It is critical to load the necessary data for the Kafka implementation and Pyspark streaming before beginning the capstone project. The project data is saved in the Google Drive repository, and after mounting the drive, the data is loaded into Google Colab.

A screenshot of a computer

Description automatically generated

Kafka serves as a distributed streaming platform that furnishes a robust and scalable framework for constructing real-time data pipelines and streaming applications. At its core, Kafka consists of vital elements such as producers, brokers, topics, partitions, and consumers, working in harmony to establish an exceptionally efficient and resilient messaging ecosystem.

Outlined below is an overview of the Kafka producer-consumer architecture:

* **Producers**: These are applications or components responsible for generating and dispatching data to Kafka topics. They transmit records (messages) to Kafka brokers, which act as intermediaries connecting producers and consumers. Producers can originate from various data sources, ranging from web servers and sensors to log files or any data-generating application.
* **Brokers**: Kafka brokers serve as servers that manage incoming messages from producers and deliver them to consumers. They constitute the heart of the Kafka cluster and maintain the publish-subscribe model. Brokers store incoming records persistently on disk.
* **Topics**: Topics function as logical channels or categories to which records are published. They operate as message queues, with producers writing data and consumers reading from them. Each topic can be divided into multiple partitions for data storage, and these partitions are replicated across different brokers to ensure fault tolerance.
* **Partitions**: Topics are divided into partitions to distribute the processing load and enable parallelism. Each partition represents an ordered and unchangeable sequence of records. Producers can designate a key while sending a record, aiding in determining the partition where the record will be stored.
* **Consumers**: Consumers are applications or components that retrieve and process data from Kafka topics. They subscribe to one or more topics and consume records from the partitions within those topics. Consumers can participate in a consumer group, facilitating load sharing and parallel processing.
* **Consumer Groups**: Consumer groups are logical sets of consumers that collaborate to consume data from Kafka topics. Each consumer group is assigned to one or more partitions of a topic. A given partition can be consumed by only one consumer within a consumer group at any given time, allowing for concurrent message processing.
* **Offset Management**: Kafka tracks the position of each consumer in every partition using offsets. Offsets denote the position of the last record consumed by a consumer in a partition. This feature permits consumers to resume from where they left off in case of failures or rebalancing.
* **ZooKeeper (Optional, for older Kafka versions)**: In previous Kafka versions, ZooKeeper was employed for cluster state management. However, in more recent iterations, Kafka utilizes its internal metadata management system.

The Kafka producer-consumer architecture offers a host of advantages, including remarkable scalability, fault tolerance, and real-time data processing capabilities. As a result, it has gained popularity as the preferred solution for constructing data pipelines, event-driven applications, log aggregation systems, and other streaming data solutions.

**Yelp Dataset :** Yelp review dataset contains below fields,

* "review\_id": An exclusive identifier assigned to each review.
* "user\_id": Indicates the identification of the user responsible for composing the review.
* "business\_id": Holds the identification of the reviewed business.
* "stars": Represents the user's rating of the business, measured on a scale from 1 (lowest) to 5 (highest).
* "useful": Reflects the count of individuals who deemed the review helpful.
* "funny": Reflects the count of individuals who found the review amusing.
* "cool": Reflects the count of individuals who found the review impressive.
* "text": Contains the actual textual content of the review.

**Kafka Installation:** Following scripts are used for installing the kafka and zookeeper for further operation,

**Step 1: Installation of Kafka and zeekeper.**



**Step 2: Running Kafka and Zookeeper in daemon mode on port 9092**

A screenshot of a computer

Description automatically generated

**Step 3: Creating new topics to use it in the producer and consumer call for sending messages.**

A computer code with text

Description automatically generated

**Step 4 : As in this step install the open jdk and kafka client for further opration**

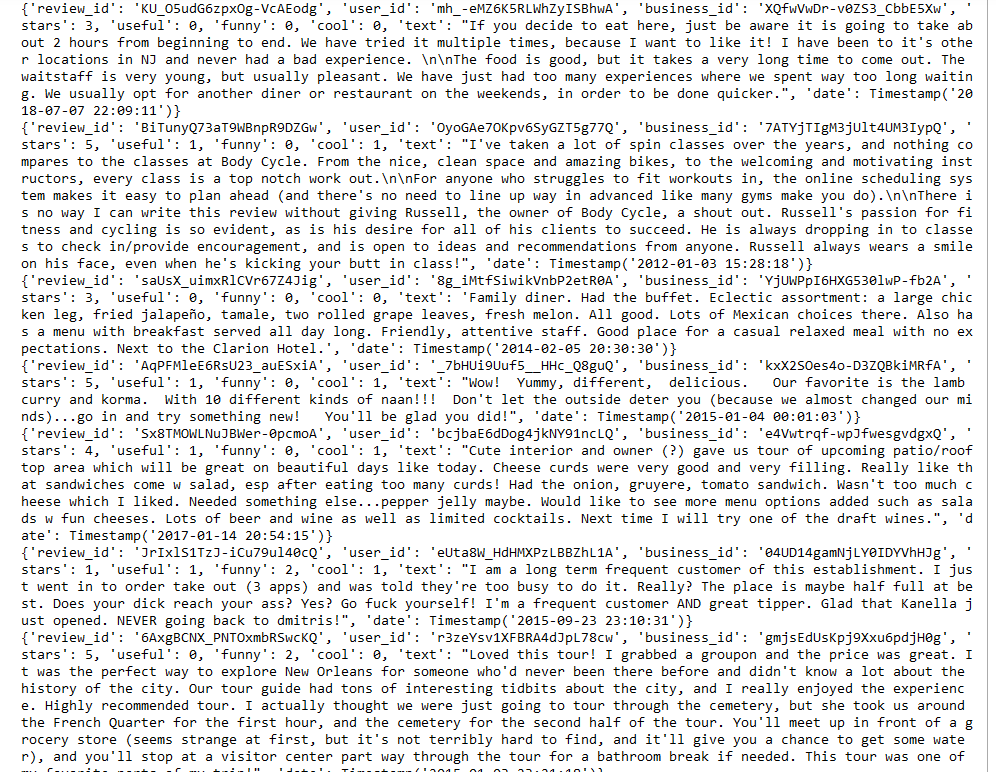
A screenshot of a computer

Description automatically generated

After performing the basic set up of kafka next step is to load the data. For doing the same below are the steps followed in the code ,

1. Import all the necessary library
2. Read the data from the shared json file
3. Then Kafka producer logic has been implemented for sending the json data into kafka server allowing to use by any consumer
4. Followed by Kafka consumer logic which is ready to accept the messages sent by producer.

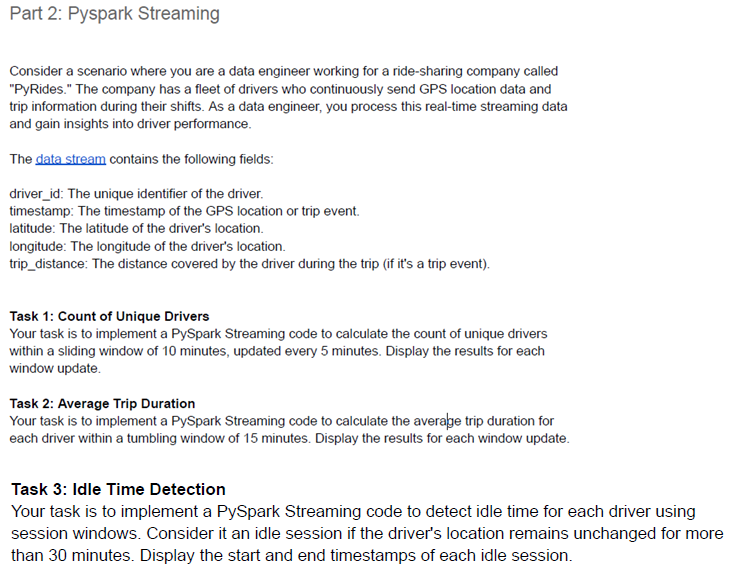
**Output : Kafka producer sending message:**



**Kafka consumer sending message:**

A close up of a text

Description automatically generated



PySpark Streaming constitutes a key component within the Apache Spark ecosystem, facilitating the manipulation and analysis of real-time streaming data. This component furnishes an abstraction layer tailored for the management of continuous data streams, granting developers the ability to engage with streaming data through the familiar APIs of Spark.

* **Advantages of PySpark Streaming:**
  + **Unified API**:PySpark Streaming seamlessly integrates with the Spark ecosystem, enabling users to harness their existing proficiency in Spark's RDD (Resilient Distributed Dataset) transformations and actions for real-time data processing.
  + **High-level Abstractions:** Within PySpark Streaming, elevated abstractions like DStreams (Discretized Streams) mitigate the intricacies of managing streaming data, allowing developers to channel their focus into the core business logic.
  + **Fault Tolerance:** Similar to other Spark elements, PySpark Streaming ensures fault tolerance via lineage information. This characteristic facilitates the recovery of lost data that might arise from node failures.
  + **Scalability:** PySpark Streaming scales adeptly to accommodate substantial streams of real-time data by capitalizing on Spark's capability for distributed computing.
  + **Windowing Support:** The framework encompasses support for assorted windowing operations, including sliding and tumbling windows, which enable the processing of data within designated time intervals.
  + **Wide-ranging Data Source Compatibility:** PySpark Streaming adeptly ingests data from a variety of sources, encompassing Kafka, Flume, HDFS, S3, and more. This versatility renders it suitable for integration with diverse data pipelines.
  + **Seamless Batch Processing Integration:** PySpark Streaming seamlessly harmonizes with batch processing within the Spark environment. This seamless integration permits users to amalgamate real-time and batch processing for holistic data analysis.
* **Drawbacks of PySpark Streaming:**
  + **Micro-Batching Model:** Operating under a micro-batching paradigm, PySpark Streaming processes data in small batches, leading to a certain degree of latency as data accrues over intervals.
  + **Latency:** The micro-batching approach inevitably incurs higher latency in contrast to authentic real-time processing systems that react to event-triggered updates.
  + **Resource Management Overhead:** Navigating a streaming application mandates resource upkeep for both the streaming application itself and the Spark cluster, potentially adding complexity to operational management.
  + **Challenges in Low-latency Scenarios:** PySpark Streaming might not be the most fitting choice for use cases demanding exceedingly low-latency processing, where minimal delay between data arrival and processing is imperative.
  + **Learning Curve:** Despite abstracting complexities tied to real-time data processing, PySpark Streaming necessitates users to acquaint themselves with its concepts and APIs, entailing some initial learning effort.
  + **Resource Consumption:** Given Spark's resource-intensive nature, orchestrating and maintaining a Spark cluster for streaming duties demands significant resources and infrastructure support.

**Pyride data information:**

* "driver\_id": This identifier is exclusive to each driver. In this scenario, both journeys were carried out by the driver with the ID "D001."
* "timestamp": This attribute denotes the time when the trip event was recorded. The timestamp appears to adhere to the Unix time format, which quantifies the seconds elapsed since 00:00:00 on Thursday, 1 January 1970, Coordinated Universal Time (UTC), accounting for leap seconds.
* "latitude" and "longitude": These attributes signify the geographical coordinates of the driver when the event was logged. They indicate the driver's position on the Earth's surface.
* "trip\_distance": This metric quantifies the distance covered in the trip, measured in units such as miles or kilometers.
* "event\_type": This attribute describes the nature of the event. In this instance, it is labeled as "Trip," implying a connection to a driver's journey. Additionally, the "GPS" label suggests that the driver is either inactive or not currently engaged in a trip.

**Output of each task :**

**Task 1 :**

A screenshot of a computer

Description automatically generated

**Task 2 :**

A screenshot of a computer

Description automatically generated

**Task 3: There are no idle drivers available for 30 min of duration.**

A screen shot of a computer program

Description automatically generated

**Conclusion:**

|  |  |
| --- | --- |
| Task | Conclusive Statement |
| Task 1 | The first stage in this operation is to collect information for all drivers functioning inside the supplied timestamp range. The dataset is reviewed in order to find unique driver identifiers. A careful Excel analysis revealed that a total of 47 unique driver IDs were captured between the hours of 5:00 AM and 5:15 AM. This observation corresponds exactly with the code's findings. The methodical methodology provides precise identification of drivers and their respective activity during the time span specified. |
| Task 2 | The task entails calculating precise journey lengths. A comprehensive technique was used, considering both "Trip" and "GPS" event kinds. This choice sought to establish successive timestamps that are critical for distinguishing between trips and idle intervals. Following the calculation of time durations for each event, the effort switched to identifying "Trip" event entries. This enabled a clear correlation between entries and their associated journey lengths. The addition of a 15-minute tumbling window method improved the study. The ability to group trip data in such frames allowed for simple averaging, allowing the estimation of average trip durations. This method preserves timestamp integrity while providing information regarding drivers' journey habits. |
| Task 3 | This task interprets the dataset as driver-related data, differentiating between non-trip ("GPS") and trip ("Trip") states. The purpose is to identify periods of inactivity or idle time in the driver as represented by "GPS" events. To remedy this, event durations are calculated, and instances that surpass the 30-minute threshold are labelled as idle time. This technique correctly recognises periods of idleness while rejecting events that fall below a certain threshold. The analysis was done to driver D001, demonstrating that idle times were successfully detected. The congruence of manual analysis and code-based discoveries confirms the approach's success. This method provides information about drivers' activities and idle times. |