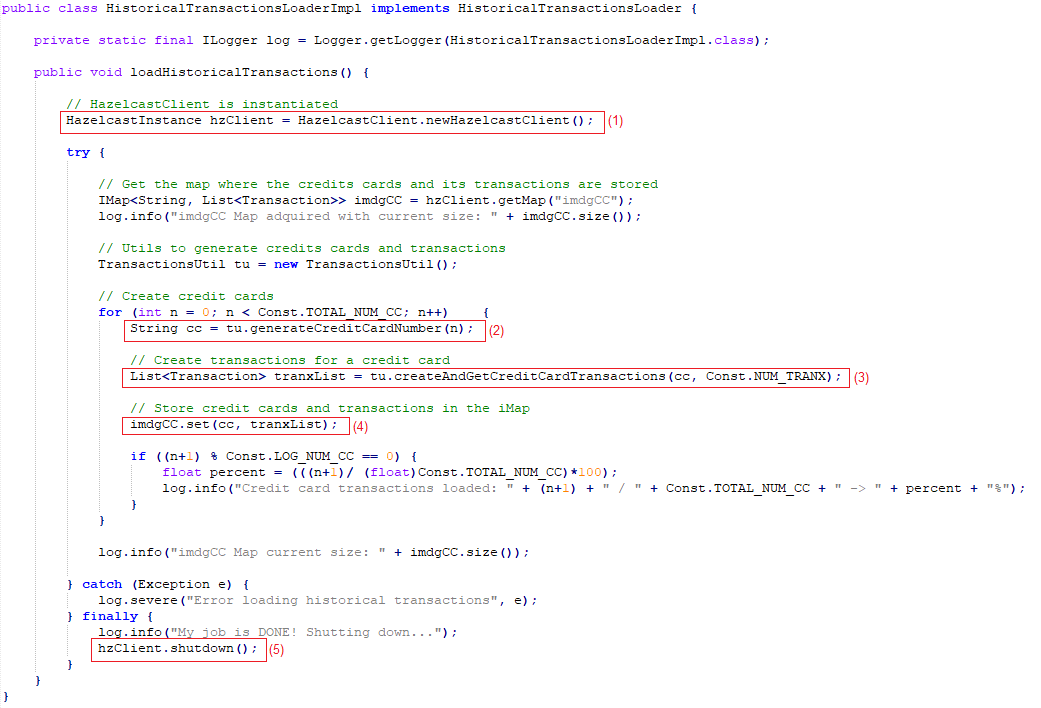
It’s a Hazelcast instance (1) which its main objectives are: bring support to the IMap data structure described above and to perform the fraud detection process using an Entry Processor.

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

## Historical transactions loader

Is an independent process that runs in a separate JVM and whose task is to load Hazelcast cluster with historical data. Historical data consists of 30 million credit cards and 20 transactions per card.

* Historical transactions loader process is implemented by the **‘HistoricalTransactionsLoaderImpl’** class.



1. ‘HistoricalTransactionsLoaderImpl’ class

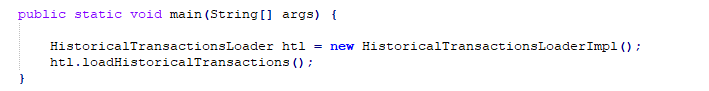
This class implements the ‘HistoricalTransactionsLoader’ interface and creates a Hazelcast client (1) to generate the historical data and loading it into the cluster’s Map.

Credit cards numbers and historical transactions are generated using 'TransactionsUtil' methods (2) and (3).

The set method is used, instead of put, to store data in the Map because is more efficient[[2]](#footnote-2) and we don't need the stored value to be returned (4).

Once all the data is stored in the Map the client shutdowns itself (5).

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



1. ‘HistoricalTransactionRunner’ class

## Transactions Generator process

* ‘**TransactionsGenerator’** is the source that sends transactions to the application for fraud detection. No changes were needed to the transaction generator process.

## Fraud detection process

* **‘FraudDetectionServer’** is the consumer of transactions sent by ‘TransactionsGenerator’. The process entails applying of business rules on the received transaction and subsequently registering the results.
* The business logic of the fraud detection process is implemented by the **‘FraudDetectionExecutorImpl’** class.



1. ‘FraudDetectionExecutorImpl’ class

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

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 these operations. The entry processor executes a read and updates upon the member where the data resides and this eliminates costly network hops[[3]](#footnote-3).

To improve the system’s performance, the entry processor is executed within a Callable Task using the ExecutorService framework. This allows to asynchronously execute fraud detection tasks and manage its results[[4]](#footnote-4) [[5]](#footnote-5). Therefore, two threads are created, one to send transactions to the fraud detection calculations and other to consume the result of these calculations.

Both threads uses a buffer to store the futures and to process them respectively (2).

The ‘putBuffer’ thread obtains the ExecutorService instance with the number of threads indicates by the ThreadNumber (which is defined in the fraudDetection.properties) (3), gets the transactions sent by the transactions generator and submits the FraudDetectionCallableTask to the executor service storing in the buffer a result of type future (4).

The ‘getBuffer’ thread reads the buffer of futures and processes the results (5).

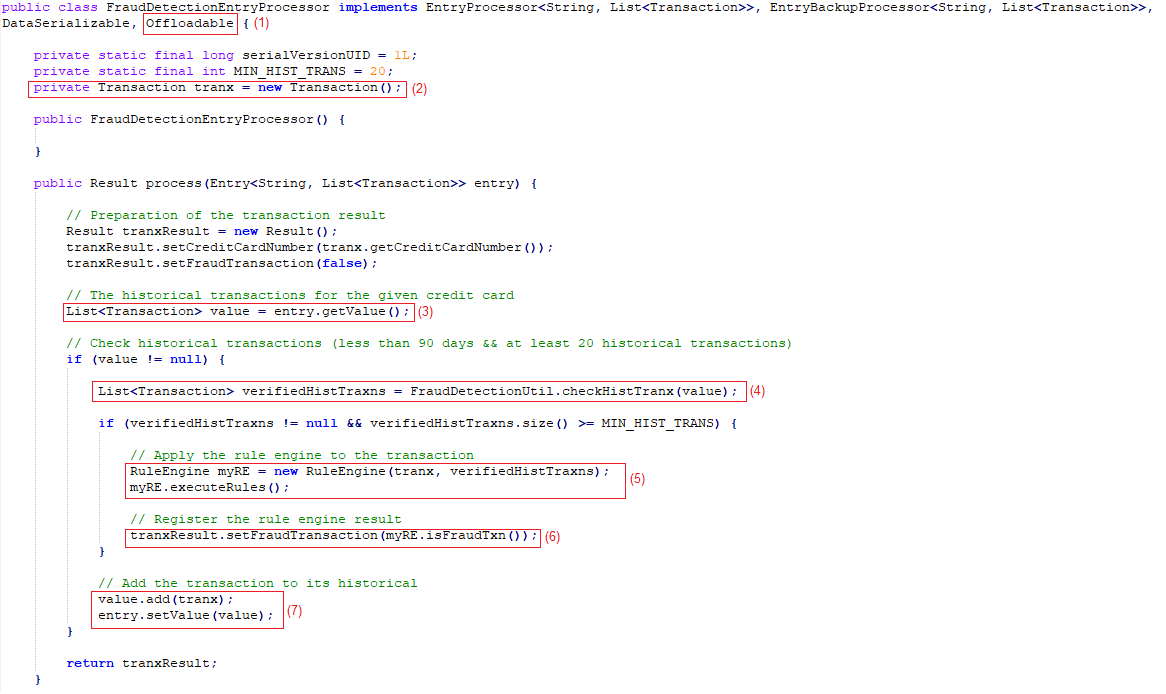
* **‘FraudDetectionCallableTask’** is the implementation of the Callable Task that executes the entry processor.



1. ‘FraudDetectionCallableTask’ class

The ‘call’ method executes the entry processor and returns the result of the fraud detection rules applied to the transaction (1).

* 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

The entry processor implements the Offloadable interface to improve the performance[[6]](#footnote-6) (1).

The entry processor contains the transaction that is going to be processed (2).

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 (3) and (4).

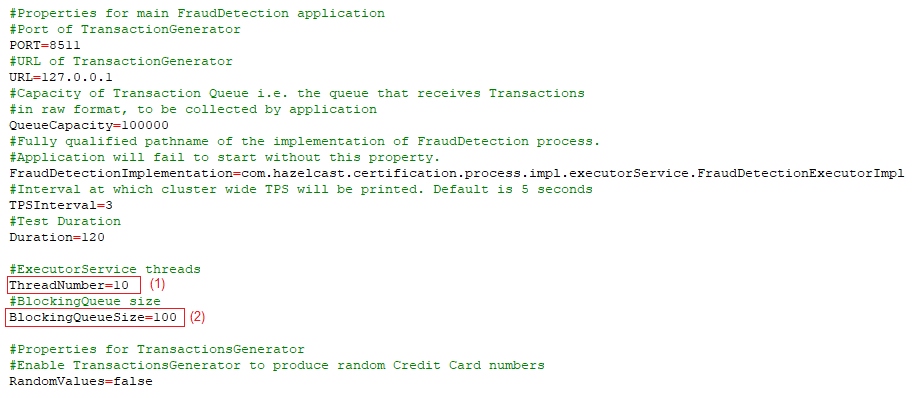
Historical transactions that don't meet the requirements don't participate in fraud detection process. The current transaction neither participates.

Rule engine for fraud detection process is provided with validated historical transactions of the credit card (5).

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

## Configuration files

* Two new entries has been set in the **‘FraudDetection.properties’** file.



1. ‘FraudDetection.properties’ file

‘ThreadNumber’ indicates the number of threads used by the executor service. (1)

‘BlockingQueueSize’ indicates the size of the futures buffer.

* **‘hazelcast.xml’** file.



1. ‘hazelcast.xml’ file

With the amount of historical data that we have to load in the members of the cluster, the default partition count must be modified according to our needs (1). See in more detail in the following chapters of this document.

HD Memory is used in order to exploit hardware memory more efficiently without the need of oversized clusters[[7]](#footnote-7) (2).

‘Result’ and ‘Transaction’ domain classes has been changed from DataSerializable to IdentifiedDataSerializable for a faster serialization of objects[[8]](#footnote-8) (3).

# 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 detection operations with low latency.

Because cluster members are responsible for storing credit card and transactions data and performing fraud detection operations, the scalability of the system is achieved by being able to increase the number of cluster members or by increasing the resources (CPU, RAM) of existing members.

Adding new members to the cluster (horizontal scalability) is the easiest way using Hazelcast, since it allows increasing the total amount of RAM memory in the cluster to store more data and increase the capacity of transaction processing.

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

# Sizing of the cluster to achieve a particular performance

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

This implies that the minimum configuration of the cluster must allow the storage of those 51 GB of data plus those corresponding to the asynchronous backup that has been defined.

In addition, it is necessary to take into account the expected data growth so that the system can store them and perform the fraud detection operations.

To achieve a particular performance, the cluster has to be configured in such a way that the combination of RAM and processing capacity will allow to reach the objective.

On the other hand, we need to increase the partition count to the point where data load per-partition is around 50 MB. That gives us the number of **1049 partitions**, which is also a prime number, and that’s mean that the data will be distributed to the members almost evenly and we will have capacity enough to assume the data growth.

# 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 and transactions data is 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 chosen 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.

# Success criteria completion

## Application must store 30 million credit cards before commencing Fraud Detection for incoming transactions

This criteria was accomplish in an Amazon AWS environment with 8 members of the clusters running the IMDG process (runIMDG.sh) and 1 client running the historical transaction loader process (runHistTxnLoader.sh). The 9 instance types used were m5a.xlarge with 4 vCPUs and 16 GB of RAM memory.

Credit cards numbers are generated using the ‘generateCreditCardNumber’ method of the ‘TransactionsUtil’ class within the historical transaction loader process (HistoricalTransactionsLoaderImpl.java).

## Each credit card must also have 20 historical transactions, generated in past 90 days

This criteria were accomplished along with the previous one by using the ‘createAndGetCreditCardTransactions’ method of the ‘TransactionsUtil’ class within the historical transaction loader process (HistoricalTransactionsLoaderImpl.java).

## Each incoming transaction that is received from TransactionsGenerator gets added to the collection of historical transactions of a credit card

‘FraudDetectionEntryProcessor’ class is the responsible to store the transaction received, and processed, to the historical transactions of the entry's credit card.

## Rule Engine for fraud detection process requires historical transactions (excluding current transaction) of a credit card. Therefore, RuleEngine must be provided with the current transaction that is received from TransactionsGenerator and all historical transactions of the same credit card

For every transaction received the ‘FraudDetectionExecutorImpl’ class executes a callable task that set the transaction into the ‘FraudDetectionEntryProcessor’ class. The Entry Processor then gets the credit card of the transaction and uses it to obtain the historical transactions of the given credit card from the map. Transaction and historical transactions are provided to the rule engine to perform the fraud detection process.

## No historical transaction will be valid for more than 90 days i.e. any historical transaction that has passed its expiry must not participate in Fraud Detection process.

Once the ‘FraudDetectionEntryProcessor’ gets the historical transactions, first it excludes the transactions with more than 90 days old (using ‘checkHistTranx’ method of the ‘FraudDetectionUtil’ class). Then it checks that the credit card has at least 20 valid historical transactions (not including the transaction given for fraud detection process).

## Result of every transaction that was read by invoking getNextTransaction() must be registered with the application (registerResult()) after running fraud detection process.

This was perform by the ‘FraudDetectionExecutorImpl’ class.

# 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 / 50 MB = 1049 partitions**

1. https://docs.hazelcast.org/docs/latest-dev/manual/html-single/index.html#storage [↑](#footnote-ref-1)
2. https://blog.hazelcast.com/performance-top-5-1-map-put-vs-map-set/ [↑](#footnote-ref-2)
3. https://docs.hazelcast.org/docs/latest-dev/manual/html-single/index.html#entry-processor [↑](#footnote-ref-3)
4. https://docs.hazelcast.org//docs/latest-development/manual/html/Distributed\_Computing/Executor\_Service/Implementing\_a\_Callable\_Task.html [↑](#footnote-ref-4)
5. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html [↑](#footnote-ref-5)
6. https://docs.hazelcast.org/docs/latest-dev/manual/html-single/index.html#offloadable-entry-processor [↑](#footnote-ref-6)
7. https://blog.hazelcast.com/introduction-hazelcast-hd-memory/ [↑](#footnote-ref-7)
8. https://docs.hazelcast.org/docs/latest-dev/manual/html-single/index.html#identifieddataserializable [↑](#footnote-ref-8)