## Contents

## Requirements:

1. **Real-time Visualization**: Dashboard to visualize top items detected by video cameras in different geographical locations.
2. **Low Latency**: Results should be available as soon as events are published.
3. **Handling Duplicates**: Account for duplicate events that may occur upstream.

## Overall Architecture:

1. **Edge Devices (Video Cameras):**
   - Event data is generated from video cameras deployed at various geographical locations.
   - Each event represents an item detected by the camera and includes geographical_location_oid, item_name, timestamp, video_camera_oid, detection_oid and timestamp_detected
2. **Data Ingestion:**
   - Event data is ingested via Apache Kafka.
   - Buffers the data and makes it available for downstream processing.
3. **Stream Processing:**
   - Data is processed in real-time/ near-real-time using Spark Structured Streaming.
   - Duplicate event handling of items and grouping by item_name and geographical_location_oid in Dataset A will be handled before joining with Dataset B based on geographical_lcation_oid
   - Processed data will then be transformed into a structured format suitable for storage.
4. **Data Storage:**
   - Processed data is stored in Hadoop Distributed File System (HDFS) while Dataset B which is a reference table can be stored in S3.
   - The processed data is partitioned and organized into directories based on geographical_location and time period.
5. **Data Access and Analytics:**
   - Data stored in Hadoop is accessible to analytical tools and frameworks for further analysis.

- Spark SQL query can be used fetch and extract the items in each location, aggregate the items to get item_count and rank the items based on their counts.

6. **Dashboard Visualization:**

    - Query results can be visualized using a dashboarding tool such as Tableau, or Power BI.
    - Other analytics can also be integrated via other SQL queries, and users can use the dashboard for filtering etc.

## Data Ingestion:

**Technology Stack:** Apache Kafka
**Justification:** Highly scalable. Kafka is designed to scale horizontally and would be able to handle the large volumes of streaming data from video cameras efficiently. It can handle thousands of messages per second across multiple partitions and brokers which fits the volume of data as specified. It has built-in fault tolerance and offers low-latency message delivery, making it suitable for real-time or near-real-time streaming uses. It also has exactly-once message delivery semantics, ensuring that each message is processed exactly once by consumers, preventing further duplication of event data.
**Approach:**
Each video camera has a kafka producer that will write each event to a kafka topic and every message body will contain the five fields required in Dataset A. Ideally, each geographical location can have their own topic (Eg. cameras from X location will write to topic: raw_camera_events_x). This would help segregate data based on their locations and reduce filtering based on location in the stream processing portion. However, this might spawn an infinite amount of kafka topics which may become unmanageable if the number of locations scales. Else, they could all write to a single topic (raw_camera_events).
The Kafka cluster must be appropriately sized due to high message volume. Scaling strategies must be in place to dynamically adjust the Kafka cluster's resources to handle any increase in throughput. We can increase the number of brokers in the cluster as well as allocate more memory/CPU. The topic must be partitioned to distribute the load across multiple brokers and improve parallelism and there should be large number of partitions.
We could also set up separate kafka topics for monitoring and alerting systems (detection of anomalies, performance issues and error output)**.** There should also be retention policies for the Kafka topic to manage storage costs and ensure compliance with data retention regulations. There is also need for periodic cleanup of old data to prevent the Kafka cluster from running out of storage.

The stream processing portion (Spark) can directly consume from the kafka topic.

## Streaming Data Processing:

**Technology Stack:** Apache Spark Structured Streaming and Dataframe API
**Justification:**
Provides a high-level API for stream processing with built-in fault tolerance and scalability, allowing for the continuous processing of data with low-latency, making it suitable for real-time analytics on streaming data. It seamlessly integrates with batch processing, allowing for unified processing logic for both streaming and batch data. It also provides built-in support for handling duplicate data by maintaining state and processing event-time or processing-time windows. It is also designed for horizontal scalability, allowing for distributed processing of data across multiple nodes in a cluster. This ensures that it would be able to handle a sudden influx of data in case of data skew/ duplicated events.
**Approach:**
Each step of the data processing portion can be broken down and containerised into separate docker images. Within each container, there will be a kafka consumer to listen to messages published by the previous step. Isolating each portion of the pipeline in separate containers ensures that changes or updates to one component (Eg. fields to aggregate/ join strategy for files) do not impact the others, enhancing maintainability and scalability. Each component can

also independently publish and consume data from Kafka topics, enabling asynchronous communication and loose coupling.

Components:

1. Processing Dataset A: Checks for duplicates based on detection_oid. Duplicate events are filtered out or aggregated to ensure that only unique events are processed further. Use windowing functions or sliding windows to group events by a time interval (e.g., every minute) and location (Assuming there are many events per location id, else grouping by time interval might be sufficient). Calculate item counts within each window, which reduces the amount of data written to storage (compared to writing every event to storge)

2. Joining with Dataset B based on geographical_location_oid: Using Spark Dataframe API, join processed data from Dataset A with Dataset B (static reference table) based on the geographical_location_oid. This should be done in smaller batches to reduce the retrieval of the Dataset B and also allows the results of the joins to be quickly available.

3. Transforming Processed Data into Structured Format: This transformation involves mapping the data to a predefined schema and organizing it into a structured format such as Parquet files or structured directories. A common schema is essential for any downstream analysis and retrieval of data via queries.

4. Storage and Output**:** The transformed and structured data is stored in HDFS. It can be partitioned and organized for efficient storage and retrieval based on geographical location or timestamp. The stored data serves as an accessible repository for downstream analysis via dashboards. The processed data can be written to HDFS via the Dataframe API.

## Data Storage:

**Technology Stack:** Apache Hadoop Distributed File System (HDFS) for Dataset A, and Amazon S3 for Dataset B.
**Justification:**
HDFS is well-suited for storing large volumes of streaming data generated by video cameras in Dataset A. It provides high throughput and low-latency access, making it ideal for handling continuous streams of data. HDFS also co-locates data with computation, allowing data processing frameworks like Apache Spark to leverage data locality for efficient processing. Spark has native integration with HDFS, and the processed data can be written to HDFS via the Dataframe API. This enables efficient storage and retrieval of processed data from HDFS.

Amazon S3: Dataset B, being a static reference table, may not require the real-time access and fault tolerance features provided by HDFS. Amazon S3 provides a pay-as-you-go pricing model, allowing users to store large volumes of data at a lower cost compared to maintaining HDFS clusters. This is advantageous for storing reference data that is accessed less frequently and does not require real-time processing. It also allows for easy integration with Apache Spark and other processing frameworks.

**Approach:**

The processed data from dataset A would be stored as files in HDFS. Each file may represent a time window or a partitioned subset of data based on geographical location or timestamp. Dataset B would be stored an objects in Amazon S3 buckets, representing a file containing geographical location mappings.

## Dashboard Visualization:

**Technology Stack**: Tableau/ Power BI.
**Justification:**
Simple to use tool for users to interact with data and customisable to their needs. If the analysis required by users can be sufficiently provided via these tools, they already come with in built methods to connect with our databases.
**Approach:**
Configure the dashboarding tool to connect to the HDFS cluster where the processed results from Dataset A is stored. Utilize connectors or drivers provided by the dashboarding tool to establish a connection to HDFS and access the stored data directly. SQL-like queries can be used to query the structured data stored in HDFS. For example, HDFS stores the item_count for each location in a window timeframe. A query can be written to retrieve all the data from a specific location and rank the top items to be displayed on the dashboard. Other data can be retrieved based on user-defined filters, parameters, or criteria results visualised using charts and graphs.

## Other questions to ask the users:

1. How sensitive is the data? Determines need for implementation of appropriate security measures to protect sensitive data, including encryption in transit and at rest
2. Do we need user access controls such as to provide different levels of access for different user groups (Eg. whether a user can only see data from specific locations)
3. How much latency can they tolerate from the data being produced on edge cameras to being displayed on their dashboard. This will affect the refresh rate of data as well as the batch sizes during processing (or if we fully use streaming).