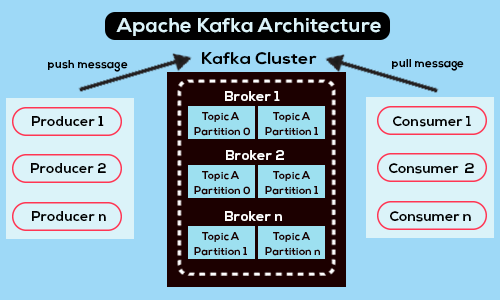
**Kafka**

Apache Kafka is one of the most popular open-source distributed streaming platforms for processing large volumes of streaming data from real-time applications.

Apache Kafka originated at LinkedIn and later became an open sourced Apache project in 2011, then First-class Apache project in 2012. Kafka is written in Scala and Java.

**Why Kafka**

* Scalability: The scalability of a system is determined by how well it can maintain its performance when exposed to changes in application and processing demands. Apache Kafka has a distributed architecture capable of handling incoming messages with higher volume and velocity. As a result, Kafka is highly scalable without any downtime impact.
* High Throughput: Apache Kafka is able to handle thousands of messages per second. Messages coming in at a high volume or a high velocity or both will not affect the performance of Kafka.
* Low Latency: Latency refers to the amount of time taken for a system to process a single event. Kafka offers a very low latency, which is as low as ten milliseconds.
* Fault Tolerance: By using replication, Kafka can handle failures at nodes in a cluster without any data loss. Running processes, too, can remain undisturbed. The replication factor determines the number of replicas for a partition. For a replication factor of ‘n,’ Kafka guarantees a fault tolerance for up to n-1 servers in the Kafka cluster.
* Reliability: Apache Kafka is a distributed platform with very high fault tolerance, making it a very reliable system to use.
* Durability: Data present on the Kafka cluster is allowed to remain persistent more on the cluster than on the disk. This ensures that Kafka’s data remains durable.-

**Topics**

A stream of messages that are a part of a specific category or feed name is referred to as a Kafka topic. In Kafka, data is stored in the form of topics. Producers write their data to topics, and consumers read the data from these topics.

**Brokers**

A Kafka cluster comprises one or more servers that are known as brokers. In Kafka, a broker works as a container that can hold multiple topics with different partitions. A unique integer ID is used to identify brokers in the Kafka cluster. Connection with any one of the kafka brokers in the cluster implies a connection with the whole cluster. If there is more than one broker in a cluster, the brokers need not contain the complete data associated with a particular topic.

**Consumers and Consumer Groups**

Consumers read data from the Kafka cluster. The data to be read by the consumers has to be pulled from the broker when the consumer is ready to receive the message. A consumer group in Kafka refers to a number of consumers that pull data from the same topic or same set of topics.

**Producers**

Producers in Kafka publish messages to one or more topics. They send data to the Kafka cluster. Whenever a Kafka producer publishes a message to Kafka, the broker receives the message and appends it to a particular partition. Producers are given a choice to publish messages to a partition of their choice.

**Partitions**

Topics in Kafka are divided into a configurable number of parts, which are known as partitions. Partitions allow several consumers to read data from a particular topic in parallel. Partitions are separated in order. The number of partitions is specified when configuring a topic, but this number can be changed later on. The partitions comprising a topic are distributed across servers in the Kafka cluster. Each server in the cluster handles the data and requests for its share of partitions. Messages are sent to the broker along with a key. The key can be used to determine which partition that particular message will go to. All messages which have the same key go to the same partition. If the key is not specified, then the partition will be decided in a round-robin fashion.

**Partition Offset**

Messages or records in Kafka are assigned to a partition. To specify the position of the records within the partition, each record is provided with an offset. A record can be uniquely identified within its partition using the offset value associated with it. A partition offset carries meaning only within that particular partition. Older records will have lower offset values since records are added to the ends of partitions.

**Replicas**

Replicas are like backups for partitions in Kafka. They are used to ensure that there is no data loss in the event of a failure or a planned shutdown. Partitions of a topic are published across multiple servers in a Kafka cluster. Copies of the partition are known as Replicas.

**Leader and Follower**

Every partition in Kafka will have one server that plays the role of a leader for that particular partition. The leader is responsible for performing all the read and write tasks for the partition. Each partition can have zero or more followers. The duty of the follower is to replicate the data of the leader. In the event of a failure in the leader for a particular partition, one of the follower nodes can take on the role of the leader.

**User Guide:**

Please refer below Kafka documentation to install and setup Kafka on your machine. <https://kafka.apache.org/quickstart>

Before we Start The kafka We Need To start Zookeeper Server. Zookeeper is the one who manages and coordinates Kafka brokers in the cluster. It is used to notify the producer and consumer about the presence of new brokers or about the failure of brokers in the Kafka cluster.

1. Please run the below command in the terminal to start zookeeper. -bin/zookeeper-server-start.sh config/zookeeper.properties
2. Please run the below command in the terminal to start the kafka server. bin/kafka-server-start.sh config/server.properties
3. Check Server Properties: bin/kafka-server-start.sh config/server.properties
4. Check All Topics: bin/kafka-topics.sh --list --bootstrap-server localhost:9092 all topics
5. Please run the below command in the terminal to work with multiple instances re-balancing features with kafka. bin/kafka-topics.sh --bootstrap-server <host>:<port> --alter --partitions <no-of-partitions-defined-in-property-file> --topic <topic-name>
6. If You Want Create Replication Factor : bin/kafka-topics.sh --create --topic <topic-name> --bootstrap-server localhost:9092 --replication-factor 1 --partitions 4
7. delete topic IF WANT: kafka-topics.sh --zookeeper localhost:2181 --delete --topic <topic-name> (bin/kafka-topics.sh --delete --topic orderstopic --bootstrap-server localhost:9092 )
8. retrieve list of topics in kafka : kafka-topics.sh --list --zookeeper localhost:2181

**Kafka** **Architecture**

In Kafka There are 3 Main things.

1. Producer

2. Kafka Cluster

3. Consumer

Normally the producer's task is to produce a message and push it to the Kafka cluster. But inside the cluster There are topics. We can create any number of topics inside this cluster. Inside those topics there are partitions.

Kafka stores their values inside the partitions.

How it’s store

It Take Key Value (Number and then Take mode value from Number of partitions)

Example : Suppose, We have 5 partitions inside the topic and our key value is 104. It takes 104 % 5 = 1 and stores data in the 1st partition.

But here we can Create More consumers to connect to one topic. It automatically handles load balancing and sends all messages to all consumers. And also we can create Brokers and set data replication factors.

Suppose we have 3 brokers and only one topic. Our replication factor is 3. Now If I store data inside that partition it will replicate that data into all three brokers. And one broker becomes Leader and access to it. If there are many values they all have one leader with load balancing. If one leader fault It automatically handles fault tolerance. One becomes a leader from the remaining users.

Consumer Consume data and we can set more than One consumer to one topic.

**Segmentation Features In COG**

Item/Customer

* Psycho graphic
* 1. Abandonment Orders – User Id , Total Amount
* 2. Abandonment Reservation – User Id , Item Id
* 3. Checkout History – User Id, Status
* 4. Order Histories – Order ID, Order Status ID, User Id
* 5. Reservation – Order Id, Customer, Item, Order Status ID
* 6. Shopping Cart Items (V5) – Cart Id , Items Id, Amount
* 7. User Profile Analytics – User Id , Order ID
* 8. User Wish List (V5)
* Geographic
* 1. Address Inputs – Coordinates, Created At, Update At ( we can Check Climate and
* other things According to the given location.
* 2. Delivery Manifest – Delivery id, Manifest Id [Manifest Table (city\_id)]
* 3. Business Directory – Name, City, Type
* 4. User Meta Data – User ID , Last Login ip, Visit Count
* Behavioral
* 1. Deliveries – Order Id, Delivery Status Id ( Delivery Status )
* 2. Driver Performance
* 3. Feedback - reservation Id [ Reservation Table ( measure Merchant Level ) ]
* 4. Order Histories – Order Status ID, User Id
* 5. Order – Order Id, Total Amount, Order Status Id ( Order Status Table )
* 6. Order Backups – Order status Id, Order Id / Order Id , Total Amount
* 7. Ratings – Reservation Id, Star Rate
* 8. Refund Request – Order Id, Customer Id ( requested Person )
* 9. Testimonials – Order Id, User Id, Rating
* 10. User Profile Analytics – User Id , Order ID

Item/Order

**Spark**

Cluster Computing Framework. For near Real Time Processing. Spark by default Provide Implicit data Parallelism ( No need any special directives or operators or functions to enable parallel execution.)

Main feature is in memory cluster computing that increases the processing speed of applications.

It can keep data in memory and quickly add multiple operations and final result stores in the disk. Spark Do batch processing 100 times faster than map reduce. That’s why this spark is going to big data processing in industry.

**Spark Core**

In comparison to Map Reduce and other Apache Hadoop components, the Apache Spark API is very friendly to developers, hiding much of the complexity of a distributed processing engine behind simple method calls. The canonical example of this is how almost 50 lines of Map Reduce code to count words.

By providing bindings to popular languages for data analysis like Python and R, as well as the more enterprise-friendly Java and Scala, Apache Spark allows everybody from application developers to data scientists to harness its scalability and speed in an accessible manner.

**Spark Mllib**

Apache Spark also bundles libraries for applying machine learning and graph analysis techniques to data at scale. Spark Mllib includes a framework for creating machine learning pipelines, allowing for easy implementation of feature extraction, selections, and transformations on any structured dataset. Ml lib comes with distributed implementations of clustering and classification algorithms such as k-means clustering and random forests that can be swapped in and out of custom pipelines with ease. Models can be trained by data scientists in Apache Spark using R or Python, saved using Mllib, and then imported into a Java-based or Scala-based pipeline for production use.

ilabs

Note that while Spark Mllib covers basic machine learning including classification, regression, clustering, and filtering, it does not include facilities for modeling and training deep neural networks. However, Deep Learning Pipelines are in the works.

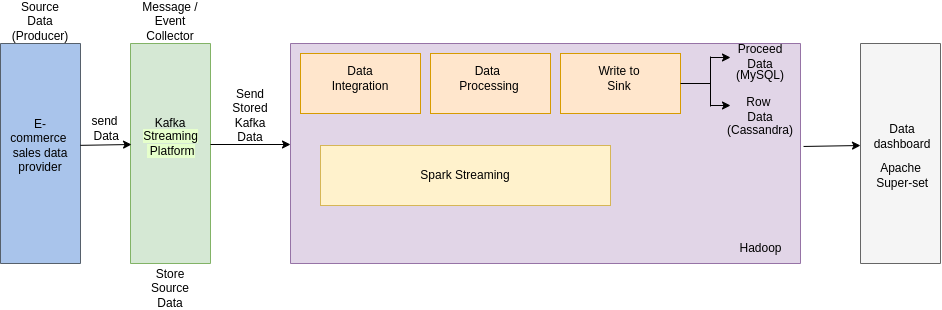
**Spark Streaming**

Spark Streaming was an early addition to Apache Spark that helped it gain traction in environments that required real-time or near real-time processing. Previously, batch and stream processing in the world of Apache Hadoop were separate things. You would write Map Reduce code for your batch processing needs and use something like Apache Storm for your real-time streaming requirements. This obviously leads to disparate code bases that need to be kept in sync for the application domain despite being based on completely different frameworks, requiring different resources, and involving different operational concerns for running them.

Spark Streaming extended the Apache Spark concept of batch processing into streaming by breaking the stream down into a continuous series of micro batches, which could then be manipulated using the Apache Spark API. In this way, code in batch and streaming operations can share (mostly) the same code, running on the same framework, thus reducing both developer and operator overhead. Everybody wins.

A criticism of the Spark Streaming approach is that micro batching, in scenarios where a low-latency response to incoming data is required, may not be able to match the performance of other streaming-capable frameworks like Apache Storm, Apache F link, and Apache Apex, all of which use a pure streaming method rather than micro-batches.

**Sample Architecture Real-Time Spark Project | Real-Time Data Analysis | Segmentation**



When the Producer sends data to the Kafka cluster. Kafka Cluster stores data inside the cluster’s topic. Here we can send data from more than one producer. In this case we can store data topic wise in the Kafka cluster. Zookeeper is the one who manages and coordinates Kafka brokers in the cluster. The data Will send for the data processing. Apache Spark is an open-source unified analytics engine for large-scale data processing. Spark provides an interface for programming clusters with implicit data parallelism and fault tolerance. Then Here I Used Cassandra Database to Store all data and Proceed Data Store In the My SQL. We Can Use Those MySQL Data To Segmentation and Machine Learning.

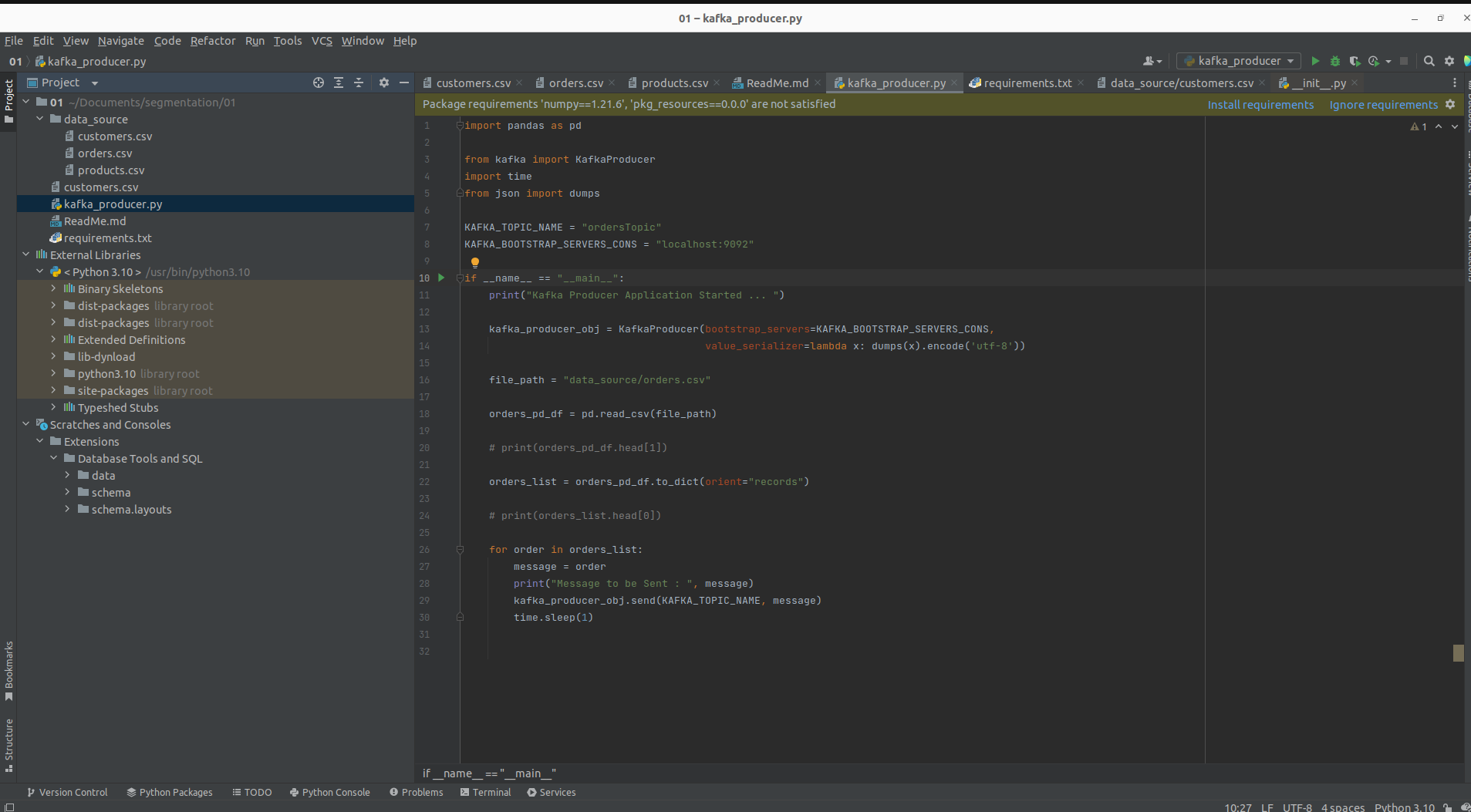
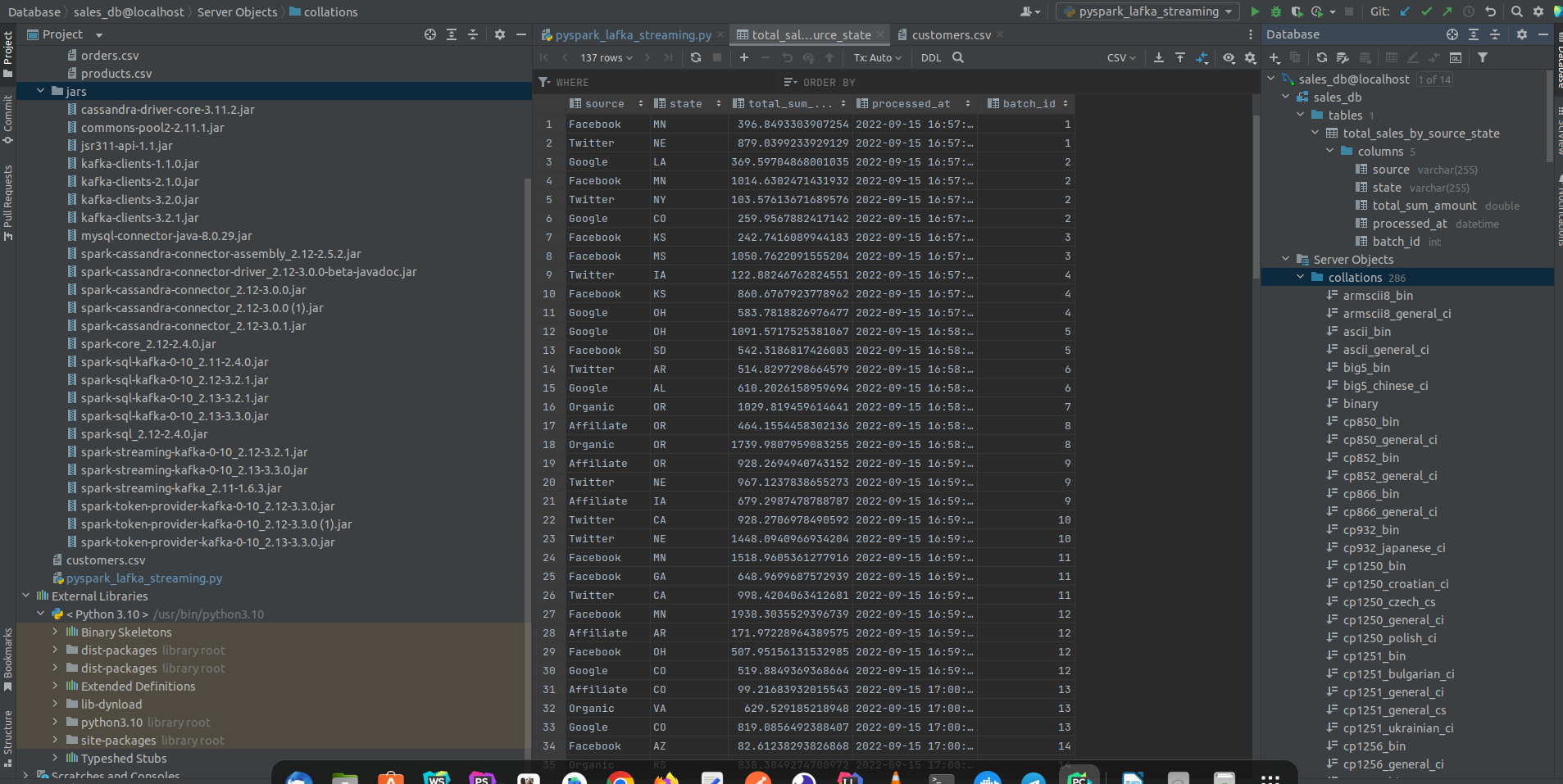
**Sample Project**

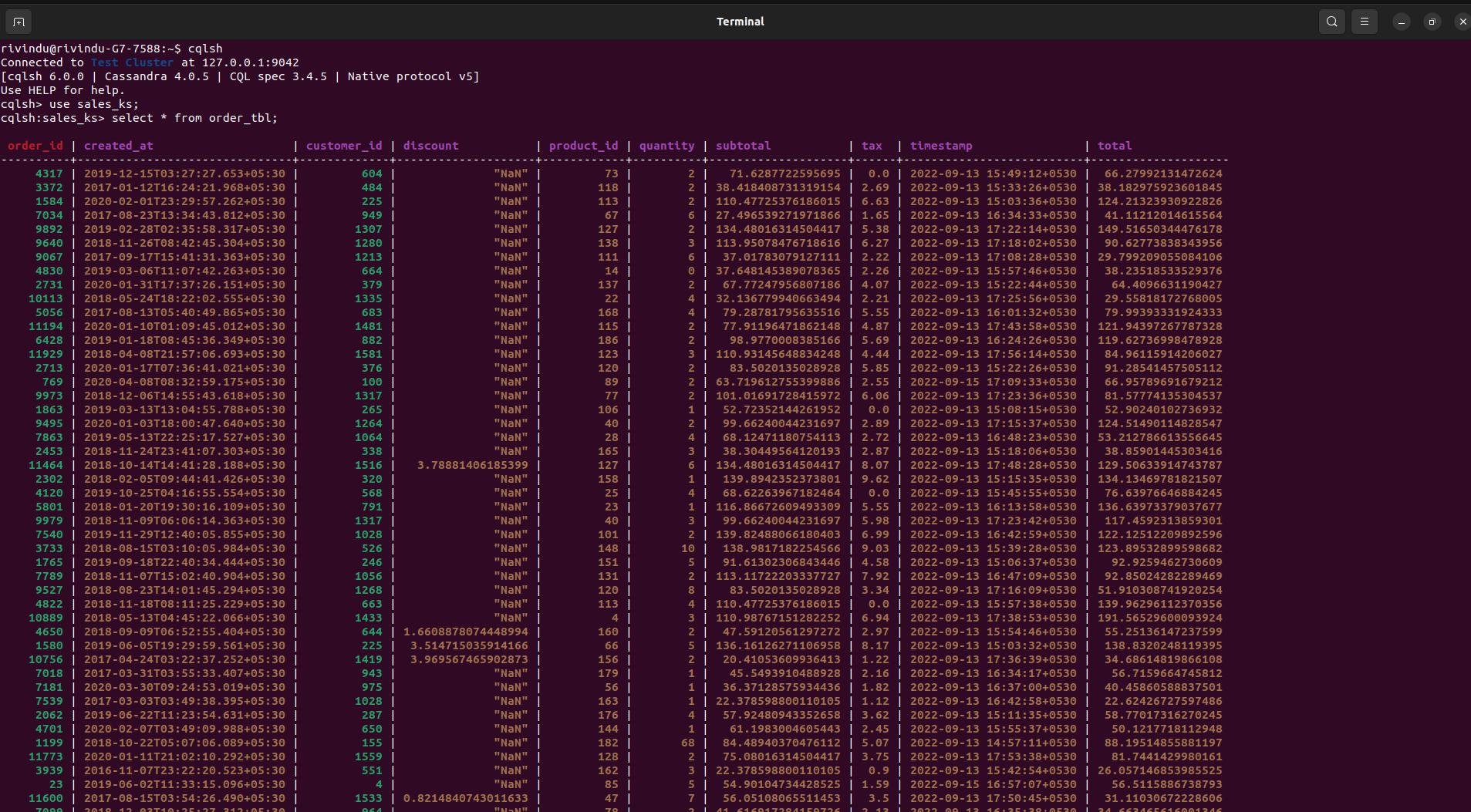
Kafka producer

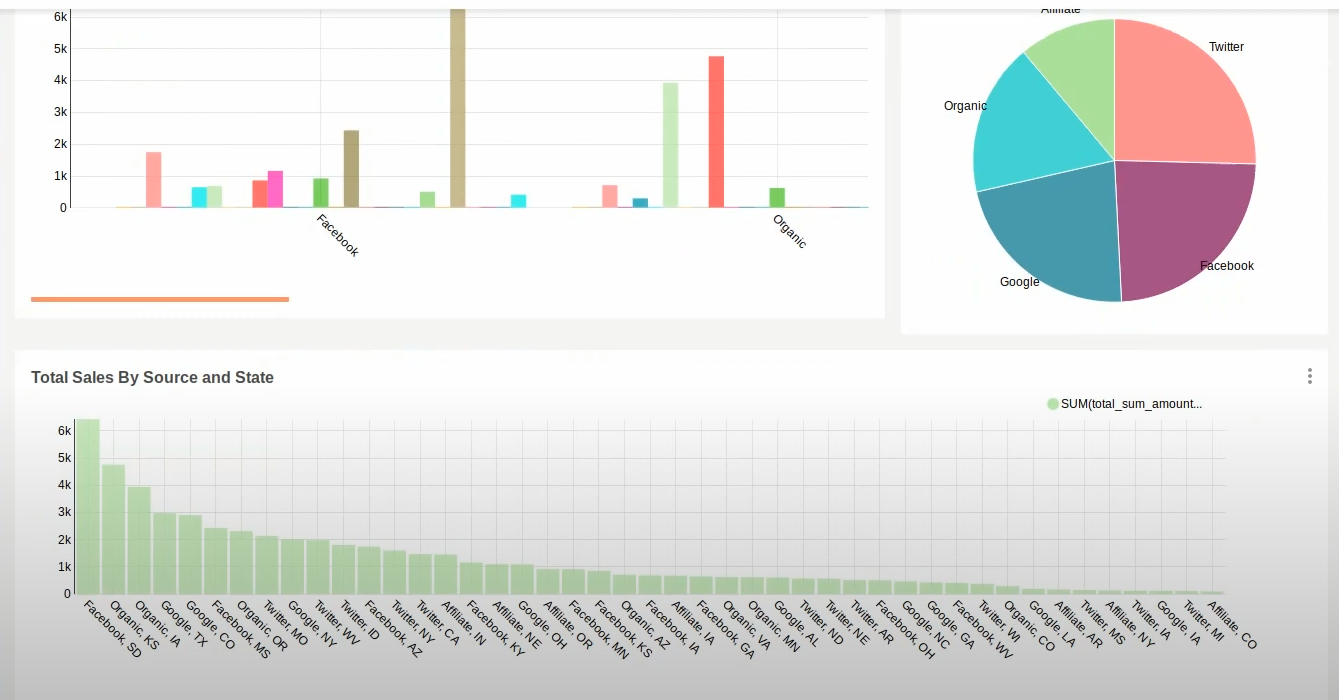
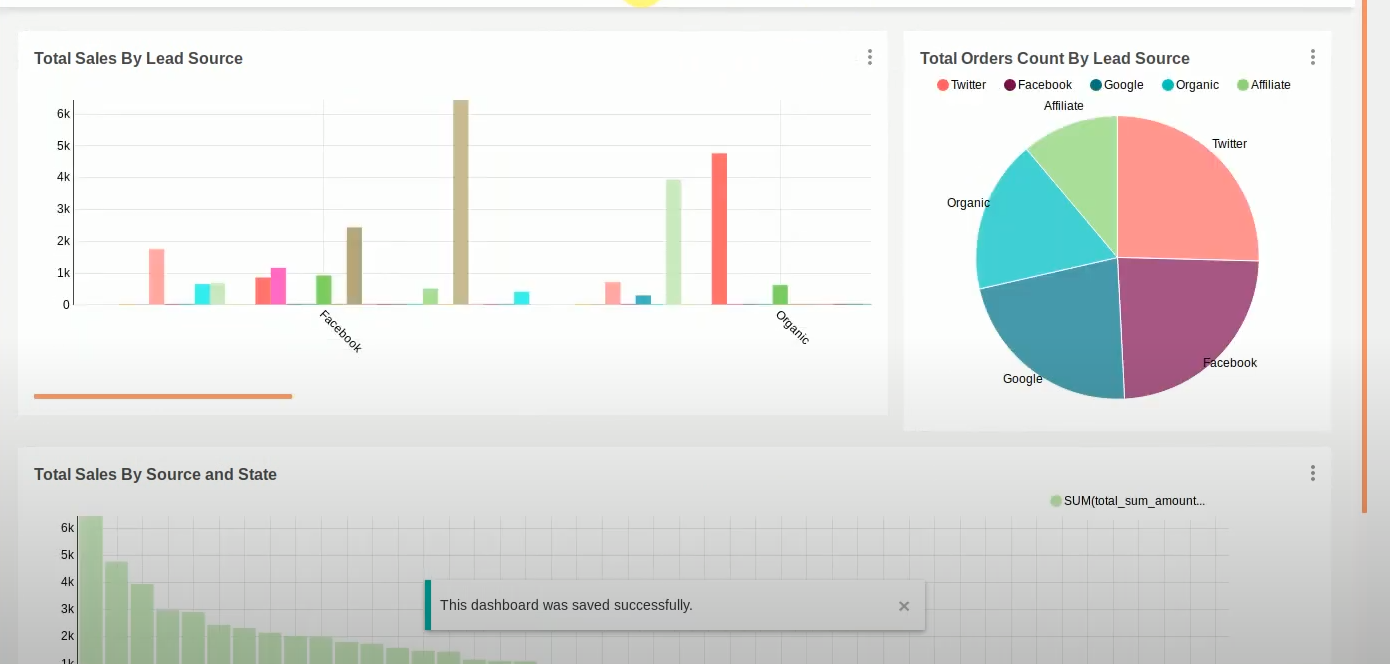
<https://github.com/rivinduchamath/Real-Time-PYSpark-Project-Part-01>

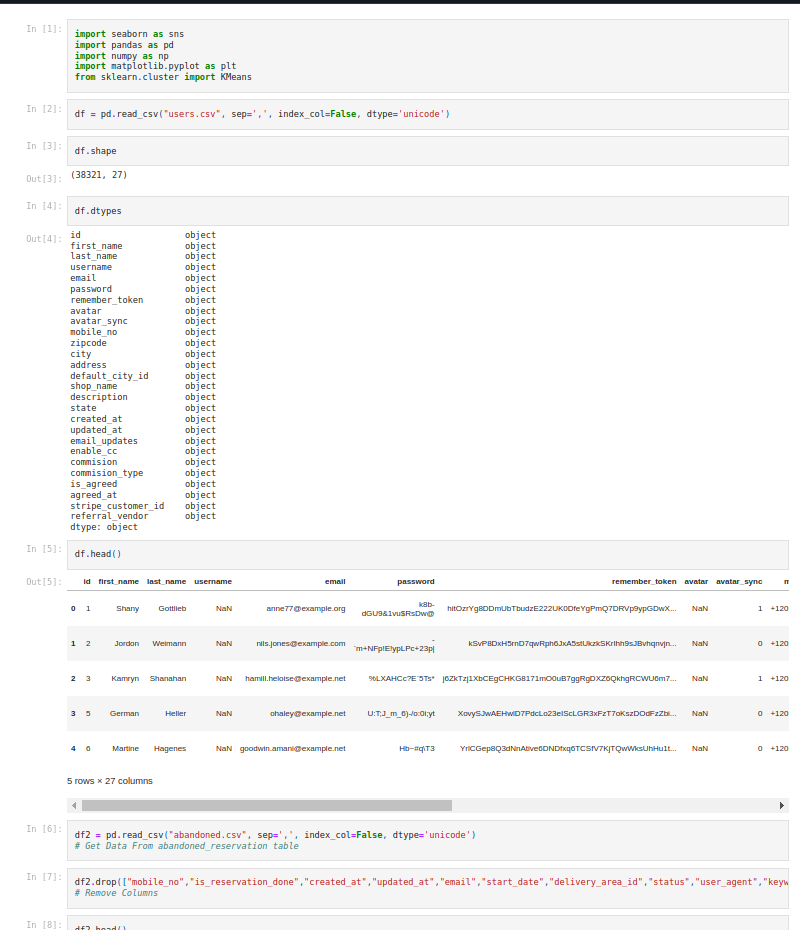
spark

<https://github.com/rivinduchamath/RealTimeSparkWorking-2>









**Segmentation**

Source Code : <https://github.com/rivinduchamath/Segmentation.git>



