Real-time Data Streaming with Kafka and Spark

Submitted By:

Milan Bhandari

MS, Software Engineering

Department of Computing Sciences

Villanova University

Fall 2019

Advisor:

Ms. Kristin Obermyer

Abstract

Every day huge volumes of data are being generated from different data sources in different companies. The challenge is not just to process this data but to stream it fast and analyze it in real-time.

The main aim of this project is to implement a system with different tools in order to maintain a data pipeline that is highly scalable and fault-tolerant for a real-time data stream. We are going to use some currently popular tools like Kafka and Spark for streaming. Kafka is a distributed streaming platform and Spark is a unified analytics engine. We are going to use Twitter as data sources gathering tweets related to Villanova University. And we are going to use memSQL as our real-time database.

For data visualization, we can use Tableau or Metabase. Besides these, we are using Docker containers and Kubernetes for managing containers. The data from the data source is fed to a Kafka topic using Source Connector, and Spark is monitoring the Kafka topic as a consumer. We are using the pipeline to ingest data into memSQL.

We are using the best tools and practices that are known today. At the end of the project, we expect to get hands-on experience on these tools and technologies.

Acknowledgments

The success and final outcome of this project required a lot of guidance and assistance from many people and we are very fortunate to have got this all along with the completion of this project. We are very glad to express our deepest sense of gratitude and sincere thanks to our supervisor Ms.Kristin Obermyer for her valuable supervision, guidance, encouragement, and support for completing this work. Her useful suggestions for this whole work and co-operation are sincerely acknowledged.

We would not have been able to do a project at this scale if it were not for Villanova University and its esteemed Computing Sciences Department. We would also like to thank Dr. Vijay Gehlot and Dr. Daniel Joyce for their approval of this project and supports. We would like to thank Mr. Najib Nadi for providing us the resources required for this project.

In the end, we would like to express our sincere thanks to all our friends and others who helped us directly or indirectly during this project.

-Milan Bhandari

Table of Contents

Architectural Study

Motivation

The main motivation of this project is to implement a system that is capable of streaming data, transforming it, and analyzing it in real-time. We are going to use a data pipeline consisting of different middleware components and following different patterns like messaging patterns and publish-subscribe architecture.

System Design

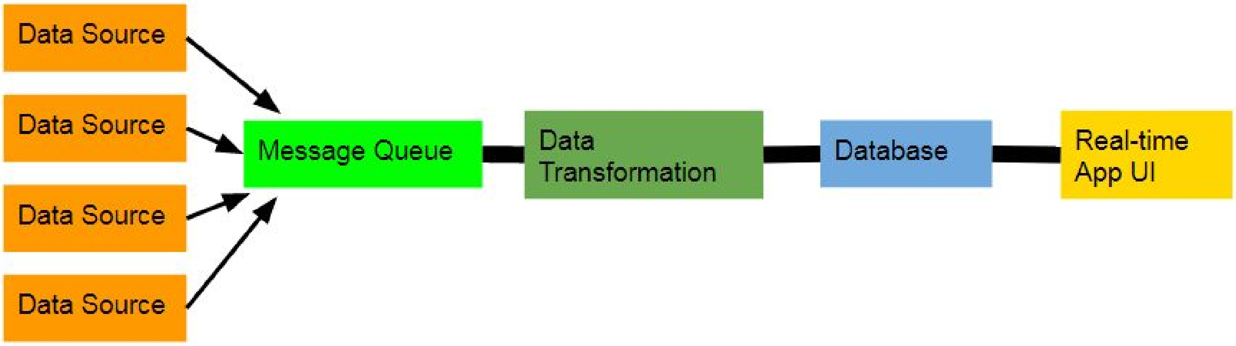


Image 1: System Architecture

Data Source

Data Sources are the producer of data for the streaming process. The data source can be anything like Twitter, stock data, etc. It can be a single source or multiple sources.

Message Queue

Queues are in FIFO - First In First Out order. In messaging, we can send a message i.e. put the message in a queue. We can receive a message i.e. get the message from the queue. The message queue is a temporary message storage. In message queue architecture, there are producers and consumers. Producers are the ones who create the messages and consumers are the ones who receive the messages. Message queue provides asynchronous communications decoupling producer from consumer.

Data Transformation

Data Transformation is the process of transforming our data from one form to another. Some of the transformation operations are like a map, filter, count, etc. It's a very important tool for analytics purposes.

Database

We store our data persistently in the database.

Real-time App

We can monitor real-time data using the UI of real-time applications. We can plot the real time data into graphs and by visualizing the graph, we can find different trends and patterns. Or, we can monitor the data in real time and notify users about any anomalies.

Technology Stack

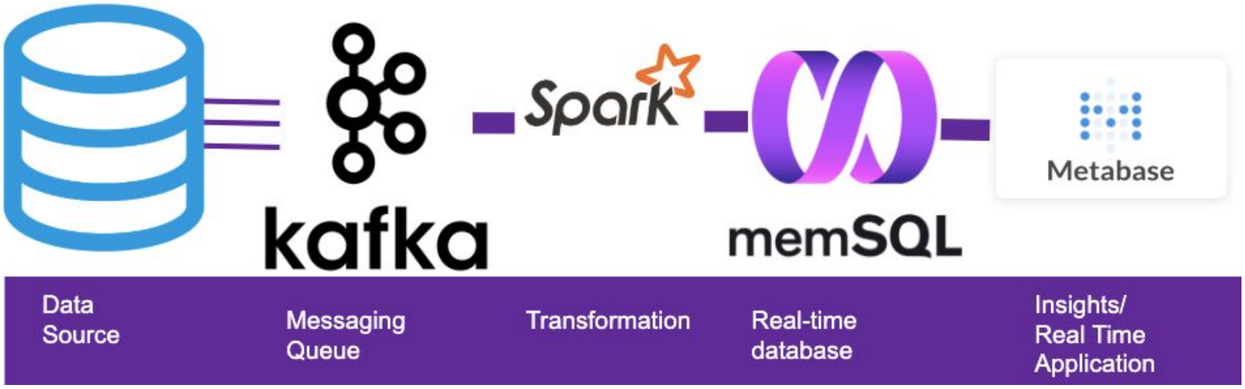


Image 2: Technology Stack

These are the main technologies used in our project:

1. Twitter API as our Data Source
2. Kafka
3. Spark
4. memSQL
5. Metabase

Due to some limitations of Tableau, we later used Metabase instead of Tableau which we will discuss in detail later.

We have used Docker for containerization. We have also used Portainer to manage these Docker containers. In programming languages we have used Java and Python. Besides these, we have used SQL for querying database.

Each of these parts of the pipeline will be explored in more detail in the following sections.

Kafka

According to Confluent, one of the platform providers for Apache Kafka:

“Apache Kafka is a community distributed streaming platform capable of handling trillions of events a day. Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log.” [1]

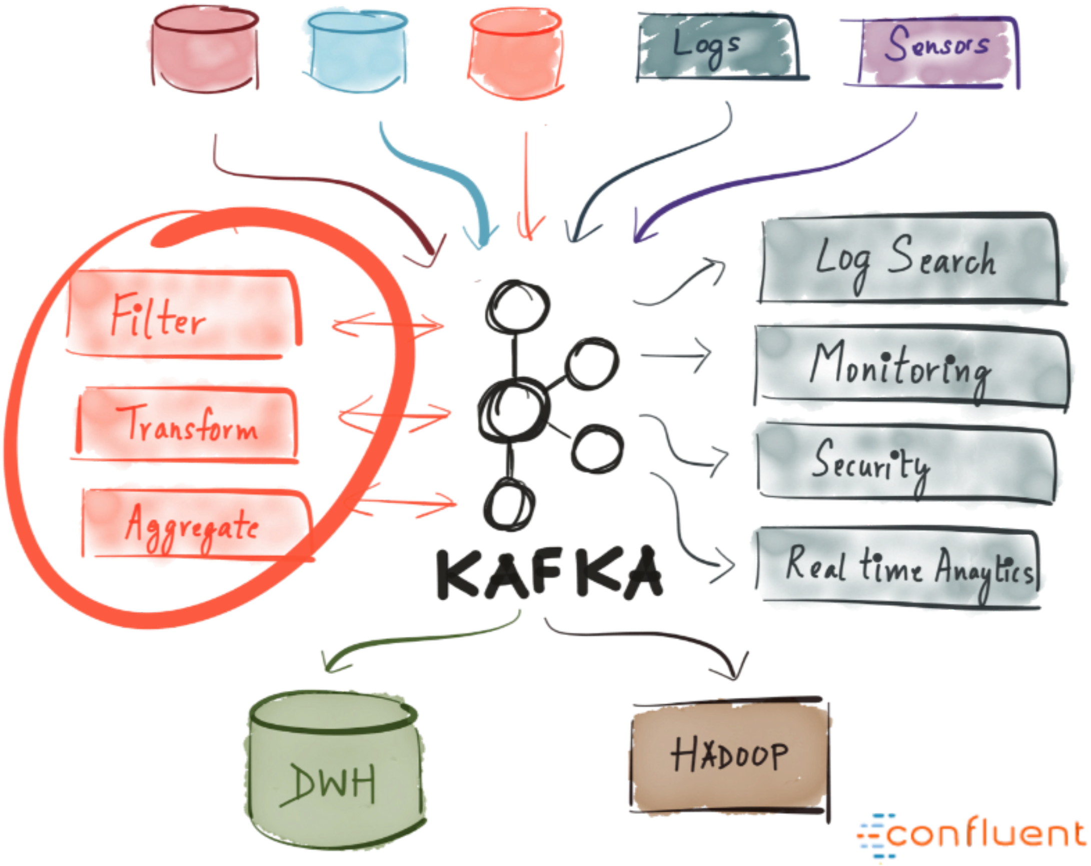


Image 3: Kafka Use Cases [1]

Kafka is a stream-processing software platform written in Scala and Java used for building real-time data pipelines and streaming apps. It is an open-source, fast, scalable, durable, and fault-tolerant publish-subscribe messaging system and can be run as a cluster on one or more servers in a distributed manner. All Kafka messages are stored in topics.

A topic is a category to which messages are stored and published. These messages are read by the Kafka Consumers from the topics. Kafka Producers are those who write messages to the topic. In our API, Spark and memSQL are examples of Kafka Consumers and Twitter API is our Kafka Producer.

Brokers are the nodes in the cluster as Kafka runs in a cluster in a distributed system. Each broker can have single or multiple partitions and each of these partitions can be either a leader or a replica for a topic. We read or write messages from the leader only and replicas are updated in accordance with the leader. If the leader node is not working then another leader is selected from the replicas. This election of the leader node is performed by ZooKeeper.

ZooKeeper is responsible to keep track of the status of the Kafka cluster nodes, Kafka topics, partitions, etc. We must run Zookeeper in order to run Kafka. The data within ZooKeeper is divided across multiple collections of nodes and this is how it achieves its high availability and consistency.[2] It is also responsible for the configuration of topics, including the list of existing topics, number of partitions, location of all replicas, the leader node, etc. ZooKeeper also maintains a list of all the brokers that are functioning at any given moment and are a part of the cluster.

Kafka is a distributed publish-subscribe messaging system. It allows Producers to persist their data in real-time to Kafka topic. In our case, whenever the Twitter API gets the new tweets, it's stored in our Kafka topic as well using Kafka Connector in real-time. Multiple brokers of Kafka can be installed as per the requirement. To work effectively, it's suggested to use at least three to five brokers. So, if any broker is down, the system will still run effectively and efficiently. Hence, it's fault-tolerant as well. Also, we can scale Kafka as per our requirement.

Spark

Spark is a distributed, data processing engine for batch and streaming modes. Spark supports Java, Python, R, and Scala. It has different libraries like SQL, machine learning, graph computation, and stream processing. Spark powers applications to rapidly query, analyze the data and transform data at scale. [3]

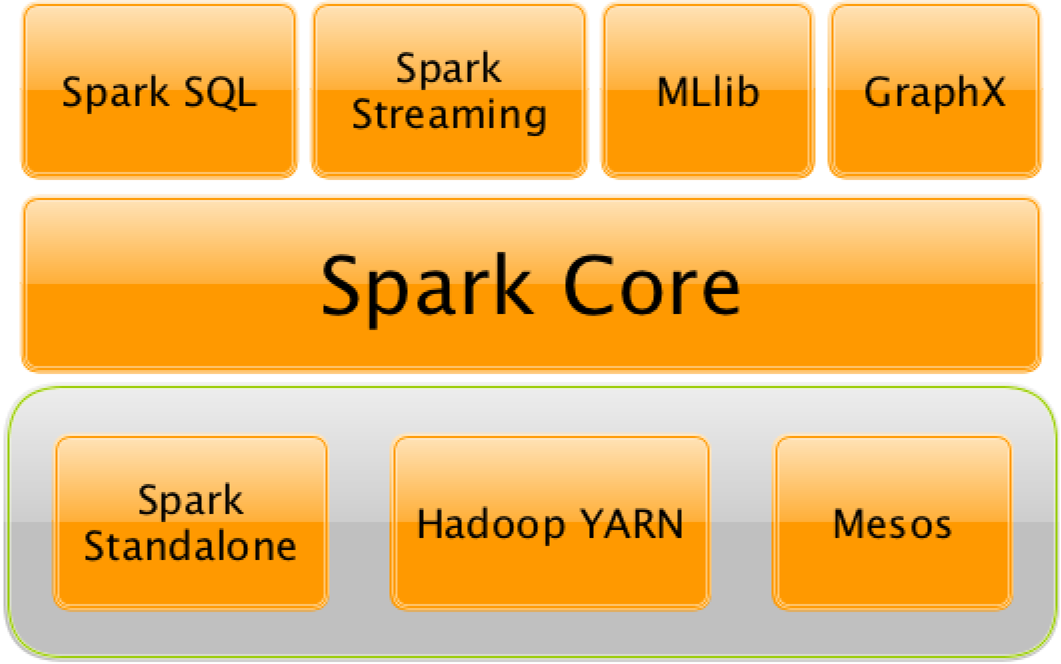


Image 4: Spark Architecture [3]

With Spark, we can access any data type from any data source. It's Structured Streaming and SQL programming models with MLlib and GraphX make it easier for developers and data scientists to build applications that exploit machine learning and graph analytics.

Use cases of spark: [4]

- Data integration and ETL
- Interactive analytics or business intelligence
- High-performance batch computation
- Machine learning and advanced analytics
- Real-time stream processing

MemSQL

MemSQL is a distributed, relational database that handles both transactions and real-time analytics at scale. Querying is done through standard SQL drivers and syntax, leveraging a broad ecosystem of drivers and applications. [5]

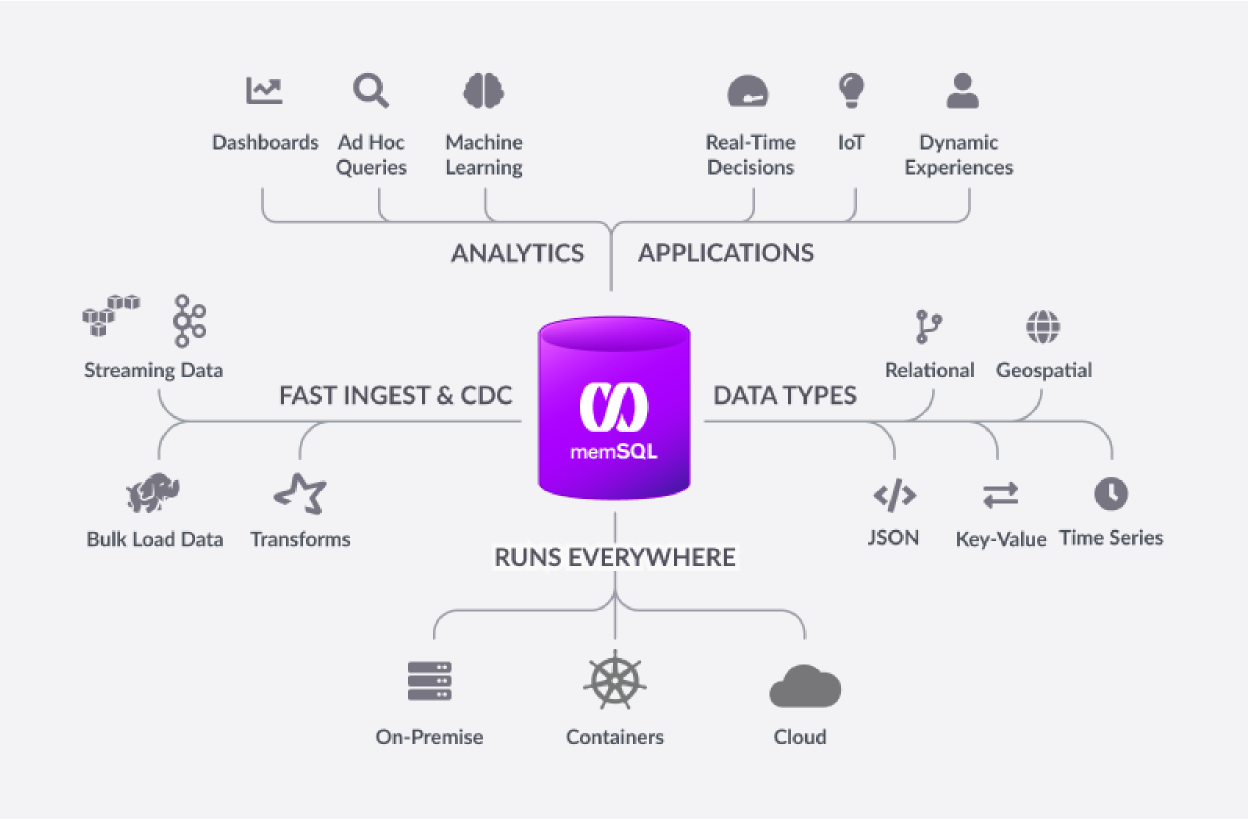


Image 5: MemSQL Use Cases [5]

We can easily integrate memSQL with Kafka, Spark, and BI (Business Intelligence) tools. It has an in-memory rowstore and on-disk columnstore which makes it suitable for both analytical and concurrent operation. It is also capable of ingesting a huge amount of streaming data from a source like Kafka with its Pipeline. Besides this, we can also transform the data before storing it into the database. MemSQL is able to store different data types along with JSON and geospatial data.

The memSQL database supports scalability and real-time analytics. MemSQL can consist of different nodes. Each node can be either an Aggregator node or a Leaf node. Aggregator node is responsible for querying leaf nodes, aggregating the results, and displaying results back to the clients. It also stores the metadata like indexes. A leaf node is a storage node. Each leaf node contains a portion of a large table. The workloads are divided between the Aggregators.

Tableau

Tableau is a powerful and fast-growing data visualization tool used in the BI industry. It helps in simplifying raw data into a very easily understandable format. [6]
We were using an Ubuntu machine for our project. Tableau Desktop has no installation file for Ubuntu. They can only be installed on a Windows machine or a Mac machine. We tried to use Docker but it didn't work. We tried to install Tableau server which is available in Ubuntu as well. But our device specification was not enough to install Tableau Server.

So, instead of Tableau, we decided to use another open-source tool called Metabase.

Metabase

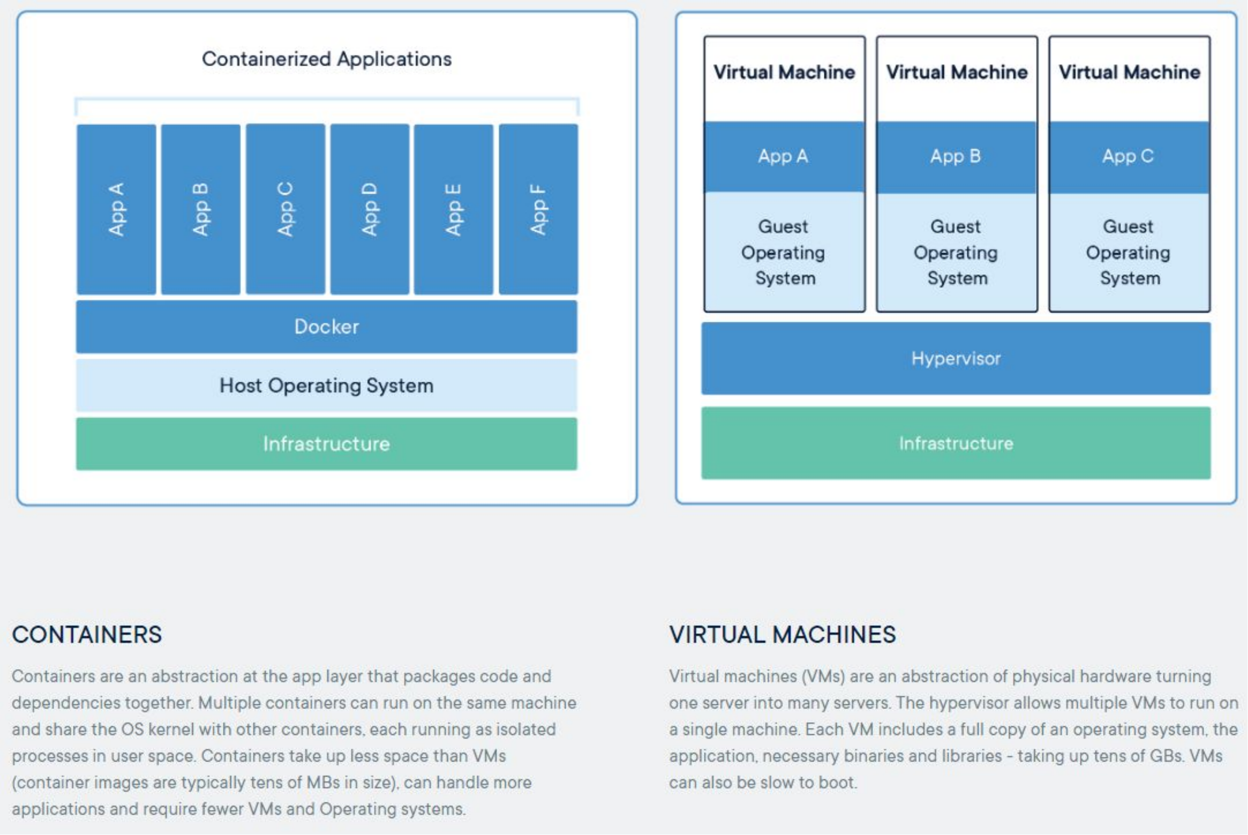
The main advantage of the Metabase is that it's open source. It is one of the most powerful BI tools. In Metabase, we can visualize the data, do the analytics, browse data, compare data, run queries, and make our own dashboard with the required information.

Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.[7] Containers are used to package up our application with all libraries and other dependencies so that we can ship our application as one package.

The purpose of Docker in our project is to make our applications' setup easier, flexible and portable. We can test and deploy our code easily. Also, we don't have to worry about which machine/operating system we are using.

Docker works like a virtual machine (VM) but we don't have to set up a whole virtual machine, instead only those individual components which the application needs in order to operate. That's why Docker is faster than VMs.



CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Image 6: Containers V/s VM

Source: https://www.docker.com/resources/what-container#/package_software

System Components

The following diagram shows the main components of the project and how they communicate with each other.

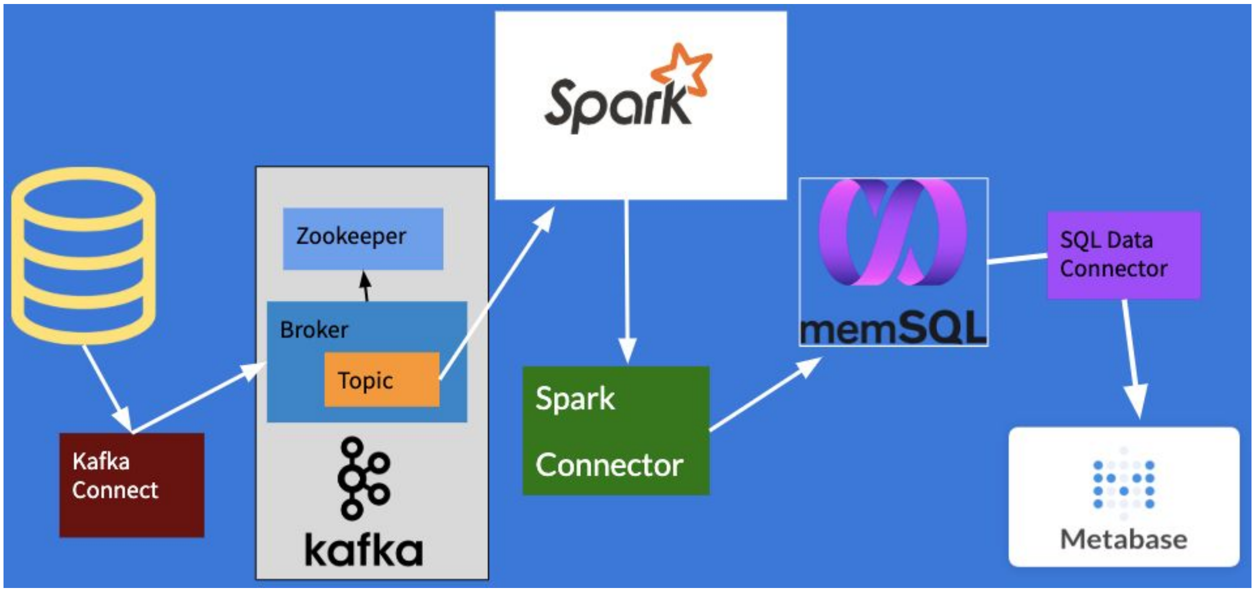


Image 7: System Components Diagram

We can connect the data source to the Kafka topic using Kafka Connect. There are different connectors available in the market. Among different connector providers, Confluent is one of the popular ones. One can go to their site and check for the connectors they need.

https://docs.confluent.io/current/connect/index.html

https://www.confluent.io/connectors/

Once we publish data to Kafka's topic, we can use Apache's Spark form data transformation and analysis purpose. The data obtained from that will be saved to the memSQL database. To connect the memSQL database with Spark, the Spark Connector is used.

https://docs.memsql.com/memsql-and-spark/v6.8/spark-connector/

We can ingest data from the database to Metabase by simply using their connector and visualize the data.

Implementation And Configuration

For the experiment, we need to stream the data from Twitter. We have to set up the environment in a way that different APIs and modules can communicate with each other. More details about the implementation and configuration of the system is explained below.

Docker Setup

For Ubuntu, follow these instructions:

https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04

For macOS:

https://docs.docker.com/docker-for-mac/install/

Kafka Setup

For Kafka installation on docker:

https://docs.confluent.io/current/quickstart/cos-docker-quickstart.html

Clone the git repo and go to directory: examples/cp-all-in-one and modify the docker-compose.yml file.

```
 connect:
    image: cnfldemos/kafka-connect-datagen:0.1.3-5.3.1
    hostname: connect
    container_name: connect
    depends_on:
      - zookeeper
      - broker
      - schema-registry
    ports:
      - "8083:8083"
    environment:
      CONNECT_BOOTSTRAP_SERVERS: 'broker:29092'
      CONNECT_REST_ADVERTISED_HOST_NAME: connect
      CONNECT_REST_PORT: 8083
      CONNECT_GROUP_ID: compose-connect-group
      CONNECT_CONFIG_STORAGE_TOPIC: docker-connect-configs
      CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_OFFSET_FLUSH_INTERVAL_MS: 10000
      CONNECT_OFFSET_STORAGE_TOPIC: docker-connect-offsets
      CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_STATUS_STORAGE_TOPIC: docker-connect-status
      CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_KEY_CONVERTER: org.apache.kafka.connect.storage.StringConverter
      CONNECT_VALUE_CONVERTER: io.confluent.connect.avro.AvroConverter
      CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL: http://schema-registry:8081
      CONNECT_INTERNAL_KEY_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
      CONNECT_INTERNAL_VALUE_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
      CONNECT_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      # CLASSPATH required due to CC-2422
      CLASSPATH:
/usr/share/java/monitoring-interceptors/monitoring-interceptors-5.3.1.jar
      CONNECT_PRODUCER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"
      CONNECT_CONSUMER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"
      CONNECT_PLUGIN_PATH:
"/usr/share/java,/usr/share/confluent-hub-components"
      CONNECT_LOG4J_LOGGERS:
org.apache.zookeeper=ERROR,org.I0Itec.zkclient=ERROR,org.reflections=ERROR
    volumes:
      -
/home/milan/Documents/DataStreamingWithKafkaAndSpark/jars:/etc/kafka-connect/ja
rs
```

The volumes are the path to the jar files related to the connector. We have to map the disk path (original) to the path of the container.

Also, make sure the docker-compose is installed to run these commands.

```
https://linuxize.com/post/how-to-install-and-use-docker-compose-on-ubuntu-18-04
/
```

Go to the directory where your docker-compose.yml file is and run the command:

```
docker-compose up -d --build
```

This will create different containers for Kafka broker, ZooKeeper, Connect, KSQL, Control Center, etc. We can set up the Portainer to view and manage these containers.

```
docker run -d -p 9000:9000 --restart always -v
/var/run/docker.sock:/var/run/docker.sock -v /data:/data --name my-portainer
portainer/portainer
```

Kafka Connect Setup

To access Twitter data, we first need to generate APIs keys. Apply via this link https://apps.twitter.com for developer access, and follow the instructions provided.

Next, download the necessary Connector:

https://www.confluent.io/hub/jcustenborder/kafka-connect-twitter

Extract the folder, the lib folder inside contains all the jars files we need. Copy that file and paste it in the location which is mapped to the container while installing Kafka Connect. In our case it's volumes: -
/home/milan/Documents/DataStreamingWithKafkaAndSpark/jars:/etc/kafka-connect/jars

Create a topic named 'twitter'. We can go to the control center http://0.0.0.0:9021 and create one from there easily.

Create a JSON file and paste this:

```
{
 "name": "twitter_source_connector",
 "config": {
   "connector.class": "com.github.jcustenborder.kafka.connect.twitter.TwitterSourceConnector",
   "twitter.oauth.accessToken": "YOUR_TOKEN",
   "twitter.oauth.consumerSecret": "SECRET_KEY",
   "twitter.oauth.consumerKey": "CONSUMER_KEY",
   "twitter.oauth.accessTokenSecret": "ACCESS_TOKEN_SECRET",
   "kafka.delete.topic": "twitter_delete",
   "value.converter": "org.apache.kafka.connect.json.JsonConverter",
   "key.converter": "org.apache.kafka.connect.json.JsonConverter",
   "value.converter.schemas.enable": false,
   "key.converter.schemas.enable": false,
   "kafka.status.topic": "twitter",
   "process.deletes": true,
   "filter.keywords": "villanova university, #NovaNation, #GoingNova, VUadmission, vucareercenter, VU_eSports, IgniteChangeGoNova, NovaAthletics, Villanova_Alum"
 }
}
```

Save this file as twitterSourceConnector.json. We have to enter our keys here that we obtained from the developer site of Twitter. Also, we can add filter keywords in json file as an attribute. We have added some hashtags related to Villanova University tweets as shown in the last attribute in the above JSON.

Go to command line and enter:

```
curl -d @twitterSourceConnector.json -H "Content-Type: application/json" -X POST http://localhost:8083/connectors
```

Replace @twitterSourceConnector.json with the directory which contains that file.

MemSQL Setup

For the installation of memSQL, we can follow this link:

https://docs.memsql.com/v6.8/guides/deploy-memsql/self-managed/memsql-tools/single-host/docker/step-1/

```
docker run -i --init \
    --name memsql-ciab \
    -e LICENSE_KEY=$LICENSE_KEY \
    -p 3306:3306 -p 8080:8080 \
    -v memsql:/memsql \
    memsql/cluster-in-a-box
```

Spark Project

For installation in the host machine:

Follow the following link:

https://spark.apache.org/docs/1.2.2/spark-standalone.html

https://ronnieroller.com/spark/kafka-message-processor

While running the sbin/start-slaves.sh command, if you get an error like this:

```
localhost: ssh: connect to host localhost port 22: Connection refused
```

Go through this link to troubleshoot:
https://stackoverflow.com/questions/17335728/connect-to-host-localhost-port-22-connection-refused

Remove SSH with the following command:

```
sudo apt-get remove openssh-client openssh-server
```

Install SSH again with:

```
sudo apt-get install openssh-client openssh-server
```

It should work now.

Maven Project

We created a new Java Maven project using Eclipse.

We have to add dependencies in the pom.xml file. At first, make sure that there is no conflict in the version of Spark, Scala, and Java. Use JDK 8 and add the following to the pom.xml file:

```
<dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.12</artifactId>
            <version>2.4.4</version>
</dependency>
<dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.12</artifactId>
            <version>2.4.4</version>
</dependency>
<dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
            <version>2.4.4</version>
</dependency>
<dependency>
            <groupId>org.apache.spark</groupId>
```

```
                <artifactId>spark-sql_2.12</artifactId>

                <version>2.4.4</version>

        </dependency>

        <dependency>

                <groupId>org.apache.spark</groupId>

                <artifactId>spark-sql-kafka-0-10_2.12</artifactId>

                <version>2.4.4</version>

        </dependency>
```

Metabase Setup

To install Metabase using Docker:

```
$ docker run -d -p 3000:3000 --name metabase metabase/metabase
```

Documentation can be found in the link below:

https://www.metabase.com/docs/latest/developers-guide.html

Data Stream from Kafka to Spark to memSQL

The detailed code is included in Appendix II.

We have to configure Spark and connect it to the Kafka server.

Spark Configuration

```
package com.grandChallenge.project.kafkaspark.config;
import org.apache.spark.SparkConf;

public class SparkConfig {
      public SparkConf sparkConf(String sparkName) {
        return new SparkConf()
                         .setAppName(sparkName)
                         .setMaster("local[*]");
    }
}
```

Kafka Configuration

```
Map<String, Object> kafkaParams = new HashMap<>();
            kafkaParams.put("bootstrap.servers", "172.17.0.3:9092");
            kafkaParams.put("key.deserializer", StringDeserializer.class);
            kafkaParams.put("value.deserializer", StringDeserializer.class);
            kafkaParams.put("group.id",
"use_a_separate_group_id_for_each_stream");
            kafkaParams.put("auto.offset.reset", "latest");
            kafkaParams.put("enable.auto.commit", false);
```

Connecting Spark and Kafka

```
JavaInputDStream<ConsumerRecord<String, String>> stream =
KafkaUtils.createDirectStream(sc,
                        LocationStrategies.PreferConsistent(),
                        ConsumerStrategies.<String, String>Subscribe(topics,
kafkaParams));
```

Connecting Spark and memSQL

```
public class DatabaseConfig {
      public static final String JDBC_DRIVER = "org.mariadb.jdbc.Driver";
      public static final String DB_URL =
"jdbc:mariadb://172.17.0.5:3306/db_twitter";
      public static final String dbUserName = "root";
      public static final String dbUserPassword = "";
      public static final String tblName = "tb_tweets";
}
```

We need to import the following:

```
import java.io.Serializable;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
//code to connect database
Class.forName(DatabaseConfig.JDBC_DRIVER);

conn = DriverManager.getConnection(DatabaseConfig.DB_URL,
DatabaseConfig.dbUserName,
DatabaseConfig.dbUserPassword);
stmt = conn.createStatement();
LocalDateTime dt = LocalDateTime.now();
sorted.forEach(record -> {
            id = generateUniqueId();
            String sql = "Insert into tb_popularTweets(id, Tweet, Count, dt)
values (" + id + ", \"" + record._2 + "\", " + record._1 + ", \"" +
dtf.format(dt) +"\")";

            try {
                  stmt.executeUpdate(sql);
            } catch (SQLException e) {
                                    e.printStackTrace();
            }
```

Data Stream from Kafka to memSQL using Pipeline

We need to create a database and table in memSQL. We can use the SQL Editor of memSQL going on this link (http://0.0.0.0:8080).

```
DROP DATABASE IF EXISTS db_twitter;
CREATE DATABASE db_twitter;
USE db_twitter;
-- A table set up to receive raw data from Kafka.
CREATE TABLE tb_tweets(
    `Id` bigint(20) NOT NULL,
    `body` text,
    `retweet_count` int,
    `created` datetime NOT NULL,
    KEY(Id) USING CLUSTERED COLUMNSTORE,
    SHARD KEY(Id)
);
```

To create a pipeline to stream data from Kafka:

```
CREATE PIPELINE twitter_pipeline
AS LOAD DATA KAFKA "172.17.0.4:29092/twitter"
WITH TRANSFORM ('http://172.19.0.1:12345/transform.py', '','')
SKIP ALL ERRORS
INTO TABLE `tb_tweets`
(Id, body, retweet_count, created);
```

To start the pipeline:

```
START PIPELINE twitter_pipeline;
```

To list any errors in the pipeline:

```
Select ERROR_MESSAGE from information_schema.pipelines_errors
```

Data Transformation with Pipeline

We can transform the data according to our requirements before storing into the database. We might have data in JSON that we don't need to save. We can select the data from JSON and only save those attributes in the database according to our requirements. In our project, from the tweets of JSON type, we have selected Id, Text, Retweet Count only. The sample code is below: The transform.py code modified from official documentation of memSQL:

```
#!/usr/bin/python
# encoding=utf8
import os
import struct
import sys
reload(sys)
sys.setdefaultencoding('utf8')
from datetime import datetime
import json
```

```
binary_stdin = sys.stdin if sys.version_info < (3, 0) else sys.stdin.buffer
binary_stderr = sys.stderr if sys.version_info < (3, 0) else sys.stderr.buffer
binary_stdout = sys.stdout if sys.version_info < (3, 0) else sys.stdout.buffer

def input_stream():
    """
        Consume STDIN and yield each record that is received from MemSQL
    """
    while True:
        byte_len = binary_stdin.read(8)
        if len(byte_len) == 8:
            byte_len = struct.unpack("L", byte_len)[0]
            result = binary_stdin.read(byte_len)
            yield result
        else:
            assert len(byte_len) == 0, byte_len
            Return

def log(message):
    """
        Log an informational message to stderr which will show up in MemSQL in
        the event of transform failure.
    """
    binary_stderr.write(message + b"\n")

def emit(message):
    """
        Emit a record back to MemSQL by writing it to STDOUT.  The record
        should be formatted as JSON, Avro, or CSV as it will be parsed by
        LOAD DATA.
    """
    binary_stdout.write(message + b"\n")

log(b"Begin transform")

# We start the transform here by reading from the input_stream() iterator.
for data in input_stream():
    tweet = json.loads(data)
    Id = tweet["Id"]
    body = tweet["Text"]
    retweet_count = tweet["RetweetCount"]
    created = datetime.now()
    out_record = (Id, body, retweet_count, created)
    out_str = "\t".join([str(field) for field in out_record])
    out = b"%s\n" % out_str
    # Since this is an identity transform we just emit what we receive.
    emit(out)

log(b"End transform")
```

MemSQL to Metabase

For connecting our database to Metabase, we can find the documentation here:
https://www.metabase.com/docs/latest/setting-up-metabase.html
We can use MySQL connector for the memSQL database. We can select this connector in the drop-down menu. For the database address, use the IP-address of the memSQL database container. The port by default is 3306. Besides these, we have to select the database and provide the credentials to access the database.

Testing

Kafka Consumer Testing

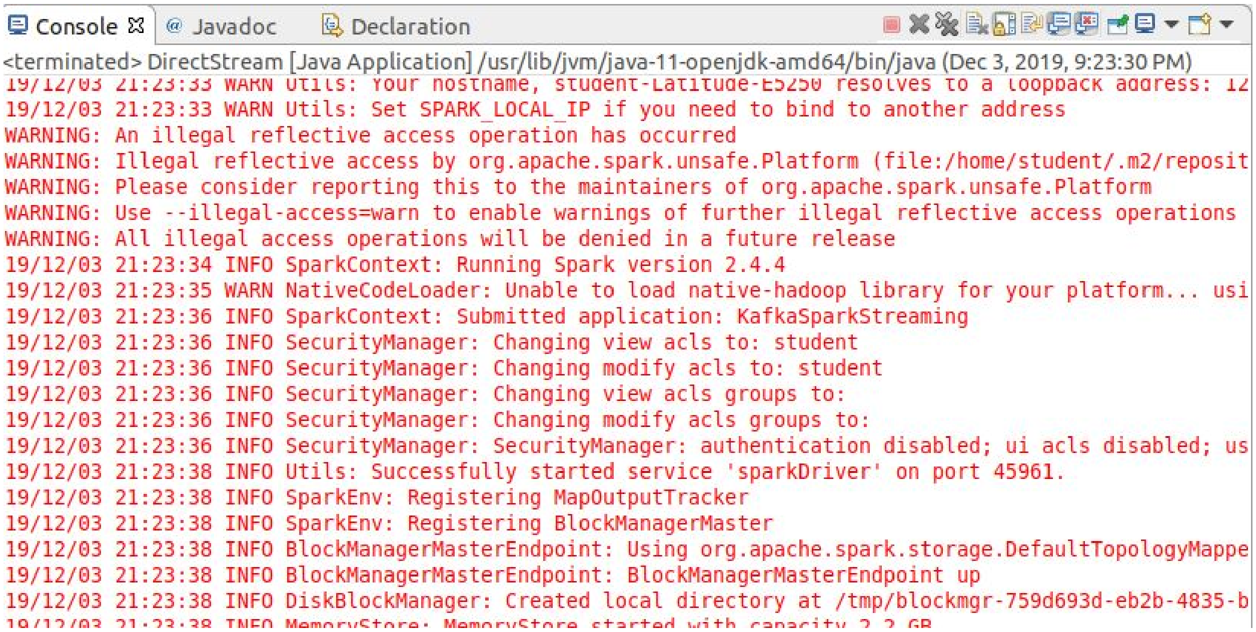
To test if we are receiving the streaming data in Kafka topic from Twitter API, we can create a Kafka Consumer that listens to that topic.

```
docker-compose exec broker kafka-console-consumer --bootstrap-server
localhost:29092 --topic twitter --from-beginning
```

To execute this command, first, we have to go to the directory which contains our docker-compose.yml file. If everything is ok, we will see the streaming data in real-time.

Spark Testing

For testing Spark, we can simply run our Maven project in Eclipse and see the console screen. We will see the connection status and other log information along with the data we are receiving.

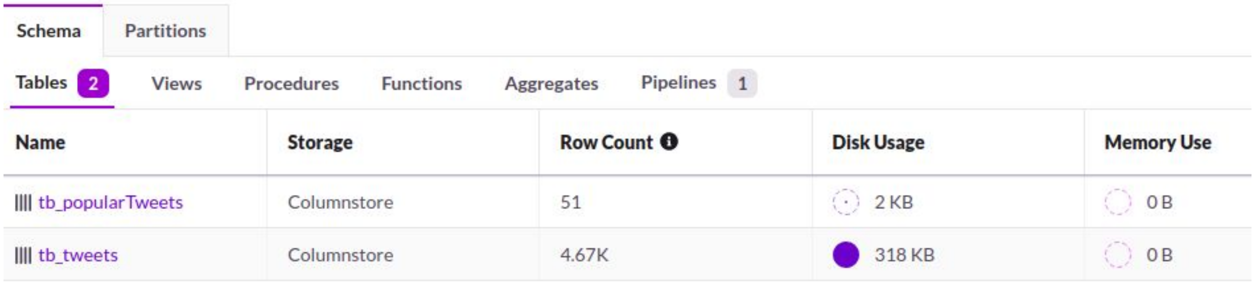


```
<terminated> DirectStream [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Dec 3, 2019, 9:23:30 PM)
19/12/03 21:23:33 WARN Utils: Your hostname, student-Latitude-E5250 resolves to a loopback address: 12
19/12/03 21:23:33 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/home/student/.m2/reposit
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
19/12/03 21:23:34 INFO SparkContext: Running Spark version 2.4.4
19/12/03 21:23:35 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... usi
19/12/03 21:23:36 INFO SparkContext: Submitted application: KafkaSparkStreaming
19/12/03 21:23:36 INFO SecurityManager: Changing view acls to: student
19/12/03 21:23:36 INFO SecurityManager: Changing modify acls to: student
19/12/03 21:23:36 INFO SecurityManager: Changing view acls groups to:
19/12/03 21:23:36 INFO SecurityManager: Changing modify acls groups to:
19/12/03 21:23:36 INFO SecurityManager: authentication disabled; ui acls disabled; us
19/12/03 21:23:38 INFO Utils: Successfully started service 'sparkDriver' on port 45961.
19/12/03 21:23:38 INFO SparkEnv: Registering MapOutputTracker
19/12/03 21:23:38 INFO SparkEnv: Registering BlockManagerMaster
19/12/03 21:23:38 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMappe
19/12/03 21:23:38 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
19/12/03 21:23:38 INFO DiskBlockManager: Created local directory at /tmp/blockmgr-759d693d-eb2b-4835-b
19/12/03 21:23:38 INFO MemoryStore: MemoryStore started with capacity 2.2 GB
```

Image 8: Spark Connection Status in Eclipse Console

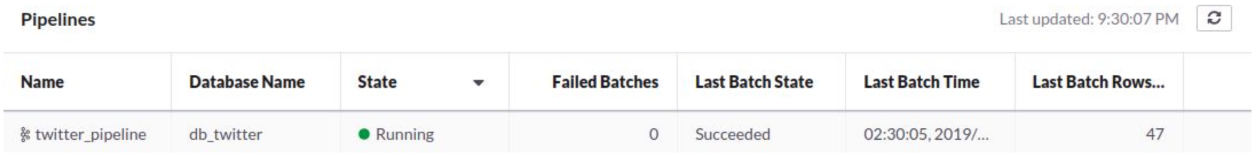
MemSQL Testing

For testing memSQL, we can go to the memSQL Studio (localhost:8080) and in the query editor, we can run the select query and see the data. Also, we can see the database and table information there with the number of rows of data. We have to verify the pipeline is running by checking the pipeline status.



Name	Storage	Row Count	Disk Usage	Memory Use
tb_popularTweets	Columnstore	51	2 KB	0 B
tb_tweets	Columnstore	4.67K	318 KB	0 B

Image 9: Table Overview in memSQL



Name	Database Name	State	Failed Batches	Last Batch State	Last Batch Time	Last Batch Rows...
twitter_pipeline	db_twitter	● Running	0	Succeeded	02:30:05, 2019/...	47

Image 10: Pipeline status in memSQL

Metabase Testing

We will be able to view our data in Metabase if it is connected properly to our database. In the image, 'Tb Popular Tweets' and 'Tb Tweets' are created from two tables in our database for storing popular hashtags in tweets and tweets related to Villanova University respectively.

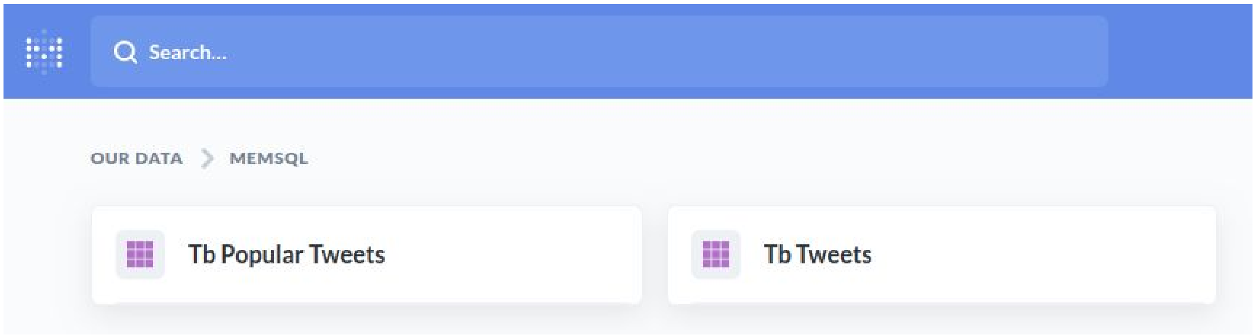


Image 11: List of connected tables in Metabase

Results

The tweets related to Villanova University were successfully streamed from Twitter API to Kafka and top hashtags in a certain time period were analyzed by using Spark and results were stored in the memSQL database. Metabase was connected to the memSQL database and results were visualized in the graphs.

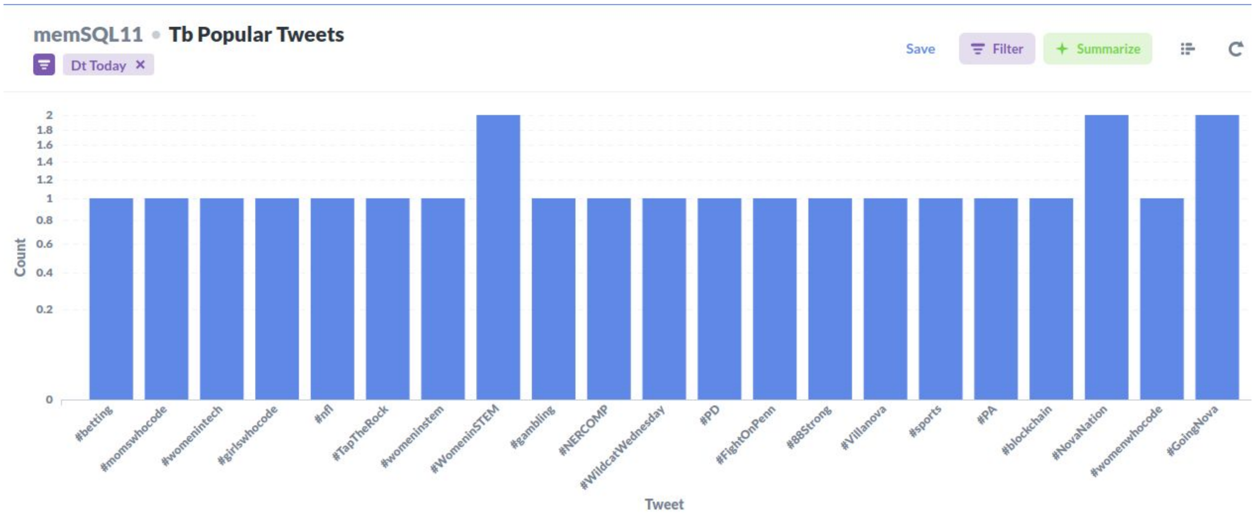


Image 12: visualization of trending tweets related to Villanova University

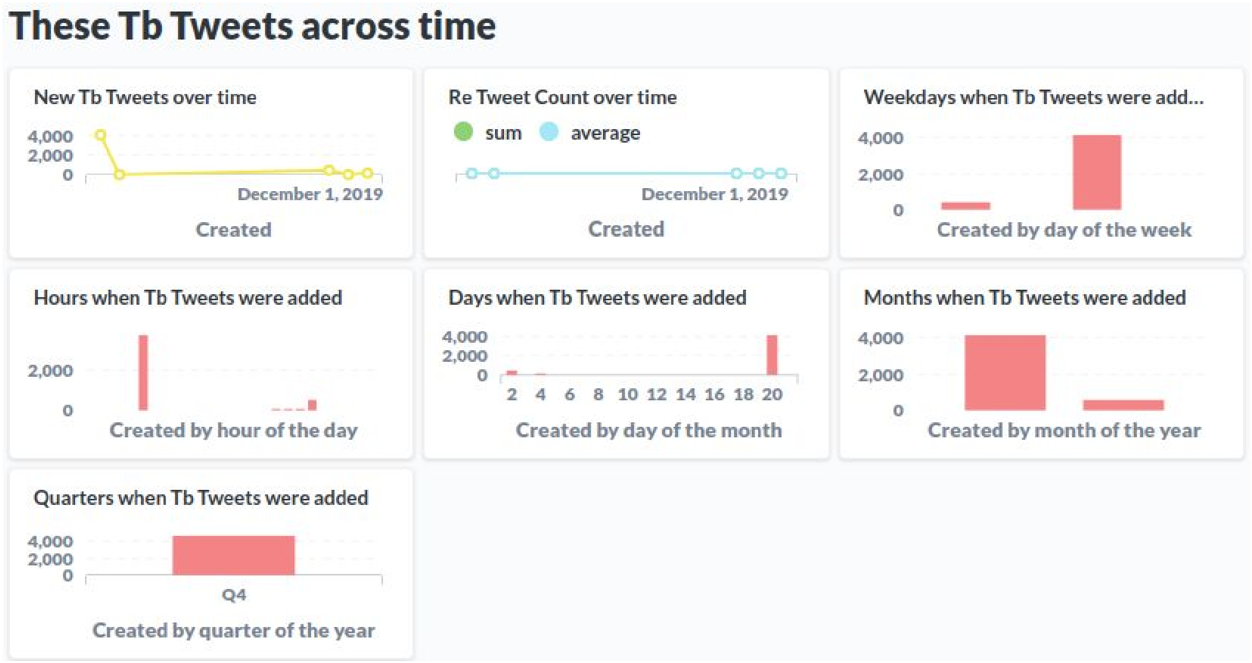

These Tb Tweets across time
New Tb Tweets over time
4,000
2,000
0
December 1, 2019
Created
Re Tweet Count over time
sum
average
December 1, 2019
Created
Weekdays when Tb Tweets were add...
4,000
2,000
0
Created by day of the week
Hours when Tb Tweets were added
2,000
0
Created by hour of the day
Days when Tb Tweets were added
4,000
2,000
0
2 4 6 8 10 12 14 16 18 20
Created by day of the month
Months when Tb Tweets were added
4,000
2,000
0
Created by month of the year
Quarters when Tb Tweets were added
4,000
2,000
0
Q4
Created by quarter of the year

Image 13: Visualization of Tweets related to Villanova University

Whenever there are any new tweets related to the filter keywords in Twitter APIs, it's persisted into Kafka topic in real-time. The Docker containers are set up to restart automatically in case of failures. For our experiment, we have installed a single instance of Kafka with one broker, which may not give 100% fault tolerance but still, we had a pretty good fault-tolerating system. If somehow, the Java project is not working and the link between Kafka and Spark is broken or say the pipeline to the memSQL database is not working, we still have all the tweets stored in the Kafka log file which is fetched later once the system starts up again. This will minimize the loss of data. To achieve more fault tolerance and to make the system more effective, we can install multiple clusters of the database, Spark and Kafka broker as well. We can scale these technologies by increasing the cluster number and its capacity as per the requirements.

Achievements And Challenges

During this project, we learned about different technologies using different programming languages. The project is a proof of concept and not ready for production but we have done different experiments here. Various documentation and blogs were referenced while doing this project which we have cited in the references.

The first challenge was to decide the data source. We tried different resources at Villanova University but were unable to find the right data source for our project. Later, we decided to go with Twitter.

We have used Docker for containerization. At the end of the project, we had a total of 12 containers running. We also learned to use Portainer to manage these Docker containers.

We have used the Java Maven project for Spark. We have also tried PySpark using Python language to connect Kafka and Spark. In our experience using PySpark is easy for prototyping but finding the related documentation in detail was really a challenge. Also, the latest version of Spark doesn’t support Python yet.

We have used agile methodology for this project setting up 2 weeks for a sprint. Tasks were divided accordingly. Sometimes, we were not able to complete the tasks of the sprint in the time frame because of the errors that needed debugging or the inability to find suitable resources and documentation. Trello was used to manage project planning. The screenshot of the project planning in the Trello board is included in Appendix I.

Another challenge we faced was setting up the Kubernetes environment as it was out of our knowledge domain and there were many containers that we had to deploy and made them work together. Although we were not able to implement the Kubernetes environment for this project, we got some knowledge about it and if time was no limit we would have spent more time on it.

Future Work

1. Kubernetes:

   We can deploy all these containers in Kubernetes for better performance, fault tolerance, and load balancing.

2. More data analysis

   We have more than 3K rows of data stored in the database per week. Right now, we are only analyzing the top hashtags in a certain time frame. We can add other analyses and visualize them to make the project more effective. We can develop a system to detect any unusual activities in the tweets.

3. Machine Learning

   On top of Spark, we can use machine learning algorithms to detect any weird or abnormal tweets. We can also analyze the positive or negative tone of the tweets.

4. Tableau Integration

   We can host our database in the server and install Tableau Desktop in the client machine.

References

[1] “What Is Apache Kafka?” *Confluent*, https://www.confluent.io/what-is-apache-kafka/.

[2] “What Is ZooKeeper and Why Is It Needed for Apache Kafka? - CloudKarafka, Apache Kafka Message Streaming as a Service.” *CloudKarafka*, https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka_what_is_zookeeper.html.

[3] Laskowski, Jacek. “Overview of Apache Spark.” *Overview of Apache Spark · The Internals of Apache Spark*, https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-overview.html#targetText=Spark aims at speed, ease, and interactive ad hoc queries.

[4] “The 5-Minute Guide to Understanding the Significance of Apache Spark.” *MapR*, https://mapr.com/blog/5-minute-guide-understanding-significance-apache-spark/.

[5] “How MemSQL Works.” *MemSQL Documentation*, https://docs.memsql.com/v6.8/introduction/how-memsql-works/.

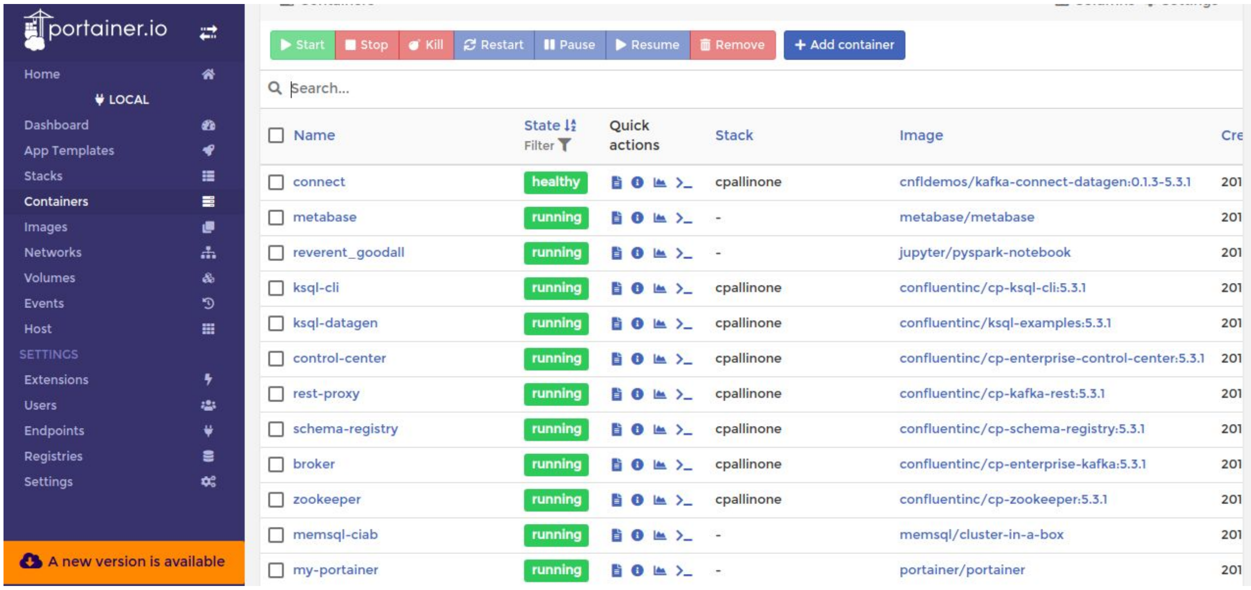
[6] “What Is Tableau? Uses and Applications.” *Guru99*, https://www.guru99.com/what-is-tableau.html.

[7] “What Is Docker?” Opensource.com, https://opensource.com/resources/what-docker.

[8] Aethersg. “Aethersg/Twitter-Stream-Spark-Kafka.” *GitHub*, 14 May 2018, https://github.com/aethersg/twitter-stream-spark-kafka.

Appendix I

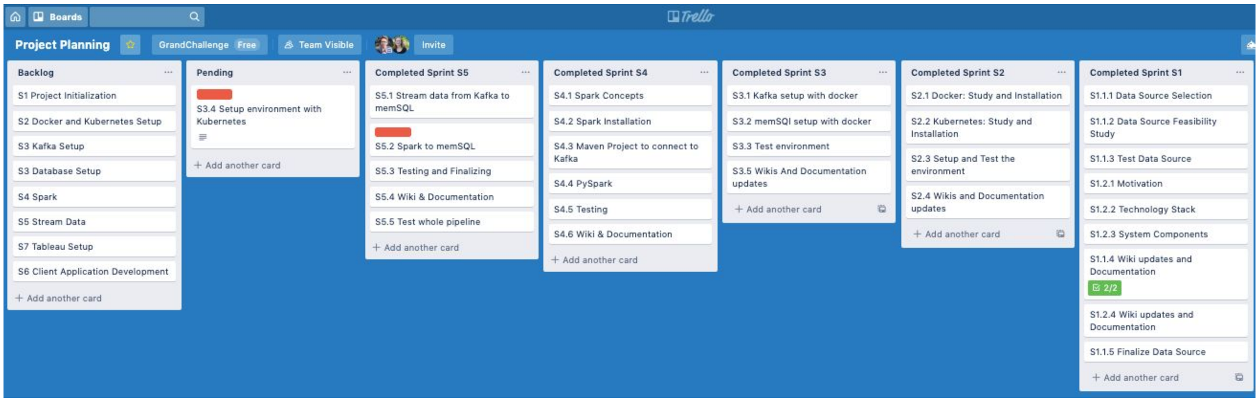
Portainer Screenshot



☐ Name	State Filter	Quick actions	Stack	Image	Cre
☐ connect	healthy		cpallinone	cnfldemos/kafka-connect-datagen:0.1.3-5.3.1	201
☐ metabase	running		-	metabase/metabase	201
☐ reverent_goodall	running		-	jupyter/pyspark-notebook	201
☐ ksql-cli	running		cpallinone	confluentinc/cp-ksql-cli:5.3.1	201
☐ ksql-datagen	running		cpallinone	confluentinc/ksql-examples:5.3.1	201
☐ control-center	running		cpallinone	confluentinc/cp-enterprise-control-center:5.3.1	201
☐ rest-proxy	running		cpallinone	confluentinc/cp-kafka-rest:5.3.1	201
☐ schema-registry	running		cpallinone	confluentinc/cp-schema-registry:5.3.1	201
☐ broker	running		cpallinone	confluentinc/cp-enterprise-kafka:5.3.1	201
☐ zookeeper	running		cpallinone	confluentinc/cp-zookeeper:5.3.1	201
☐ memsql-ciab	running		-	memsql/cluster-in-a-box	201
☐ my-portainer	running		-	portainer/portainer	201

Image: Portainer and related containers for the project

Trello Screenshot



Backlog
- S1 Project Initialization
- S2 Docker and Kubernetes Setup
- S3 Kafka Setup
- S3 Database Setup
- S4 Spark
- S5 Stream Data
- S7 Tableau Setup
- S6 Client Application Development

Pending
- S3.4 Setup environment with Kubernetes

Completed Sprint S5
- S5.1 Stream data from Kafka to memSQL
- S5.2 Spark to memSQL
- S5.3 Testing and Finalizing
- S5.4 Wiki & Documentation
- S5.5 Test whole pipeline

Completed Sprint S4
- S4.1 Spark Concepts
- S4.2 Spark Installation
- S4.3 Maven Project to connect to Kafka
- S4.4 PySpark
- S4.5 Testing
- S4.6 Wiki & Documentation

Completed Sprint S3
- S3.1 Kafka setup with docker
- S3.2 memSQl setup with docker
- S3.3 Test environment
- S3.5 Wikis And Documentation updates

Completed Sprint S2
- S2.1 Docker: Study and Installation
- S2.2 Kubernetes: Study and Installation
- S2.3 Setup and Test the environment
- S2.4 Wikis and Documentation updates

Completed Sprint S1
- S1.1.1 Data Source Selection
- S1.1.2 Data Source Feasibility Study
- S1.1.3 Test Data Source
- S1.2.1 Motivation
- S1.2.2 Technology Stack
- S1.2.3 System Components
- S1.1.4 Wiki updates and Documentation ☑ 2/2
- S1.2.4 Wiki updates and Documentation
- S1.1.5 Finalize Data Source

Image: Trello Board For Project Planning

Appendix II

Code Snippet

DirectStream.java

```
package com.grandChallenge.project.kafkaspark;

import java.util.ArrayList;
import java.util.List;

public class DirectStream {
      public static void main(String[] args) throws Exception {
            List<String> topicList = new ArrayList<>();

            topicList.add("twitter");
            //KafkaSpark kafkaStream = new
KafkaSpark("KafkaSparkStreaming", topicList, 1);

            Consumer kafkaConsumer = new Consumer("KafkaSparkStreaming",
topicList, 1);
      }
}
```

HashTagUtils.java

```
package com.grandChallenge.project.kafkaspark;

import java.util.*;
import java.util.regex.*;

public class HashTagsUtils {
       private static final Pattern HASHTAG_PATTERN =
Pattern.compile("#\\w+");

          public static Iterator<String> hashTagsFromTweet(String text) {
              List<String> hashTags = new ArrayList<>();
              Matcher matcher = HASHTAG_PATTERN.matcher(text);
              while (matcher.find()) {
                  String handle = matcher.group();
                  hashTags.add(handle);
              }
              return hashTags.iterator();
          }
}
```

Consumer.java

```
package com.grandChallenge.project.kafkaspark;

import java.io.Serializable;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaInputDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.kafka010.ConsumerStrategies;
import org.apache.spark.streaming.kafka010.KafkaUtils;
import org.apache.spark.streaming.kafka010.LocationStrategies;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.grandChallenge.project.kafkaspark.config.DatabaseConfig;

import scala.Tuple2;

/**
 * @author student
 *
 */
public class Consumer implements Serializable {
      private static final DateTimeFormatter dtf =
DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
      int id;
      Connection conn = null;
      Statement stmt = null;

      public Consumer(String sparkName, List<String> topicList, int
threadsNum) throws InterruptedException, SQLException {
            SparkConf sparkConf = new
SparkConf().setAppName(sparkName).setMaster("local[*]");
            JavaStreamingContext sc = new JavaStreamingContext(sparkConf,
Durations.seconds(500));
            storeSparkToDatabase(topicList, threadsNum, sc);
      }

      private void storeSparkToDatabase(List<String> topicList, int
threadsNum, JavaStreamingContext sc)
                  throws InterruptedException, SQLException {
            Collection<String> topics = topicList;

            Map<String, Object> kafkaParams = new HashMap<>();
            kafkaParams.put("bootstrap.servers", "172.17.0.3:9092");
            kafkaParams.put("key.deserializer",
StringDeserializer.class);
            kafkaParams.put("value.deserializer",
StringDeserializer.class);
            kafkaParams.put("group.id",
"use_a_separate_group_id_for_each_stream");
            kafkaParams.put("auto.offset.reset", "latest");
            kafkaParams.put("enable.auto.commit", false);

            JavaInputDStream<ConsumerRecord<String, String>> stream =
KafkaUtils.createDirectStream(sc,
                        LocationStrategies.PreferConsistent(),
                        ConsumerStrategies.<String,
String>Subscribe(topics, kafkaParams));

            JavaDStream<String> lines = stream.map(x -> x.value());

lines.flatMap(HashTagsUtils::hashTagsFromTweet).mapToPair(hashTag -> new
Tuple2<>(hashTag, 1))
                        .reduceByKey(Integer::sum)

                        .mapToPair(Tuple2::swap).foreachRDD(rrdd -> {
```

```
System.out.println("\n\n\n-----------------------------------------------
----------------\n\n\n");
                               List<Tuple2<Integer, String>> sorted;
                               JavaPairRDD<Integer, String> counts =
rrdd.sortByKey(false);
                               sorted = counts.collect();
                               Gson gson = new GsonBuilder().create();


                               try {

Class.forName(DatabaseConfig.JDBC_DRIVER);

                                     conn =
DriverManager.getConnection(DatabaseConfig.DB_URL,
DatabaseConfig.dbUserName,

DatabaseConfig.dbUserPassword);
                                      stmt = conn.createStatement();
                                     LocalDateTime dt =
LocalDateTime.now();
                                     sorted.forEach(record -> {
                                           id = generateUniqueId();
                                           String sql = "Insert into
tb_popularTweets(id, Tweet, Count, dt) values (" + id + ", \""
                                                       + record._2 + "\",
" + record._1 + ", \"" + dtf.format(dt) +"\")";

                                           try {
                                                 stmt.executeUpdate(sql);
                                           } catch (SQLException e) {
                                                 // TODO Auto-generated
catch block

                                                 e.printStackTrace();
                                           }

                                     });
                               } catch (Exception se) {
                                     se.printStackTrace();
                               } finally {
                                     try {
                                           assert conn != null;
                                           conn.close();
                                     } catch (SQLException e) {
                                           e.printStackTrace();
                                     }
                                     try {
```

```
                            conn.close();
                        } catch (SQLException se) {
                            se.printStackTrace();
                        }
                    }

                    // System.out.println(String.format(" %s
(%d)", record._2, record._1)));
                });


        sc.start();
        sc.awaitTermination();

    }

    public static int generateUniqueId() {
        UUID idOne = UUID.randomUUID();
        String str = "" + idOne;
        int uid = str.hashCode();
        String filterStr = "" + uid;
        str = filterStr.replaceAll("-", "");
        return Integer.parseInt(str);
    }

}
```