EB5001 - Big Data Engineering for Analytics

Project Report & Experience Summary

Team 17

Bharat Nagaraju (A0178258N)

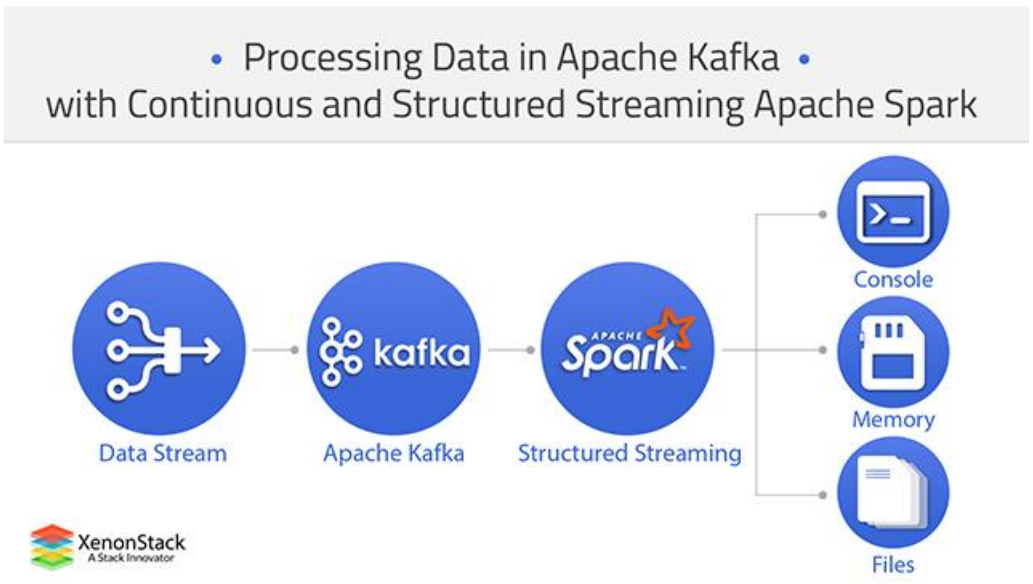
Vigneshram Selvaraj (A0178215A)

Gaurav Sharma (A0150332)

Table of Contents

1) Executive Summary

Design an end-to-end to solution for a bigdata based system to provide a real time analytics and provides insights. The design of such system can be broken down to have big data ingestion, distributed storage, processing and analytics model that provide real time insights aided by some visualization. To this end, we have consumed a real-time api, pre-process & transform the data and persist into a distributed storage platform such as HDFS or a NoSQL Database. Planned implementation is below



2) Big Data Challenges

There are three main challenges everyone faces when they start working on a Big Data Projects. First, Understand the data so to create a dataset which you can use to perform analysis on. This is to ensure understand what's the data is about before solving any big data problem.

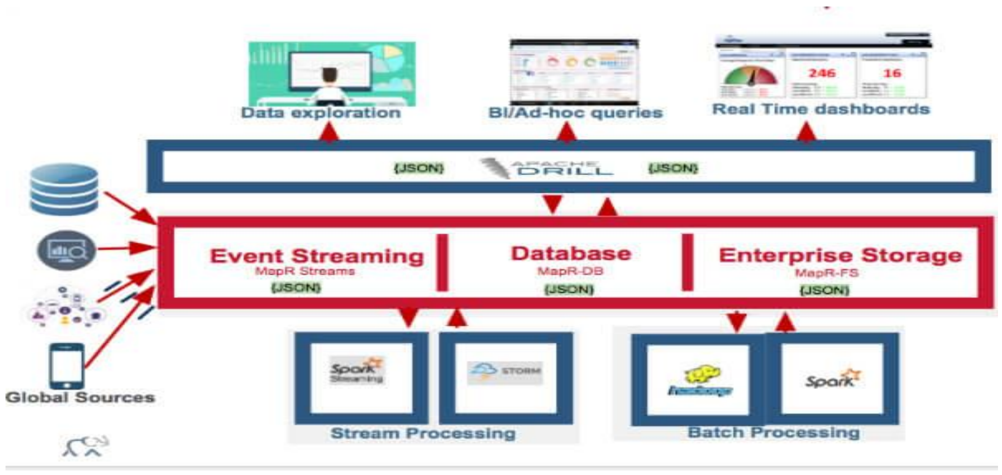
The second challenge is to understand if the data shared is clean a datasets and ensure we are using proper datasets for validation and testing of model, thus it again goes back to understand the problem what we want to solve so that we can create correct model to use.

The third challenge is the ability to capture, process, and store ever-expanding data streams while maintaining data quality. Real-time pre-processing of streaming data becomes increasingly difficult without a robust and scalable architecture. This area speaks to the need for further innovation in the big data engineering domain, which is currently built on Hadoop, Cassandra, Kafka, etc.

3) Proof of Concept

a) MapR Platform as Infrastructure

We Initially planned to use (***MapR-Sandbox-For-Hadoop-6.1.0***) which provides the single-node cluster of MapR's core data storage for files, tables, and streams, ecosystem components for Hadoop, HBase, Hive, Hue and Pig.



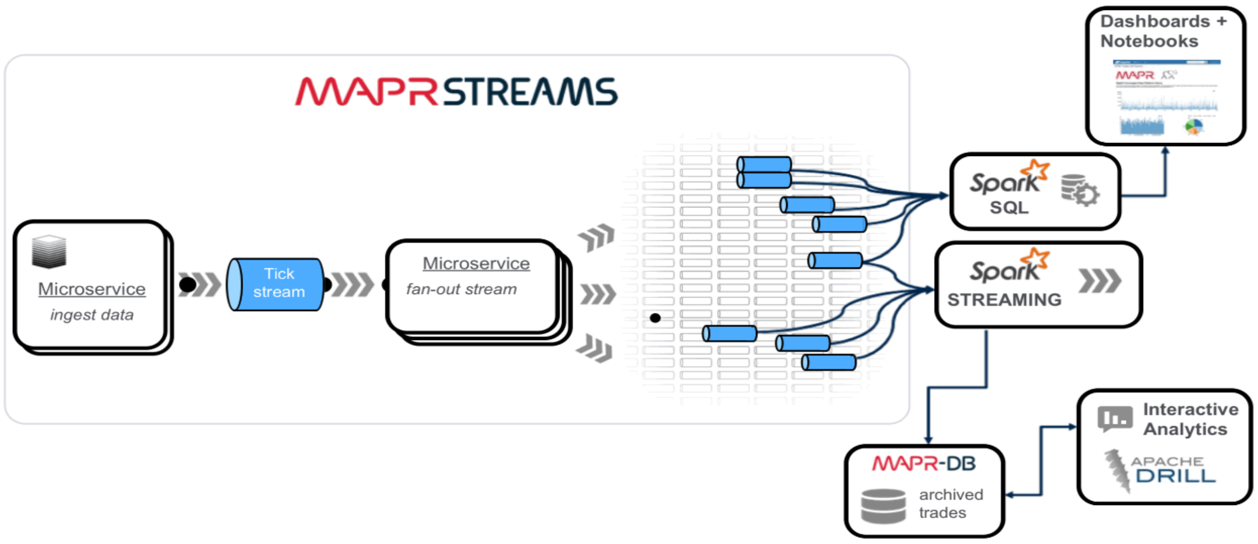
On the sandbox we then add the following packages:

- Git
- Maven
- Spark
- Kafka
- Zeppelin
- Java 1.8

We tested the infrastructure with the sample project (FinServ-application-blueprint) provided by MAPR itself, the architecture of which is explained in the following section.

b) *High level Application Architecture*

The Idea was to create an application shown in the below diagram showing how data will move in the architecture. The rounded rectangles represent processes that produce and/or consume data from MapR Streams. Java based microservices will be used to ingest and manipulate streaming data using the Kafka API. Spark and Apache Zeppelin will be used to provide streaming analytics and batch-oriented visualization.



c) *Success Factors*

- ✓ Once the said packages were installed, able to run the service using mock data
- ✓ Producer microservice streams trades data from a file-based source
- ✓ Publish the stream to Kafka distributed system.
- ✓ Spawn multiple threads to consume data from Kafka
- ✓ Dynamically create Schema based on the incoming data
- ✓ Persist to MAPR DB by applying transformation such as adding unique id for message, convert date time format and typecast the data to store in a structured fashion.
- ✓ Run analytical queries on MAPRDB
- ✓ Effective to issue command line installations

d) *Challenges*

- To store the result to Hive instead of MapR to access from visualization tools, we faced driver support issues and to instead use MapR Db for some tools required paid version to access the following support.
- Due to hive driver issues, unable to store structured data to drive from Spark

e) *Cloudera Sandbox as Infrastructure*

Next alternative was Cloudera Distributed Hadoop (CDH) version 6.3. This sandbox is very resource heavy, and it was a challenge running it on a personal laptop. Installing other required components such as SPARK, KAFKA, Zeppelin was quite easy, but running the programs consumed more resources and the Virtual machine was not responsive while using IDEs and not suitable to high speed streaming data. So, it was decided to move on with a infrastructure with more muscle power

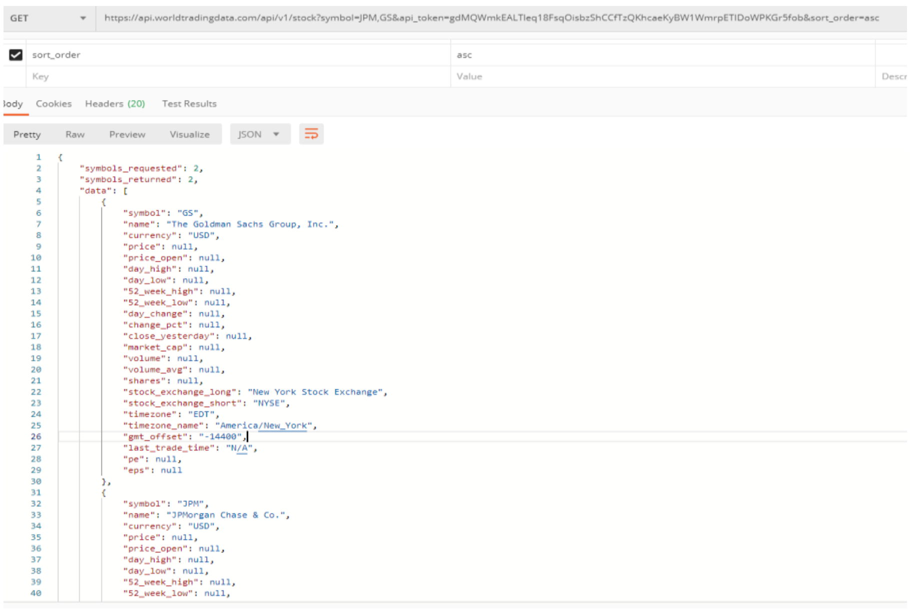
f) *Data sources – Real Time APIs*

There were few sources which we initially considered for APIs. We have ignored the sources which have been decommissioned in the past and we have considered active sources

- ➢ Yahoo Finance (De-commissioned)
- ➢ Google Finance in Google Sheets (Excel Data file not Real Time)
- ➢ Alpha Vantage (Can pass single share)
- ➢ World trading data (Can pass Multiple share)

i) World trading data:

As per there API documentation mentioned since we can pass multiple companies. However, after signing up for the API, data values were not returned as described in the project documentation and schema. Although the documentation stated several pros such as more number of allowed api calls, allowing multiple company stocks to be read, and data from wide range of market data, the effort was in vain

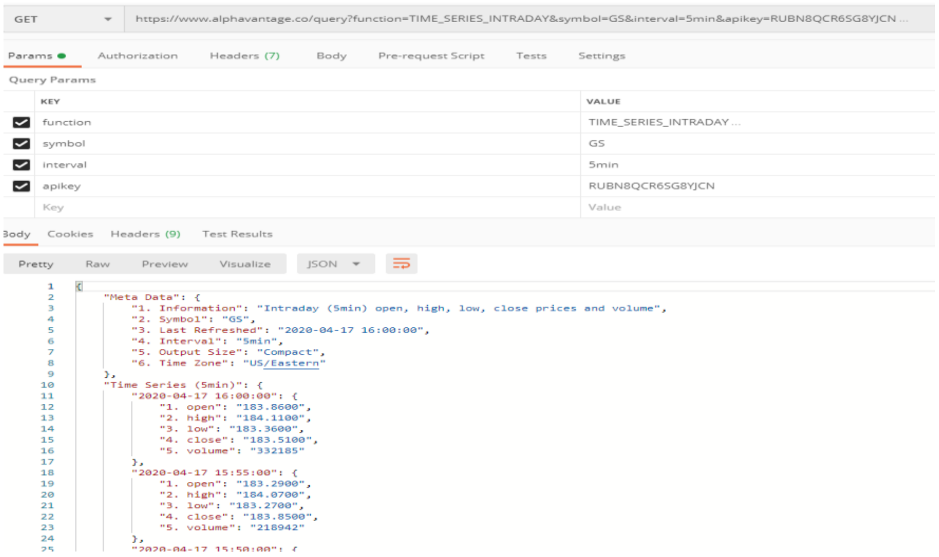


```
GET  https://api.worldtradingdata.com/api/v1/stock?symbol=JPM,GS&api_token=gdMQWmkEALTleq18FsqOisbzShCCfTzQKhcaeKyBW1WmrpETIDoWPKGr5fob&sort_order=asc

sort_order    asc
```

```
{
    "symbols_requested": 2,
    "symbols_returned": 2,
    "data": [
        {
            "symbol": "GS",
            "name": "The Goldman Sachs Group, Inc.",
            "currency": "USD",
            "price": null,
            "price_open": null,
            "day_high": null,
            "day_low": null,
            "52_week_high": null,
            "52_week_low": null,
            "day_change": null,
            "change_pct": null,
            "close_yesterday": null,
            "market_cap": null,
            "volume": null,
            "volume_avg": null,
            "shares": null,
            "stock_exchange_long": "New York Stock Exchange",
            "stock_exchange_short": "NYSE",
            "timezone": "EDT",
            "timezone_name": "America/New_York",
            "gmt_offset": "-14400",
            "last_trade_time": "N/A",
            "pe": null,
            "eps": null
        },
        {
            "symbol": "JPM",
            "name": "JPMorgan Chase & Co.",
            "currency": "USD",
            "price": null,
            "price_open": null,
            "day_high": null,
            "day_low": null,
            "52_week_high": null,
            "52_week_low": null,
```

ii) Alpha Vantage

Considering other alternative sources, Alpha Vantage turned out to provide only half the number of api calls. This results in lesser number of api calls in its free version. Number of stocks provided were less too. So, we decided to skip to next data source



```
GET  https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&symbol=GS&interval=5min&apikey=RUBN8QCR6SG8YJCN ...
```

Query Params

KEY	VALUE
function	TIME_SERIES_INTRADAY ...
symbol	GS
interval	5min
apikey	RUBN8QCR6SG8YJCN

```
{
    "Meta Data": {
        "1. Information": "Intraday (5min) open, high, low, close prices and volume",
        "2. Symbol": "GS",
        "3. Last Refreshed": "2020-04-17 16:00:00",
        "4. Interval": "5min",
        "5. Output Size": "Compact",
        "6. Time Zone": "US/Eastern"
    },
    "Time Series (5min)": {
        "2020-04-17 16:00:00": {
            "1. open": "183.8600",
            "2. high": "184.1100",
            "3. low": "183.3600",
            "4. close": "183.5100",
            "5. volume": "332185"
        },
        "2020-04-17 15:55:00": {
            "1. open": "183.2900",
            "2. high": "184.0700",
            "3. low": "183.2700",
            "4. close": "183.8500",
            "5. volume": "218942"
        },
        "2020-04-17 15:50:00": {
```

Based on the above results, we have decided to take another API which does not have any limitations and available 24*7 will be used. It will be discussed in section 6.

4) Proposed Architecture

After experimenting with a couple of different Sandbox environments and different APIs, we decided to run a sandbox on cloud and use some Realtime data source without any limitations on the number of calls made in a day. The overall landscape and the data flow along with the technology stack is shown in the diagram

We will list down the required components and set up activities and go over each component

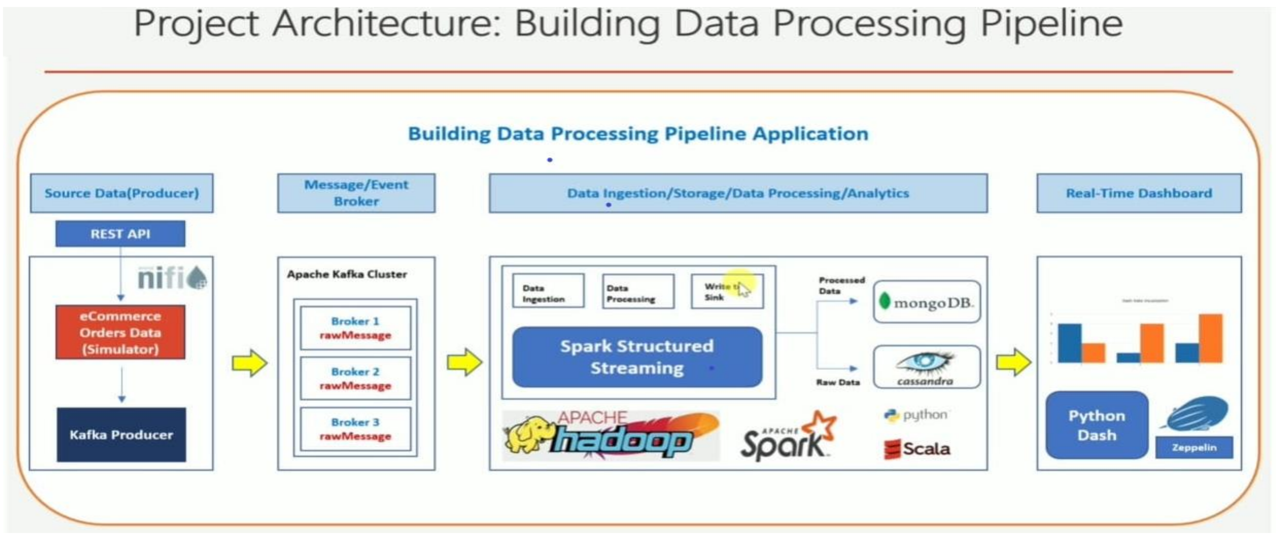


Figure: Big Data Architecture

a) Required Components

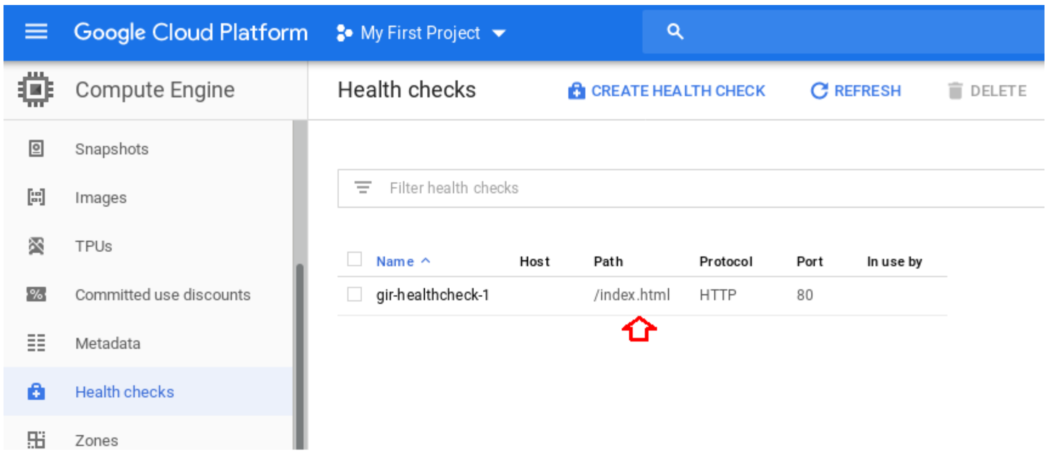
- Google Cloud – As a platform with CENTOS Operating system and storage
- Cloudera Manager - that includes – Hue, Hadoop, YARN, HIVE, HBASE, Zookeeper etc. as shown below as a basic big data platform hosted on the LINUX VM on cloud
- NIFI – As a Workflow engine to make Api calls, transform data and publish it to a real time queue-based system such as Kafka
- KAFKA – Provides a platform as Producer and Consumer to emit out data that can be used in data pipelines
- Cassandra – A columnar data source with redundancy to persist structured data
- MongoDB – Structured document datastore
- SPARK – Data Ingestion and pipeline before data persistence
- Zeppelin – Dashboard tool + Python Dash for Visualization

5) Infrastructure

As a first step, we went on to set up the infrastructure on the cloud

a) Cloud based Infrastructure

As the local Cloudera CDH was very resource heavy, we went ahead to use the cloud-based approach. We created a virtual machine in Google Cloud (GCP) on CentOS and then installed the following components. First, set up to access the remotely using SSH. Generate the public and private keys to access the server from terminal such as Putty



Name	Host	Path	Protocol	Port	In use by
gir-healthcheck-1		/index.html	HTTP	80	

Fig: Cloud Console

b) Installation and Configuration

1. Setup public/private ssh keys for accessing the VM
2. Install Cloudera Manager that includes – Hue, Hadoop, YARN, HIVE, HBASE, Zookeeper etc. as shown below
3. Install
   3.1 NIFI
   3.2 Kafka
   3.3 Cassandra
   3.4 MongoDB
   3.5 Zeppelin
4. Setup Internal/External IP for (Kafka, Cassandra, Mongo, Zeppelin)
5. Create Databases
   5.1 Key Spaces (Cassandra)
   5.2 Collections (MongoDB)
6. Create Tables
   6.1 Document (Mongo DBs)
   6.2 Tables (Cassandra)
7. Python and related libraries – PySpark, Spark, Dash and dependencies

6) Realtime data - API

Based on the initial study on different APIs, we decided use *User details* api which is been is widely used for application testing purpose and is free to use with no limitation and can we used to fetch user records in terms of country wise also.
We are using this api - https://randomuser.me/api/0.8

Schema:

```
{
  "results":[
    {
      "user":{
        "gender":"female",
        "name":{
          "title":"miss",
          "first":"purificacion",
          "last":"vidal"
},
        "location":{
          "street":"4946 calle de segovia",
          "city":"zaragoza",
          "state":"la rioja",
          "zip":95219
},
        "email":"purificacion.vidal@example.com",
        "username":"beautifulgorilla727",
        "password":"cinema",
        "dob":439887951,
        "phone":"912-128-419",
        "picture":{
          "medium":"https://randomuser.me/api/portraits/med/women/3.jpg",
          "thumbnail":"https://randomuser.me/api/portraits/thumb/women/3.jpg"
}
}
}
],
  "nationality":"ES",
  "seed":"48a3abf6be431ce905",
  "version":"0.8"
}
```

7) Conceptual Design

a) Building Data Pipelines

When building big data pipelines, we need to think on how to ingest the volume, variety, and velocity of data showing up at the gates of what would typically be a Hadoop ecosystem. Preliminary considerations such as scalability, reliability, adaptability, cost in terms of development time, etc. will all come into play when deciding on which tools to adopt our requirements.

b) NIFI

NiFi was created to automate the flow of data between systems. While the term 'dataflow' is used in a variety of contexts, we mean the automated and managed flow of information between systems. NIFI can handle messages with arbitrary sizes and provide a drag-and-drop Web-based UI. NiFi runs in a cluster and provides real-time control that makes it easy to manage the movement of data between any source and any destination. It supports disparate and distributed sources of differing formats, schema, protocols, speeds, and sizes.

Since NiFi is a data flow management and thus the flow logic does not reside in producer/consumer, but lives in the broker, thus allowing centralized control, combining the following with Kafka provides reliable stream data storage while complex dataflow logic is handled in NiFi.

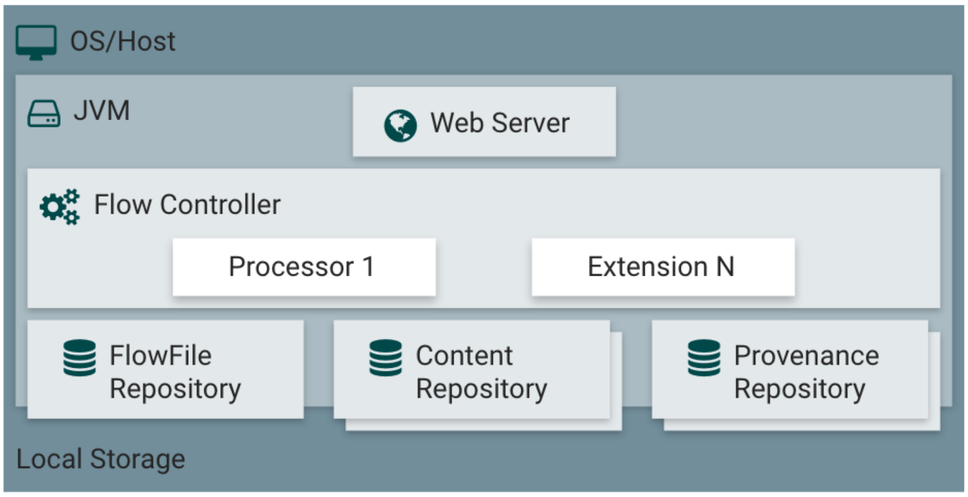


Figure: Illustration of Apache NiFi architecture.

c) *Kafka*

Kafka is a distributed, high-throughput message bus that decouples data producers from consumers. Messages are organized into topics, topics are split into partitions, and partitions are replicated across the nodes called brokers in the cluster. It offers better scalability and message durability. Kafka can be used for event processing and integration between components of large software systems. Data spikes and back-pressure (fast producer, slow consumer) are handled out-of-the-box

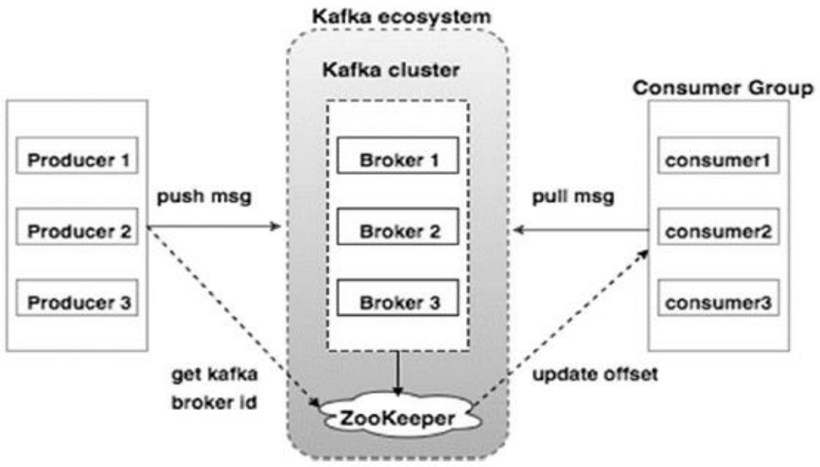


Figure: Illustration of Apache Kafka architecture.

d) *Data Storage strategy*

For building real-time data pipelines and streaming, Apache Kafka acts as a tool for processing and obtaining data for immediate use or storage in a database. It aims to provide a unified, high-throughput, low latency platform for handling real-time data feeds. It is a publish-subscribe-based messaging system that exchanges data between processes, applications and servers. Once the messages are retrieved from Kafka, strategy is to divide the data storage in two parts i.e. (Store the RAW with Transformed Data and Aggregated data for Dashboard Visualization).

e) *Store Raw Data (Cassandra):*

When the data is read from Kafka using SPARK Streaming API, Raw data needs to be persisted.

We have considered Cassandra, a columnar store to persist the data. It is a distributed, decentralized and an open-source database or a storage system. It is basically used for managing very large amounts of structured data. There is no single point of failure, providing highly available services. Below features are useful to store huge amounts of incoming data to be used for reconciling with aggregated data

Key benefits of Cassandra Architecture

a. Elastic Scalability - can be scaled up and down without any downtime
b. Open Source- compatible with other Apache projects
c. Peer-to-peer architecture - rare chances for failure as we add or remove clusters
d. Fault tolerance - high-level backup and recovery competencies based on no. of replica
e. Great analytics possibility

Solr based integrated search
Batch analytics integrating Hadoop with Cassandra
External batch analytics powered by Hadoop and Cloudera/Hortonworks
Spark based near real time analytics

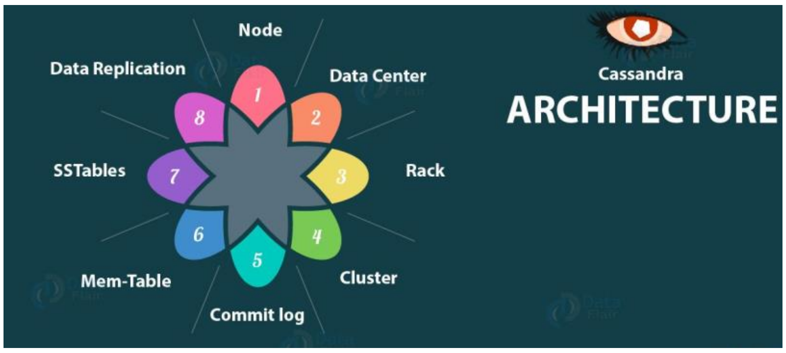


Figure: Illustration of Cassandra architecture.

f) Store Aggregated Data (MongoDB)

Perform Data aggregation by running the processing the jobs for the dashboard to show the result. After researching available options, we used Mongo, for the following reasons

It is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, it makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB The following have a powerful query processing and supports ACID transactions and joins in queries which makes it suitable to graphs display.
The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily based on needs.

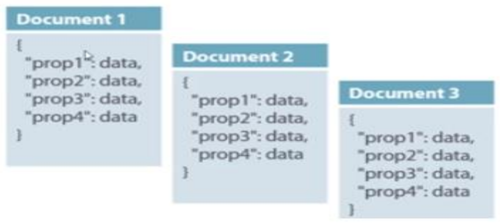


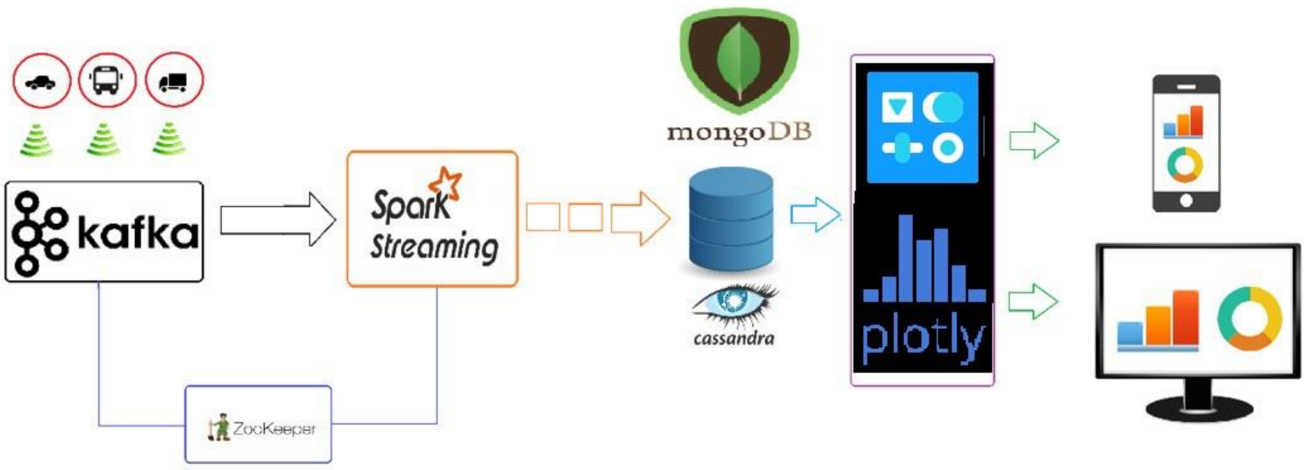
Figure: Illustration of Mongo Storage

Expressive Query Language & Secondary Indexes
Strong Consistency
Enterprise Management & Integrations
Flexibility
Scalability & Performance
Always On, Global Deployments

Figure 1: MongoDB Nexus Architecture, blending the best of relational and NoSQL technologies

8) Visualization & Analytics

For data visualization, Zeppelin, A web-based notebook that enables data-driven, interactive data analytics and collaborative documents with No-SQL, Scala, python. We used the python-dash library to show visualization. We have used the Aggregated Data stored in mongo database on the apache zeppelin using the python-dash libraries.



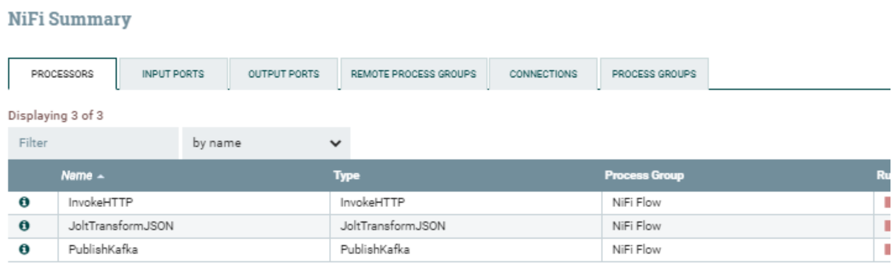
9) Solution Implementation

a) Nifi:

For the data Ingestion process, we used ***Apache Nifi***, Since the following offers many components to create data flows for any type of protocols or data sources. In our case the we are using Rest based approach to get the data.

In order to achieve our requirement, we need to create a flow. To design one such flow, we used the ***Nifi*** Graphical designer studio and drags the components from menu bar to canvas and connects them by clicking and dragging the mouse from one component to other. It has a listener component at the starting of the flow, which is the ***input***, which gets the data from source system. On the other end of there is a transmitter component like ***Output*** and there are components in between, which process the data.

Design:



NiFi Summary

PROCESSORS | INPUT PORTS | OUTPUT PORTS | REMOTE PROCESS GROUPS | CONNECTIONS | PROCESS GROUPS

Displaying 3 of 3

	Name	Type	Process Group
ℹ	InvokeHTTP	InvokeHTTP	NiFi Flow
ℹ	JoltTransformJSON	JoltTransformJSON	NiFi Flow
ℹ	PublishKafka	PublishKafka	NiFi Flow

When we run the flow clicking on the Run Icon on the Invoke Http the following will perform the rest api call to other source system and then perform the transformation defined in JoltTransformJSON and then store the result in Kafka defined in publish Kafka.

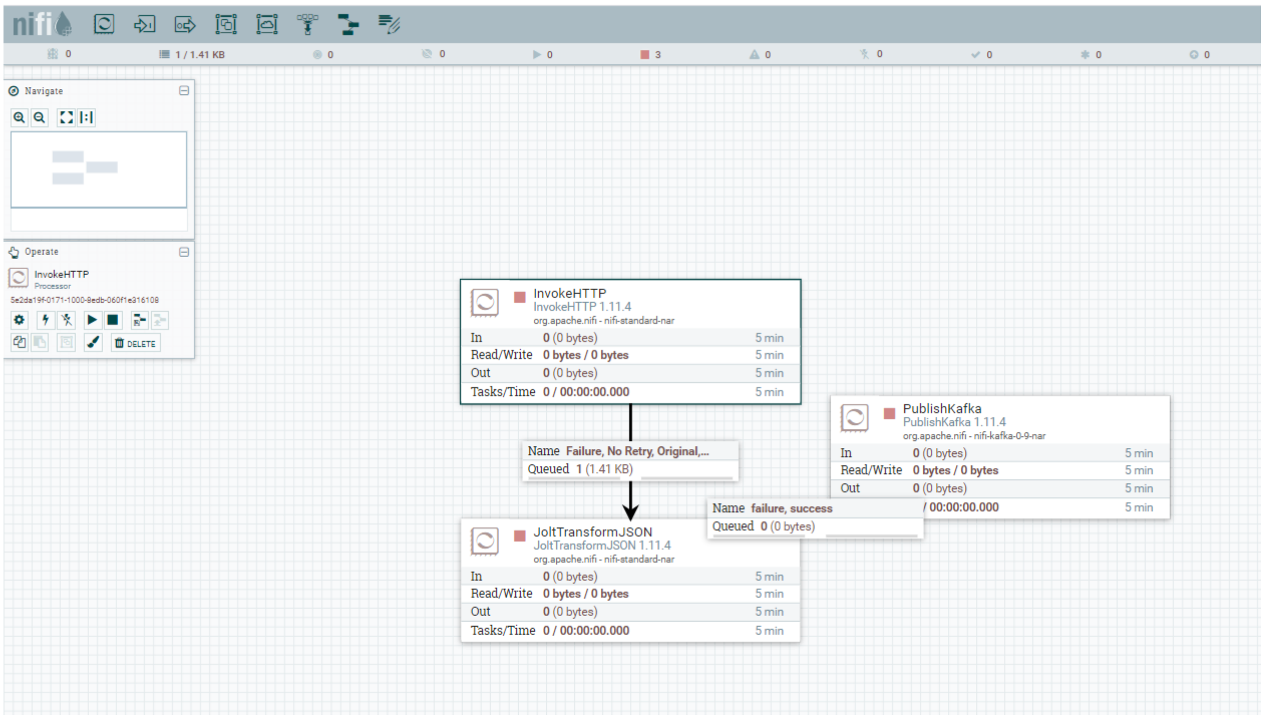
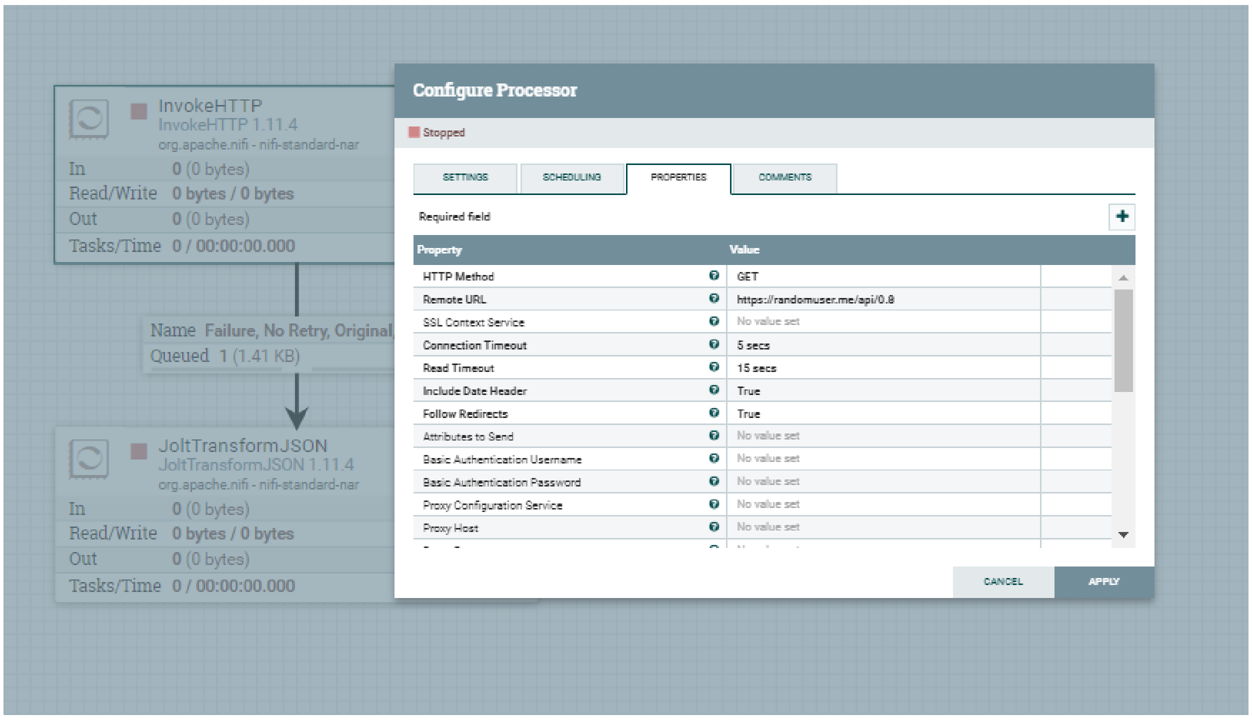
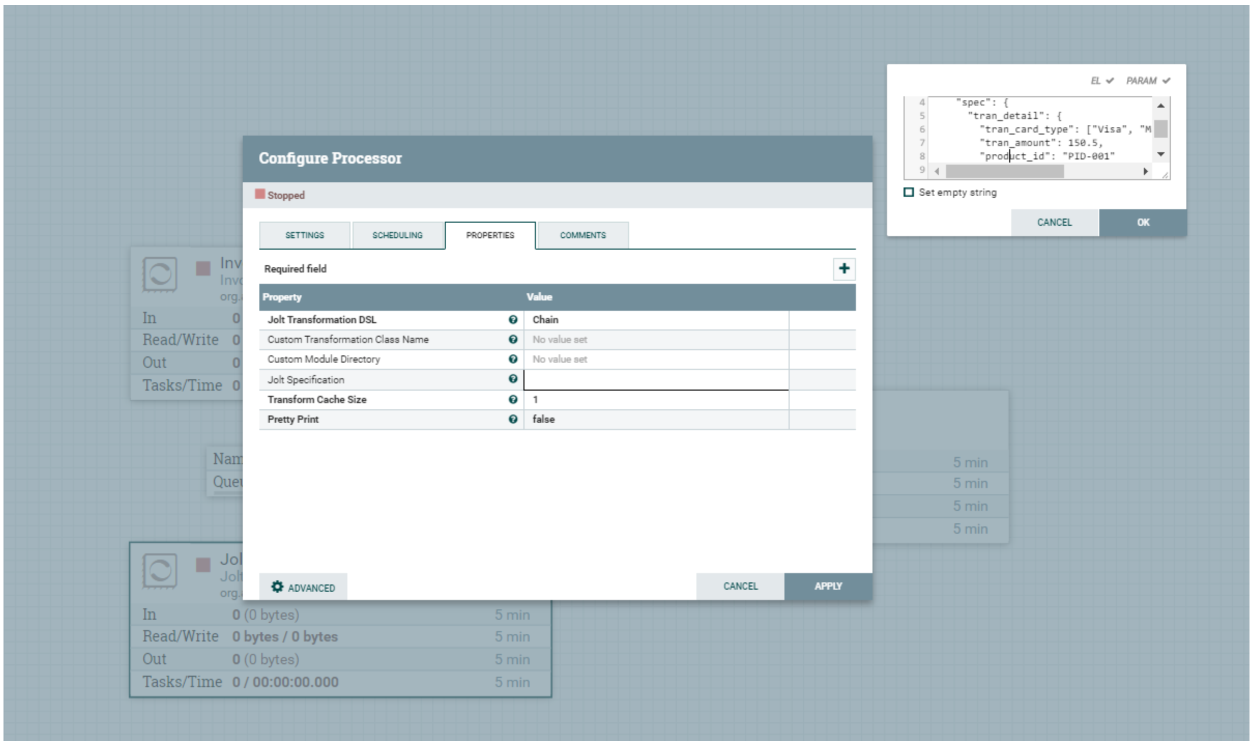


Figure. NIFI Flow Design



Property	Value
HTTP Method	GET
Remote URL	https://randomuser.me/api/0.8
SSL Context Service	No value set
Connection Timeout	5 secs
Read Timeout	15 secs
Include Date Header	True
Follow Redirects	True
Attributes to Send	No value set
Basic Authentication Username	No value set
Basic Authentication Password	No value set
Proxy Configuration Service	No value set
Proxy Host	No value set

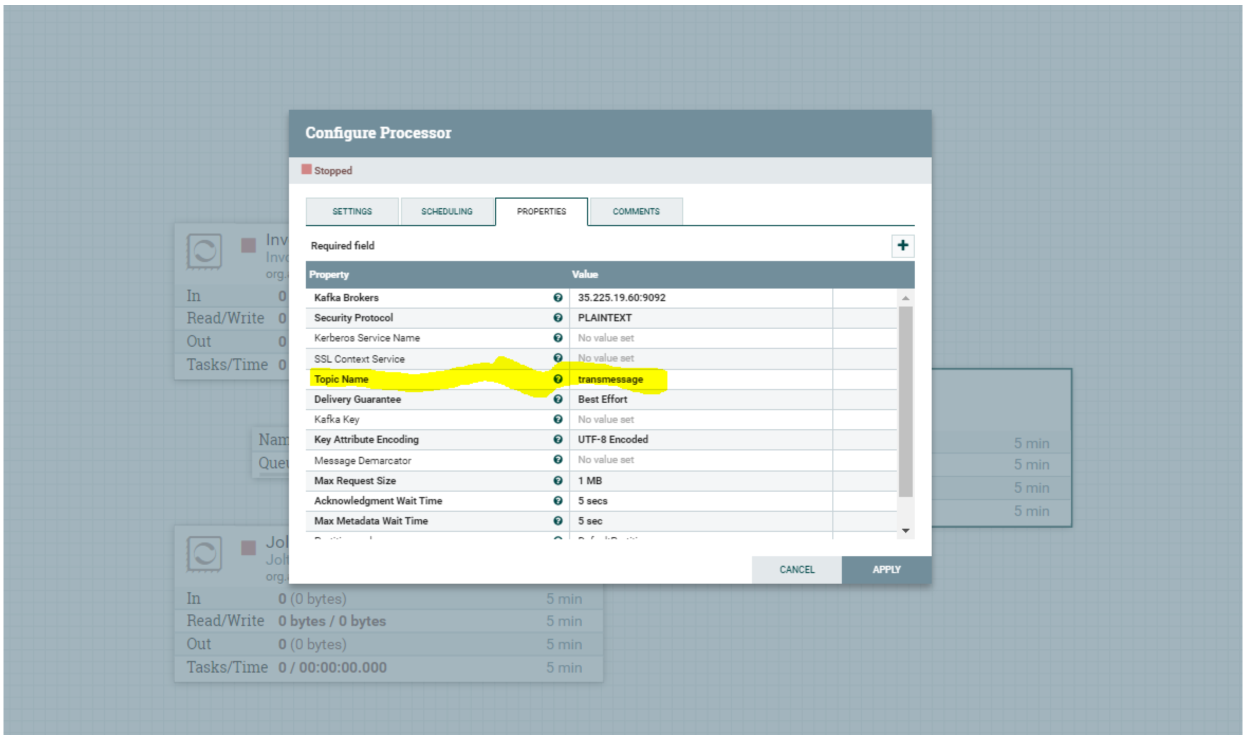
Figure. **Invoke HTTP Configuration**



Property	Value
Jolt Transformation DSL	Chain
Custom Transformation Class Name	No value set
Custom Module Directory	No value set
Jolt Specification	
Transform Cache Size	1
Pretty Print	false

```
  "spec": {
    "tran_detail": {
      "tran_card_type": ["Visa", "M
      "tran_amount": 150.5,
      "product_id": "PID-001"
```

Figure. **Transformation**



Configure Processor

Stopped

SETTINGS | SCHEDULING | PROPERTIES | COMMENTS

Required field

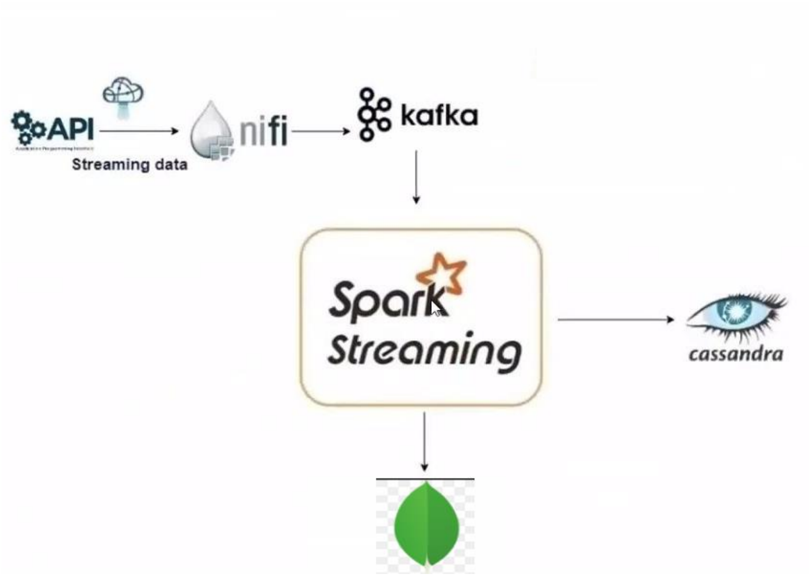
Property	Value
Kafka Brokers	35.225.19.60:9092
Security Protocol	PLAINTEXT
Kerberos Service Name	No value set
SSL Context Service	No value set
Topic Name	transmessage
Delivery Guarantee	Best Effort
Kafka Key	No value set
Key Attribute Encoding	UTF-8 Encoded
Message Demarcator	No value set
Max Request Size	1 MB
Acknowledgment Wait Time	5 secs
Max Metadata Wait Time	5 sec

CANCEL APPLY

Figure. **Publish to Kafka Topic**

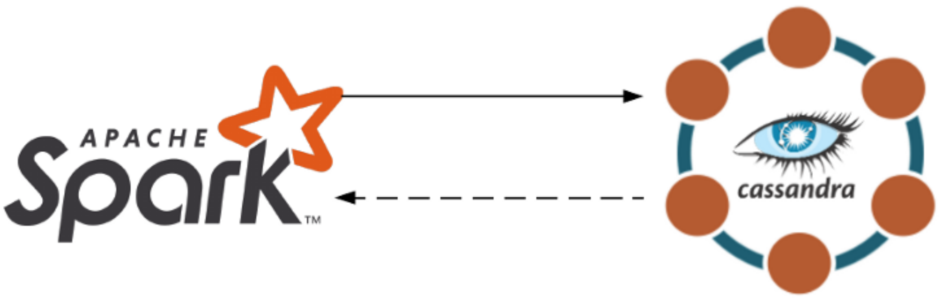
b) *Spark*

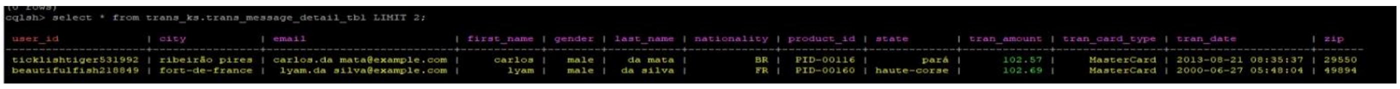
Once Nifi publishes the message to topic, Spark program reads from the consumer and performs data type check and stores it to No sql DBs as shown below



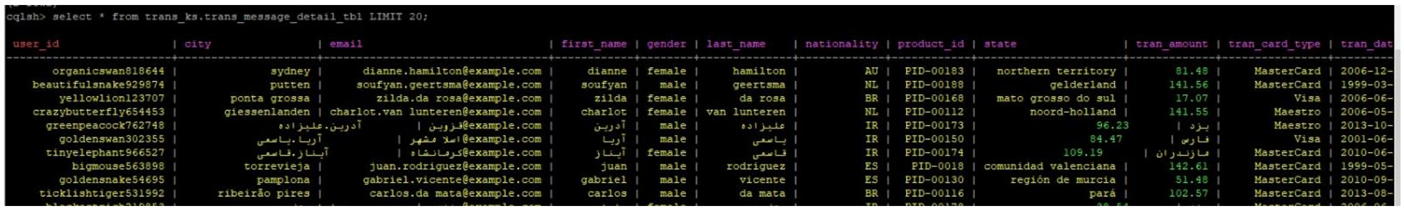
c) *Cassandra Database*

a. As per the Nifi flow the output we are pushing to Kafka which will be processed using RDDs in spark structured streaming and then pushed to created table in Cassandra. The schema of the table is shown below that includes user details such as user_id, city,email,first_name, last_name, nationality, product purchased, tran_amount, tran_card_type, tran_date, zip





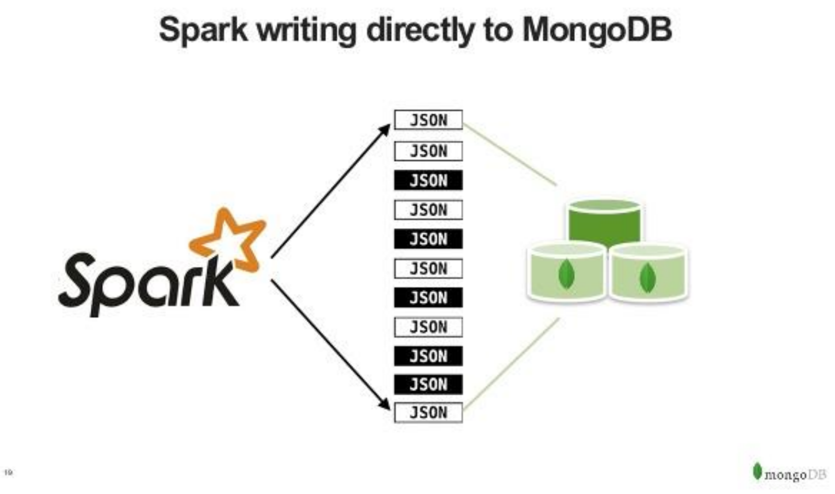
```
cqlsh> select * from trans_ks.trans_message_detail_tbl LIMIT 2;

 user_id               | city             | email                         | first_name | gender | last_name | nationality | product_id | state        | tran_amount | tran_card_type | tran_date           | zip
-----------------------+------------------+-------------------------------+------------+--------+-----------+-------------+------------+--------------+-------------+----------------+---------------------+-------
 ticklishtiger531992 | ribeirão pires | carlos.da mata@example.com |     carlos |   male |   da mata |          BR |  PID-00116 |         pará |      102.57 |     MasterCard | 2013-08-21 08:35:37 | 29550
 beautifulfish218849 | fort-de-france |  lyam.da silva@example.com |       lyam |   male |  da silva |          FR |  PID-00160 | haute-corse |      102.69 |     MasterCard | 2000-06-27 05:48:04 | 49894
```



```
cqlsh> select * from trans_ks.trans_message_detail_tbl LIMIT 20;

 user_id                   | city                  | email                                 | first_name | gender | last_name    | nationality | product_id | state                    | tran_amount | tran_card_type | tran_dat
---------------------------+-----------------------+---------------------------------------+------------+--------+--------------+-------------+------------+--------------------------+-------------+----------------+---------
       organicswan818644 |                sydney |     dianne.hamilton@example.com |     dianne | female |     hamilton |          AU |  PID-00183 |      northern territory |       81.48 |     MasterCard | 2006-12-
    beautifulsnake929874 |                putten |    soufyan.geertsma@example.com |    soufyan |   male |     geertsma |          NL |  PID-00188 |              gelderland |      141.56 |     MasterCard | 1999-03-
      yellowlion123707 |          ponta grossa |        zilda.da rosa@example.com |      zilda | female |      da rosa |          BR |  PID-00168 |     mato grosso do sul |       17.07 |           Visa | 2006-06-
  crazybutterfly654453 |         giessenlanden | charlot.van lunteren@example.com |    charlot | female | van lunteren |          NL |  PID-00112 |          noord-holland |      141.55 |        Maestro | 2006-05-
     greenpeacock762748 |           آدرین.علیزاده       | قزوین@example.com |      آدرین |   male |     علیزاده |          IR |  PID-00173 |                     96.23 | یزد |        Maestro | 2013-10-
      goldenswan302355 |            آریا.یاسمی          | اسلا مشهر@example.com |       آریا |   male |       یاسمی |          IR |  PID-00150 |                    84.47 | فارس |           Visa | 2001-06-
    tinyelephant966527 |           آیناز.قاسمی          | کرمانشاه@example.com |      آیناز | female |       قاسمی |          IR |  PID-00174 |               109.19 | مازندران |     MasterCard | 2010-06-
       bigmouse563898 |            torrevieja |     juan.rodriguez@example.com |       juan |   male |    rodriguez |          ES |   PID-0018 | comunidad valenciana |      142.61 |     MasterCard | 1999-05-
      goldensnake54695 |              pamplona |     gabriel.vicente@example.com |    gabriel |   male |      vicente |          ES |  PID-00130 |       región de murcia |       51.48 |     MasterCard | 2010-09-
     ticklishtiger531992 |        ribeirão pires |      carlos.da mata@example.com |     carlos |   male |      da mata |          BR |  PID-00116 |                   pará |      102.57 |     MasterCard | 2013-08-
```

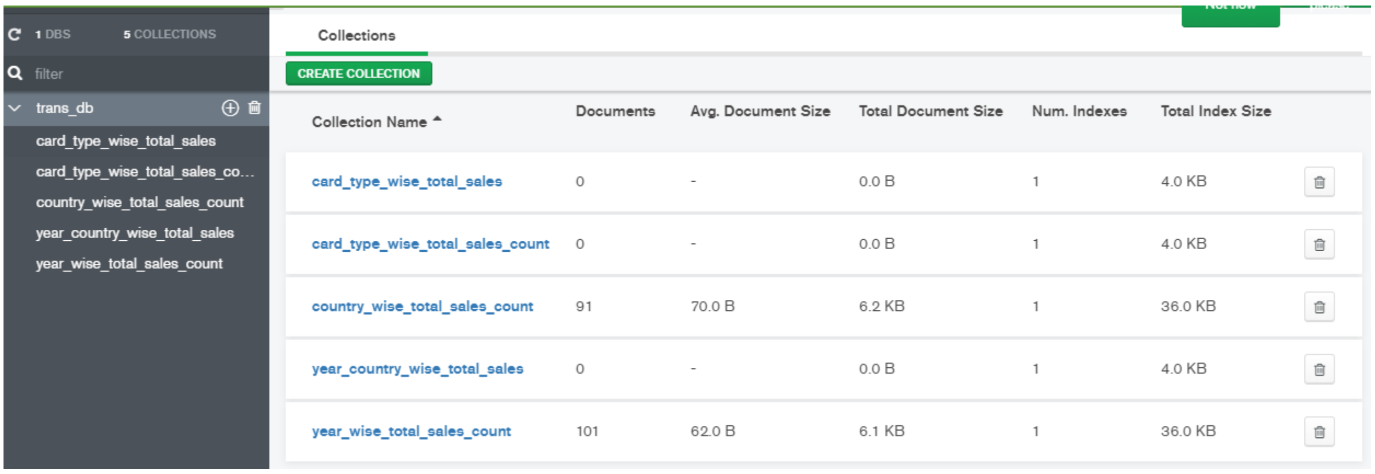
d) *Mongo Database*

b. We are using mongo database which will store the api result which is currently running every 30sec with the transformation job set in the Nifi flow for each row in the table.



We have used Mongo database to stores aggregated data post transformation. The Following aggregated data present in the Mongo database table is later used for generating the graph in real time. Transaction amount is aggregated by country and year and stored in mongodb collection. This will be stored for every batch of 200 incoming messages. The data is then aggregated to most recent 100 messages and displayed on ui

Mongo DB Aggregated Data Schema:



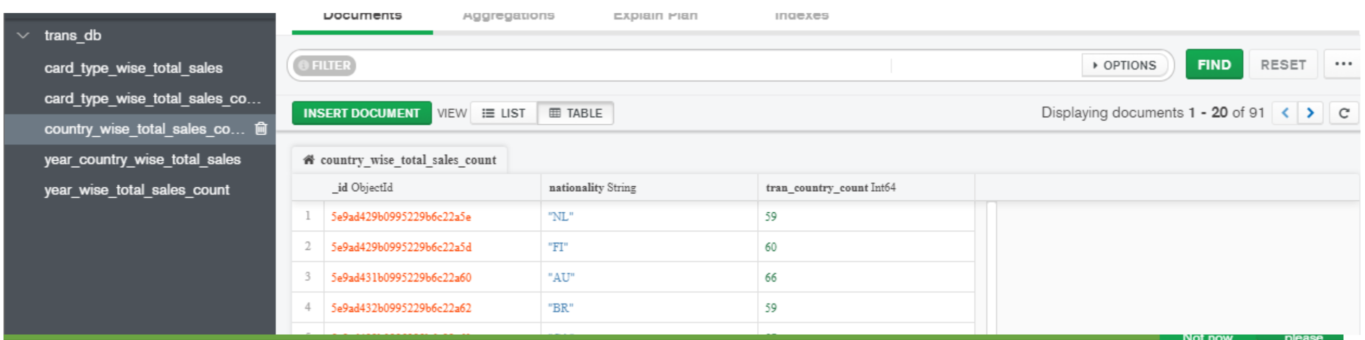
1 DBS 5 COLLECTIONS

- trans_db
  - card_type_wise_total_sales
  - card_type_wise_total_sales_co…
  - country_wise_total_sales_count
  - year_country_wise_total_sales
  - year_wise_total_sales_count

Collections

CREATE COLLECTION

Collection Name	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size
card_type_wise_total_sales	0	-	0.0 B	1	4.0 KB
card_type_wise_total_sales_count	0	-	0.0 B	1	4.0 KB
country_wise_total_sales_count	91	70.0 B	6.2 KB	1	36.0 KB
year_country_wise_total_sales	0	-	0.0 B	1	4.0 KB
year_wise_total_sales_count	101	62.0 B	6.1 KB	1	36.0 KB



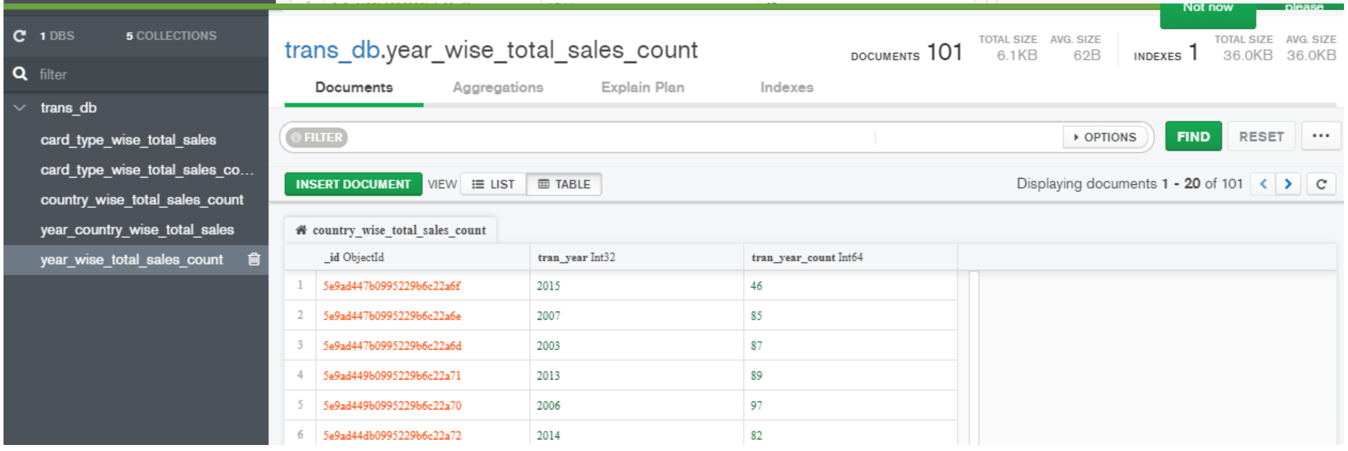
Documents Aggregations Explain Plan Indexes

INSERT DOCUMENT VIEW LIST TABLE

Displaying documents 1 - 20 of 91

country_wise_total_sales_count

	_id ObjectId	nationality String	tran_country_count Int64
1	5e9ad429b0995229b6c22a5e	"NL"	59
2	5e9ad429b0995229b6c22a5d	"FI"	60
3	5e9ad431b0995229b6c22a60	"AU"	66
4	5e9ad432b0995229b6c22a62	"BR"	59



trans_db.year_wise_total_sales_count

DOCUMENTS 101 TOTAL SIZE 6.1KB AVG. SIZE 62B INDEXES 1 TOTAL SIZE 36.0KB AVG. SIZE 36.0KB

Documents Aggregations Explain Plan Indexes

INSERT DOCUMENT VIEW LIST TABLE

Displaying documents 1 - 20 of 101

country_wise_total_sales_count

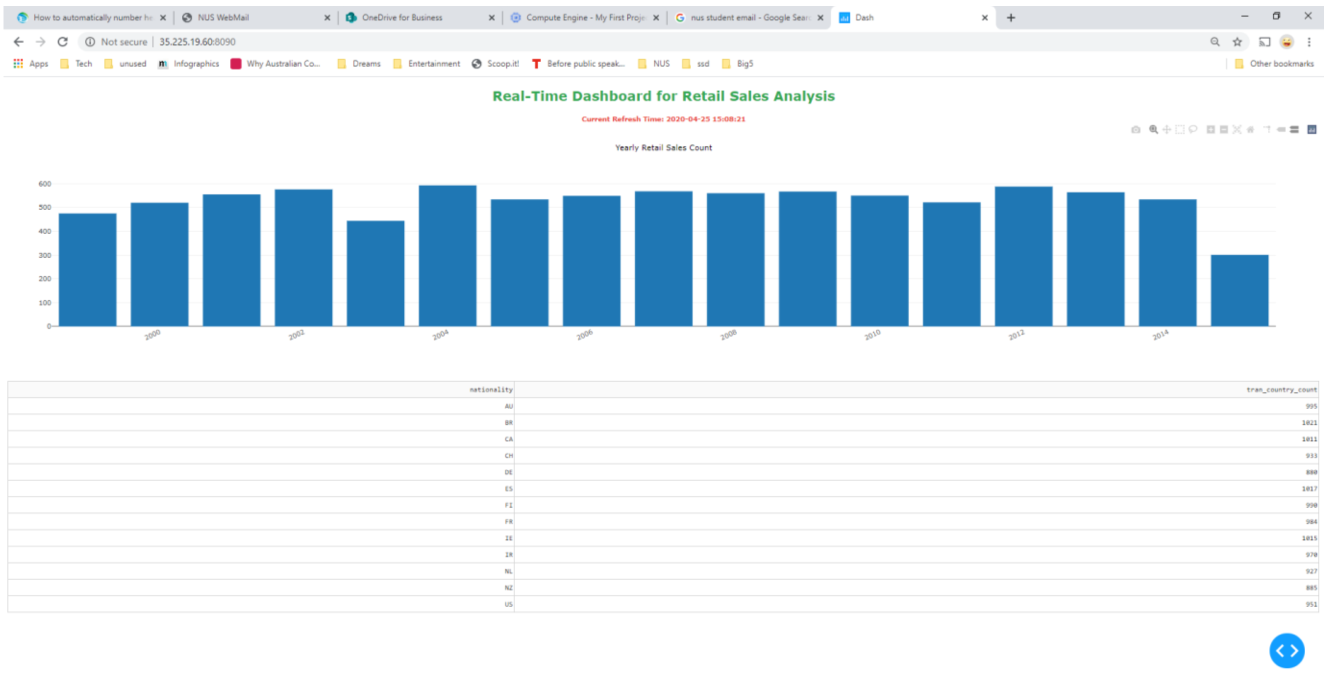
	_id ObjectId	tran_year Int32	tran_year_count Int64
1	5e9ad447b0995229b6c22a6f	2015	46
2	5e9ad447b0995229b6c22a6e	2007	85
3	5e9ad447b0995229b6c22a6d	2003	87
4	5e9ad449b0995229b6c22a71	2013	89
5	5e9ad449b0995229b6c22a70	2006	97
6	5e9ad44db0995229b6c22a72	2014	82

e) *Real Time Dashboard*

Aggregated sales data is displayed in the form of table and charts and updated in a Realtime

Algorithm for the change:

```
year_total_sales = db_object.year_wise_total_sales_count
   year_total_sales_df = pd.DataFrame(list(year_total_sales.find().sort([("_id",
pymongo.DESCENDING)]).limit(100)))
   df1 = year_total_sales_df.groupby(['tran_year'], as_index=False).agg({'tran_year_count':
"sum"})

   country_total_sales = db_object.country_wise_total_sales_count
   country_total_sales_df = pd.DataFrame(list(country_total_sales.find().sort([("_id",
pymongo.DESCENDING)]).limit(100)))
  df2 = country_total_sales_df.groupby(['nationality'],
as_index=False).agg({'tran_country_count': "sum"})
```


Not secure | 35.225.19.60:8090
Dash
Real-Time Dashboard for Retail Sales Analysis
Current Refresh Time: 2020-04-25 15:08:21
Yearly Retail Sales Count
0
100
200
300
400
500
600
2000
2002
2004
2006
2008
2010
2012
2014

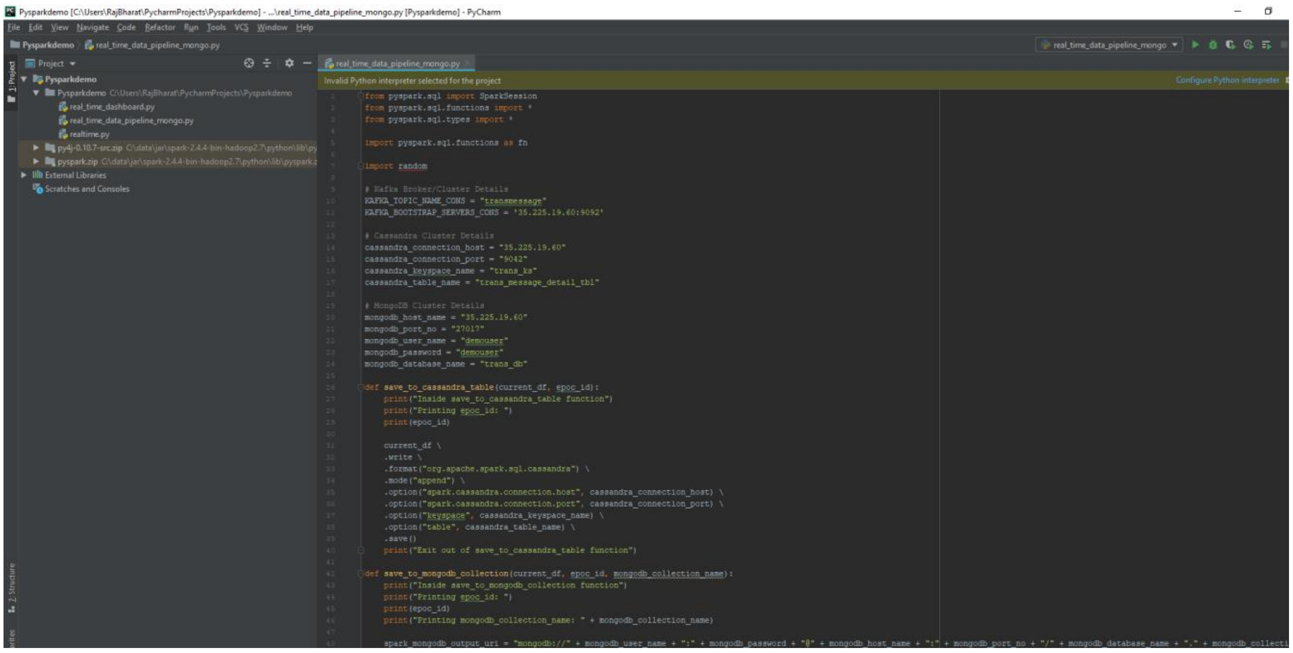
nationality	tran_country_count
AU	995
BR	1021
CA	1011
CH	933
DE	880
ES	1017
FI	990
FR	984
IE	1015
IR	970
NL	927
NZ	885
US	951

Fig. Dashboard: Yearly Sales count

10) Coding and Debugging

As the Server Instance is hosted on cloud, to write the programs and run – We must install the below software's locally and set up the necessary configuration

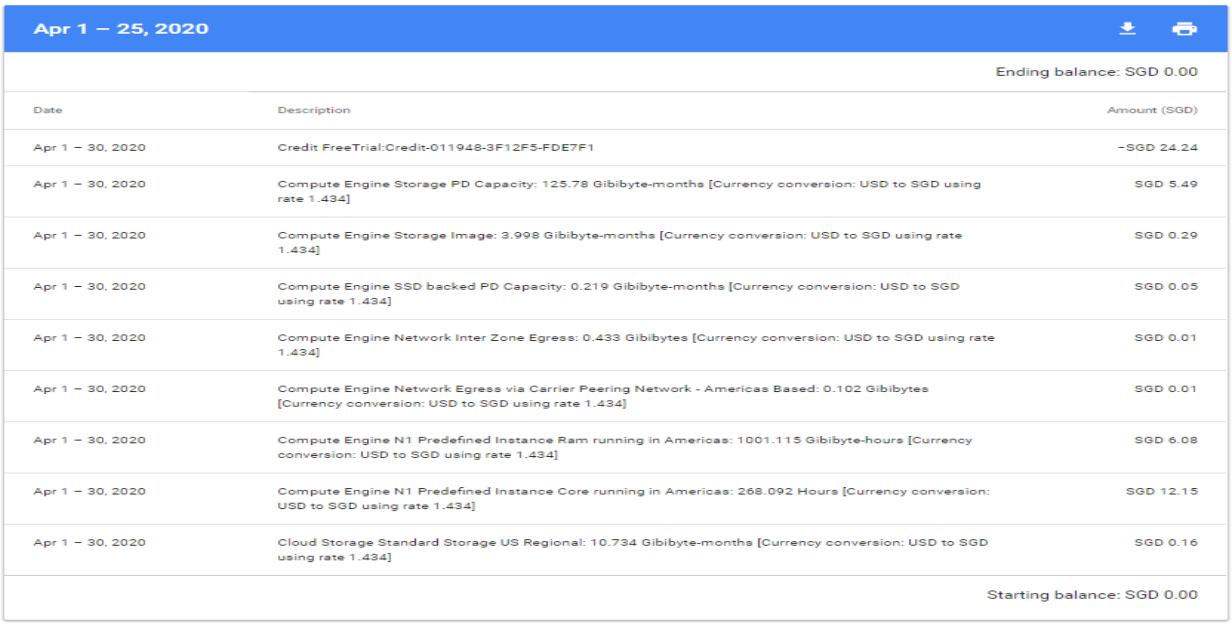
- ✓ Spark / Spark 2.4.1 with necessary proxy node settings
- ✓ Scala IDE / IntelliJ to run Scala Spark programs
- ✓ Python / PySpark using PyCharm IDE
- ✓ Necessary Python packages such as Dash and dependencies
- ✓ Compass for access to MongoDB
- ✓ Cassandra can be viewed from Command line only.



```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

import pyspark.sql.functions as fn

import random

# Kafka Broker/Cluster Details
KAFKA_TOPIC_NAME_CONS = "transmessage"
KAFKA_BOOTSTRAP_SERVERS_CONS = '35.225.19.60:9092'

# Cassandra Cluster Details
cassandra_connection_host = "35.225.19.60"
cassandra_connection_port = "9042"
cassandra_keyspace_name = "trans_ks"
cassandra_table_name = "trans_message_detail_tbl"

# MongoDB Cluster Details
mongodb_host_name = "35.225.19.60"
mongodb_port_no = "27017"
mongodb_user_name = "demouser"
mongodb_password = "demouser"
mongodb_database_name = "trans_db"

def save_to_cassandra_table(current_df, epoc_id):
    print("Inside save_to_cassandra_table function")
    print("Printing epoc_id: ")
    print(epoc_id)

    current_df \
    .write \
    .format("org.apache.spark.sql.cassandra") \
    .mode("append") \
    .option("spark.cassandra.connection.host", cassandra_connection_host) \
    .option("spark.cassandra.connection.port", cassandra_connection_port) \
    .option("keyspace", cassandra_keyspace_name) \
    .option("table", cassandra_table_name) \
    .save()
    print("Exit out of save_to_cassandra_table function")

def save_to_mongodb_collection(current_df, epoc_id, mongodb_collection_name):
    print("Inside save_to_mongodb_collection function")
    print("Printing epoc_id: ")
    print(epoc_id)
    print("Printing mongodb_collection_name: " + mongodb_collection_name)

    spark_mongodb_output_uri = "mongodb://" + mongodb_user_name + ":" + mongodb_password + "@" + mongodb_host_name + ":" + mongodb_port_no + "/" + mongodb_database_name + "." + mongodb_collecti
```

11) Cost Breakdown Table

Consumption was well within the limits. Including Data buckets, Computing – Utilization was less than 30 dollars



Apr 1 – 25, 2020

Ending balance: SGD 0.00

Date	Description	Amount (SGD)
Apr 1 – 30, 2020	Credit FreeTrial:Credit-011948-3F12F5-FDE7F1	-SGD 24.24
Apr 1 – 30, 2020	Compute Engine Storage PD Capacity: 125.78 Gibibyte-months [Currency conversion: USD to SGD using rate 1.434]	SGD 5.49
Apr 1 – 30, 2020	Compute Engine Storage Image: 3.998 Gibibyte-months [Currency conversion: USD to SGD using rate 1.434]	SGD 0.29
Apr 1 – 30, 2020	Compute Engine SSD backed PD Capacity: 0.219 Gibibyte-months [Currency conversion: USD to SGD using rate 1.434]	SGD 0.05
Apr 1 – 30, 2020	Compute Engine Network Inter Zone Egress: 0.433 Gibibytes [Currency conversion: USD to SGD using rate 1.434]	SGD 0.01
Apr 1 – 30, 2020	Compute Engine Network Egress via Carrier Peering Network - Americas Based: 0.102 Gibibytes [Currency conversion: USD to SGD using rate 1.434]	SGD 0.01
Apr 1 – 30, 2020	Compute Engine N1 Predefined Instance Ram running in Americas: 1001.115 Gibibyte-hours [Currency conversion: USD to SGD using rate 1.434]	SGD 6.08
Apr 1 – 30, 2020	Compute Engine N1 Predefined Instance Core running in Americas: 268.092 Hours [Currency conversion: USD to SGD using rate 1.434]	SGD 12.15
Apr 1 – 30, 2020	Cloud Storage Standard Storage US Regional: 10.734 Gibibyte-months [Currency conversion: USD to SGD using rate 1.434]	SGD 0.16

Starting balance: SGD 0.00

12) Final Words

We are able to leverage on the cloud infrastructure and read streaming real time api data, process the data and store the raw data in a columnar store for data reconciliation and persistence. To show the statistics of the purchase history by country and year – and show the trends on graph and table at a real time. This can be further analyzed looking for a specific reason.

13) Future Work

- Develop a scalable, resilient multi-node production scale design

- Apply Machine learning to predict real time figures for sales to aid planning and scheduling for goods
- We can explore to use managed ETL cloud services like Google cloud Dataflow which supports importing bounded (batch) raw data from sources such as relational Google Cloud SQL databases (MySQL or PostgreSQL, via the JDBC connector) and files in Google Cloud Storage for building big data pipelines.
- Wherever applicable, we will try to leverage on managed cloud services
    - Infrastructure will be managed by the cloud provider wherever possible
    - To allow us to concentrate on the business functionality and coding the business functions
    - Scalability and availability are typically taken care of when managed cloud services are used
- Trend seems to be moving towards a container-centric world:
    - We can explore Kubernetes with Spark jobs to see whether it is easier to deploy Spark jobs with such an architecture

14) Bibliography

- https://www.datanyze.com/market-share/databases--272/apache-cassandra-market-share
- https://www.computerworld.com/article/3412345/the-best-nosql-database-options-for-the-enterprise.html#slide2
- https://github.com/mapr-demos/finserv-application-blueprint
- https://towardsdatascience.com/best-5-free-stock-market-apis-in-2019-ad91dddec984
- https://docs.cloudera.com/documentation/director/latest/topics/director_gcp_setup.html
- If the design is interesting , Please drop a note to rajbharat.87@gmail.com or bharat.nagaraju@u.nus.edu for set up guide
- Demo url: https://youtu.be/bvUCOKwvamw