Oppe 2: Stock Analysis Pipeline

Submitted by Shramana Sinha, 23f1002703

Setup

1. Created a gcs bucket for this assignment.

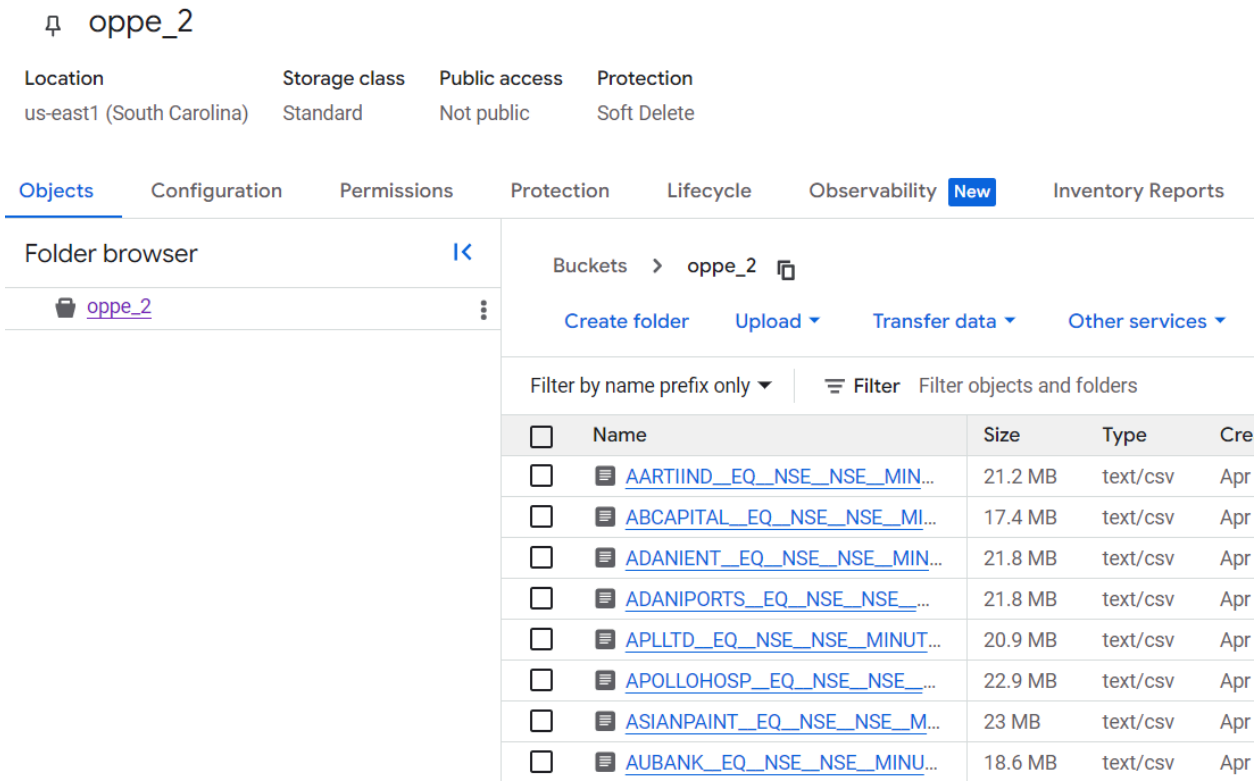

Figure: Google Cloud Storage bucket containing stock data CSV files

2. Spun up a vm to run the zookeeper and kafka servers.

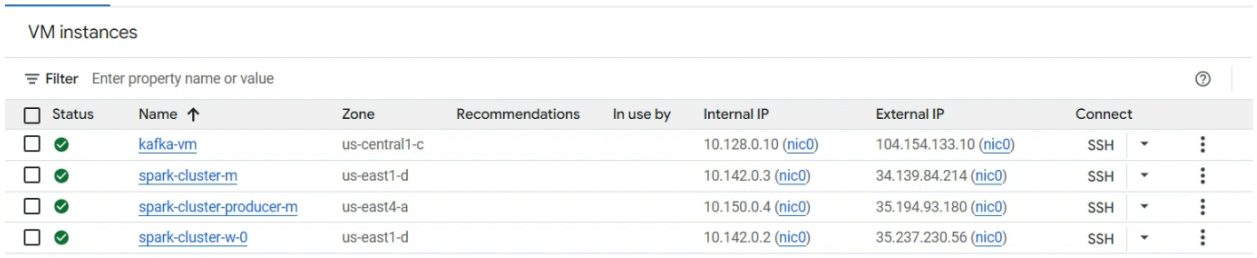

Figure: Kafka VM instance (kafka-vm) hosting the message broker service

3. Spun up 2 dataproc clusters - 1 for producer and 1 for consumer.

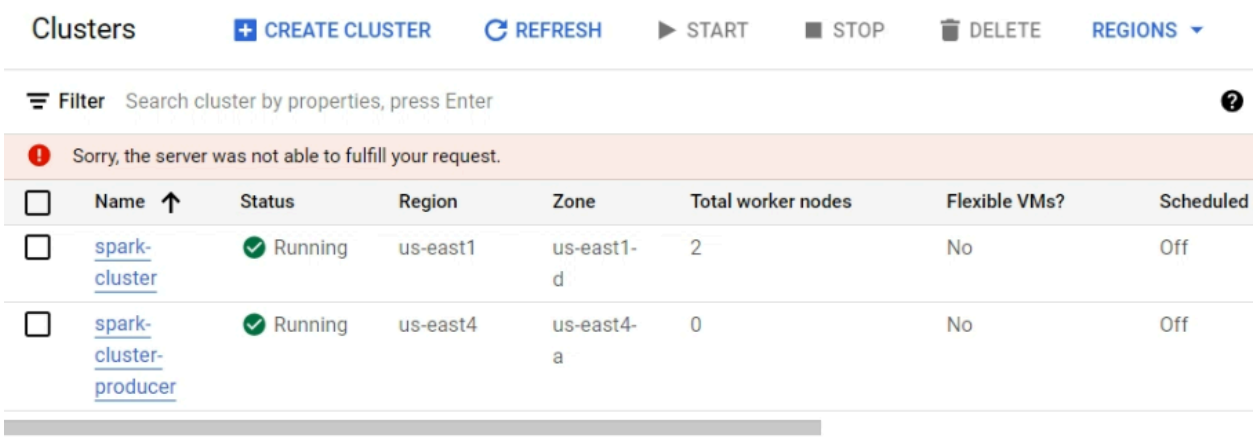

Figure: Dataproc clusters configured for Spark batch and streaming processing

4. Created a google pub/sub topic.

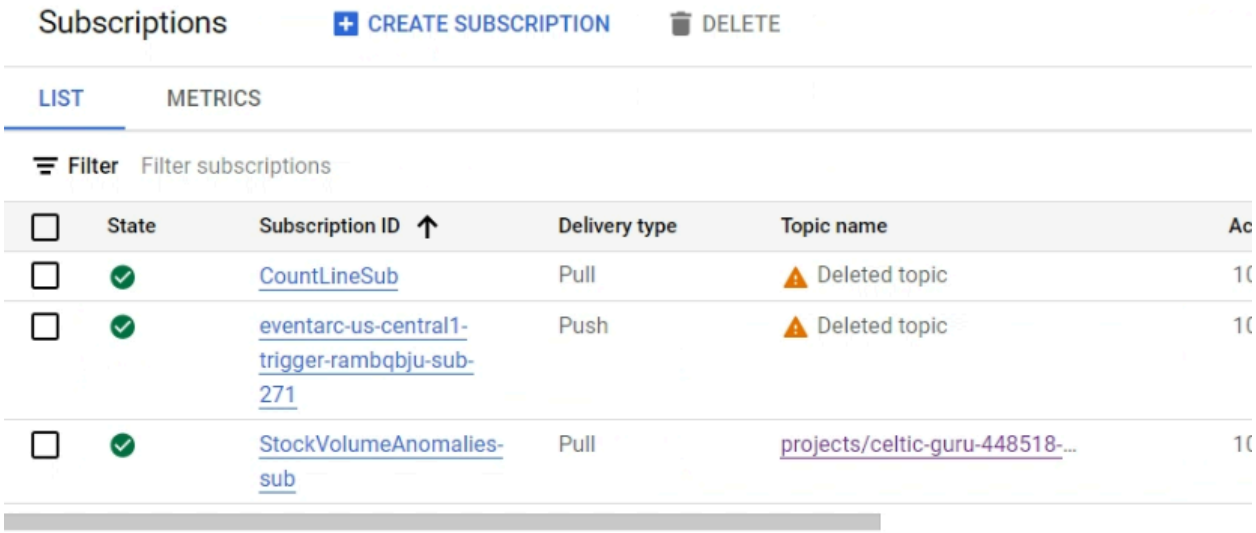

Figure: Google PubSub topic "StockVolumeAnomalies" for anomaly notifications

5. Downloaded the files from the google drive folder to the google cloud shell using the below code:

```python
import os
from google.oauth2 import service_account
from googleapiclient.discovery import build
from googleapiclient.http import MediaIoBaseDownload
import io
folder_id = "1bt94v6pTR6s58c8HFX1FU_9zMoXFX4cf"
credentials = service_account.Credentials.from_service_account_file(
    'service-account-key.json',
    scopes=['https://www.googleapis.com/auth/drive.readonly']
```

```
)
service = build('drive', 'v3', credentials=credentials)
def download_folder(folder_id, output_dir='.'):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    results = service.files().list(
        q=f"'{folder_id}' in parents",
        fields="files(id, name, mimeType)").execute()
    items = results.get('files', [])
    if not items:
        print(f'No files found in folder {folder_id}')
        return
    for item in items:
        if item['mimeType'] == 'application/vnd.google-apps.folder':
            subfolder_path = os.path.join(output_dir, item['name'])
            print(f"Processing subfolder: {item['name']}")
            download_folder(item['id'], subfolder_path)
        else:
            print(f"Downloading file: {item['name']}")
            request = service.files().get_media(fileId=item['id'])

            file_path = os.path.join(output_dir, item['name'])
            with io.FileIO(file_path, 'wb') as f:
                downloader = MediaIoBaseDownload(f, request)
                done = False
                while not done:
                    status, done = downloader.next_chunk()
                    print(f"Download {int(status.progress() * 100)}%")
download_folder(folder_id, 'downloaded_files')
```

6. Uploaded the csv files from the google cloud shell local storage to the gcs bucket.

Implementation Details

Batch Processing Implementation

1. Data Loading:

- Reads from GCS using the path pattern `gs://oppe_2/*.csv`

```
Python
gcs_path = "gs://oppe_2/*.csv"
```

- Uses Spark's CSV reader with `inferSchema` to automatically detect data types

```python
df = spark.read.option("header", "true").option("inferSchema",
"true").csv(gcs_path)
```

2. Data Cleaning:

- Filters out rows with null values in the columns using boolean expressions

```python
df_clean = df.filter(
    col("timestamp").isNotNull()
    & col("close").isNotNull()
    & col("open").isNotNull()
    & col("high").isNotNull()
    & col("low").isNotNull()
    & col("volume").isNotNull()
)
```

- Explicitly casts columns to appropriate types to ensure data consistency

```python
df_clean = (
    df_clean.withColumn("timestamp", col("timestamp").cast("timestamp"))
    .withColumn("close", col("close").cast("double"))
    .withColumn("open", col("open").cast("double"))
    .withColumn("high", col("high").cast("double"))
    .withColumn("low", col("low").cast("double"))
    .withColumn("volume", col("volume").cast("long"))
)
```

3. Data Enrichment:

- Uses `input_file_name()` function to extract the source filename. This allows tracking which file each stock record came from.

```python
df_clean = df_clean.withColumn("stock_ticker", F.input_file_name())
```

4. Data Sorting:

- Orders data by stock ticker and timestamp to ensure time-series analysis accuracy

```python
df_sorted = df_clean.orderBy("stock_ticker", "timestamp")
```

5. Kafka Integration:

- Uses `selectExpr` to create the key-value structure needed for Kafka
- Serializes the entire row structure as JSON using `to_json(struct(*))`
- Uses stock ticker as the Kafka message key for partitioning
- Uses Spark's Kafka connector to write directly to the topic

```python
df_sorted.selectExpr(
    "stock_ticker AS key",
    "to_json(struct(*)) AS value"
).write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "104.154.133.10:9092") \
    .option("topic", "stock-trades") \
    .save()
```

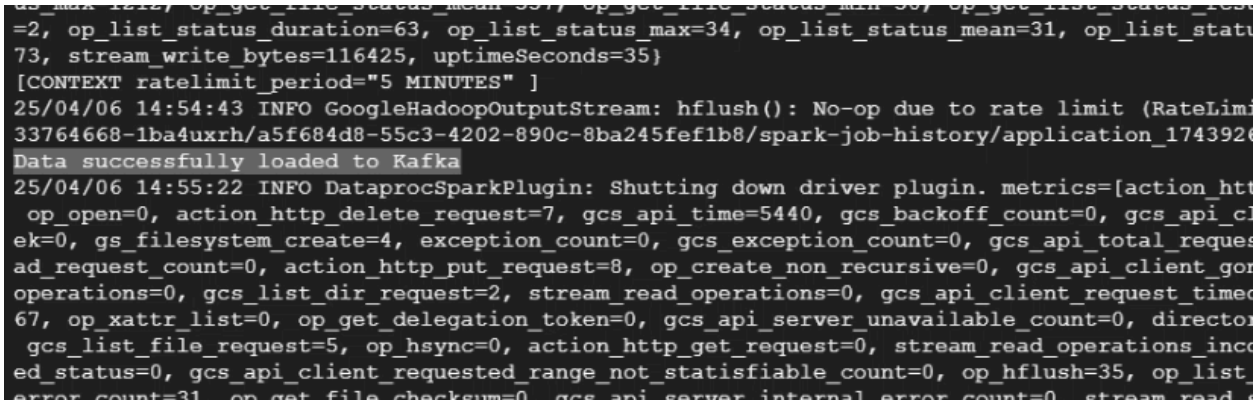

Figure: Producer job execution showing successful data loading to Kafka

Figure: Kafka topic verification showing successful message ingestion from the producer

Streaming Analytics Implementation

1. Configuration and Setup

- Creates Spark session and configure it with graceful shutdown to properly close resources and optimize the shuffle partitions for better parallelism

```python
spark = SparkSession \
    .builder \
    .appName("StockAnomalyDetection-Improved") \
    .config("spark.streaming.stopGracefullyOnShutdown", "true") \
    .config("spark.sql.shuffle.partitions", "8") \
    .getOrCreate()
```

- Defines a strict schema to enforce data types during JSON parsing

```python
stock_schema = StructType([
    StructField("stock_ticker", StringType(), True),
    StructField("timestamp", TimestampType(), True),
    StructField("open", DoubleType(), True),
    StructField("high", DoubleType(), True),
    StructField("low", DoubleType(), True),
    StructField("close", DoubleType(), True),
    StructField("volume", LongType(), True)
])
```

- Initializes Google PubSub client for publishing alerts

```python
PROJECT_ID = "celtic-guru-448518-f8"
TOPIC_NAME = "StockVolumeAnomalies"
publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path(PROJECT_ID, TOPIC_NAME)
```

2. Stream Reading and Parsing

- Reads from Kafka using Structured Streaming with configuration for:
    - Starting from earliest offset to process all historical data
    - Limiting each trigger to 500 records for controlled batch sizes

```python
raw_stream = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "104.154.133.10:9092") \
    .option("subscribe", "stock-trades") \
    .option("startingOffsets", "earliest") \
    .option("maxOffsetsPerTrigger", 500) \
    .load()
```

- Parses JSON data with explicit schema for type safety

```python
parsed_stream = raw_stream \
    .select(
        col("key").cast("string"),
        from_json(col("value").cast("string"), stock_schema).alias("data")
    ) \
    .select("key", "data.*")
```

- Applies a 15-minute watermark to handle late-arriving data

```python
watermarked_stream = parsed_stream \
    .withWatermark("timestamp", "15 minutes")
```

3. PubSub Publishing Function

- Creates a structured message with relevant anomaly details
- Converts to JSON and encodes as bytes for PubSub compatibility
- Returns the publish future for tracking completion

```python
def publish_to_pubsub(stock_ticker, timestamp, volume, avg_volume,
volume_change_pct):
    message = {
        "stock_ticker": stock_ticker,
        "timestamp": str(timestamp),
        "volume": volume,
        "avg_volume": avg_volume,
        "volume_change_pct": volume_change_pct,
        "warning": f"Traded Volume more than 2% of its average"
    }
```

```
    message_data = json.dumps(message).encode('utf-8')
    future = publisher.publish(topic_path, data=message_data)
    return future
```

4. Batch Processing Function with Anomaly Detection

- Performs early exit for empty batches to save processing time

```
Python
if batch_df.count() == 0:
        return
```

- Uses window functions to calculate:
    - Previous minute's close price using `lag`

```
Python
windowSpec = Window.partitionBy("stock_ticker").orderBy("timestamp")
with_prev_close = stock_data \
            .withColumn("prev_minute_close", lag("close", 1).over(windowSpec))
```

    - 10-minute average volume using avg

```
Python
volume_window = Window.partitionBy("stock_ticker") \
                            .orderBy("timestamp") \
                            .rowsBetween(-10, -1)
with_metrics = with_prev_close \
            .withColumn("avg_10min_volume", avg("volume").over(volume_window))
```

- Implements anomaly detection logic with multiple conditions (discussed in the section below)
- Collects volume anomalies for PubSub publishing

```
Python
volume_anomalies = anomalies.filter(col("is_volume_anomaly") == True)
if volume_anomalies.count() > 0:
     print(f"Publishing {volume_anomalies.count()} volume anomalies to PubSub")
     volume_anomalies_list = volume_anomalies.select(
                "stock_ticker", "timestamp", "volume",
                "avg_10min_volume", "volume_change_pct"
```

```
            ).collect()
        for row in volume_anomalies_list:
                publish_to_pubsub(
                    row["stock_ticker"],
                    row["timestamp"],
                    row["volume"],
                    row["avg_10min_volume"],
                    row["volume_change_pct"]
                )
```

- Provides detailed console output of all the anomalies for monitoring

```
Python
if anomalies.count() > 0:
            print(f"Batch ID: {batch_id} - Found {anomalies.count()}
anomalies")
            anomalies.select(
                "stock_ticker", "timestamp", "close", "volume",
"prev_minute_close",
                "avg_10min_volume", "price_change_pct", "volume_change_pct",
"anomaly_type"
            ).show(truncate=False)
```

5. Stream Processing Execution

- Uses `foreachBatch` to apply the custom processing function, discussed above.
- Sets processing time trigger of 1 minute for regular batch execution
- Uses "update" output mode to process only new/updated data

```
Python
streaming_query = watermarked_stream \
    .writeStream \
    .foreachBatch(process_batch) \
    .outputMode("update") \
    .trigger(processingTime="1 minute") \
    .start()
```

- Waits for termination to keep the application running

```
Python
streaming_query.awaitTermination()
```

Anomaly Detection Implementation Breakdown

1. **Price Change Calculation**:
    - Calculates percentage change from previous minute
    - Handles null values with default of 0

```python
.withColumn("price_change_pct",
                        when(col("prev_minute_close").isNotNull(),
                             (col("close") - col("prev_minute_close")) /
col("prev_minute_close") * 100)
.otherwise(lit(0)))
```

2. **Volume Change Calculation**:
    - Calculates percentage change from 10-minute average
    - Includes check for positive average to avoid division by zero
    - Handles null values with default of 0

```python
.withColumn("volume_change_pct",
                        when(col("avg_10min_volume").isNotNull() &
(col("avg_10min_volume") > 0),
                             (col("volume") - col("avg_10min_volume")) /
col("avg_10min_volume") * 100)
                        .otherwise(lit(0)))
```

3. **Anomaly Classification**:
    - Price Anomalies (A1): Flags trades where absolute price change from the previous minute's trade close price (`price_change_pct`) exceeds 0.5%.

```python
.withColumn("is_price_anomaly",
                        when(col("prev_minute_close").isNotNull(),
                             F.abs(col("price_change_pct")) > 0.5)
                        .otherwise(lit(False)))
```

    - Volume Anomalies (A2): Flags trades where volume exceeds average traded volume for the last 10 minutes (`volume_change_pct`) by 2% or more.

```python
.withColumn("is_volume_anomaly",
```

```
                    when(col("avg_10min_volume").isNotNull(),
                         col("volume_change_pct") > 2.0)
                    .otherwise(lit(False)))
```

- Defines the anomaly type of the data as `price`, `volume`, `price+volume` based on which kind of anomaly is true.

```
Python
.withColumn("anomaly_type",
                    when(col("is_price_anomaly") & col("is_volume_anomaly"),
lit("Price+Volume"))
                    .when(col("is_price_anomaly"), lit("Price"))
                    .when(col("is_volume_anomaly"), lit("Volume"))
                    .otherwise(lit("None")))
```

- Adds whether the data is anomaly by checking whether the `is_price_anomaly` or `is_volume_anomaly` column value is true or not and filters out the anomaly data based on it.

```
Python
.withColumn("is_anomaly",
                    col("is_price_anomaly") | col("is_volume_anomaly"))
anomalies = anomaly_df.filter(col("is_anomaly") == True)
```

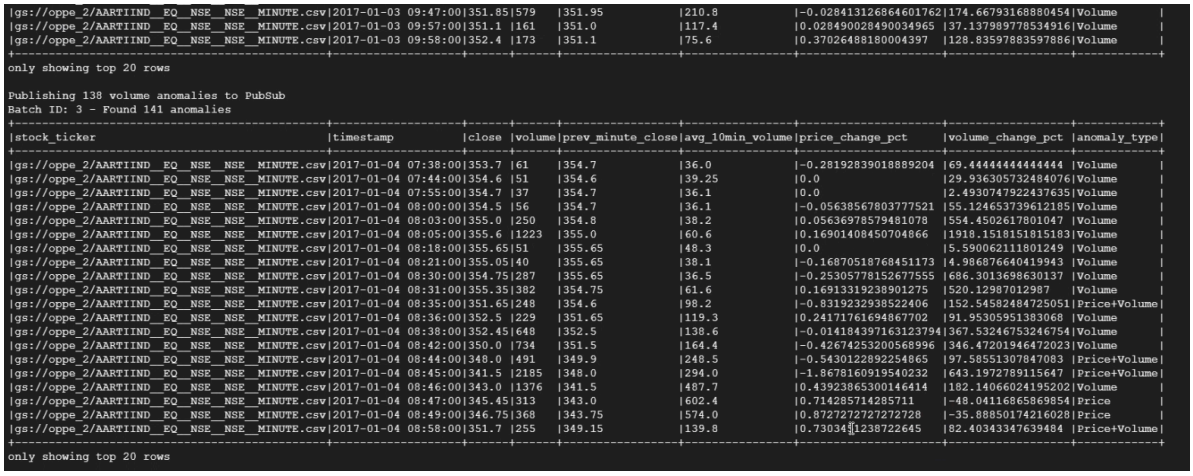

Figure: Consumer streaming execution demonstrating successful anomaly detection

Publish time	Attribute keys	Message body ↑
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE
Apr 6, 2025, 6:10:40 PM	–	{"stock_ticker": "gs://oppe_2/AARTIIND__EQ__NSE__NSE__MINUTE

body.stock_ticker	body.timestamp	body.volume
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 04:42:00	235
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 04:47:00	393
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 04:51:00	819
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 04:53:00	205
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 05:04:00	379
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 05:06:00	1043
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 05:08:00	812
gs://oppe_2/AARTIIND__EQ__NSE__NSE__MI...	2019-03-18 05:09:00	371

body.volume_change_pct	body.warning
9.864422627395976	Traded Volume more than 2% of its average
87.58949880668257	Traded Volume more than 2% of its average
364.5490640952921	Traded Volume more than 2% of its average
8.868826340945294	Traded Volume more than 2% of its average
800.2375296912113	Traded Volume more than 2% of its average
1141.6666666666665	Traded Volume more than 2% of its average
358.23927765237016	Traded Volume more than 2% of its average
45.83333333333333	Traded Volume more than 2% of its average

Figure: PubSub message dashboard showing captured volume anomalies with parsed payload