CSCI-E88C Final Project

Charles Lariviere

2022-12-08

Contents

1 [50 points] Project Proposal and Summary

1.1 Project Goal and Problem Statement

Creative Market is an online marketplace that allows creatives to buy and sell digital design assets, such as fonts, templates, photos, and web themes. Users can browse a catalog of millions of digital assets and purchase licenses that allow them the use of the asset, with prices ranging from a few dollars to a few thousand dollars.

To find assets, users can browse the various curated collections and pages, or use the search engine to find the specific type of asset they are looking for. Given the large size of the catalog, content delivery and personalization is crucial to provide a great user experience and return highly-relevance assets that are likely to meet the user's needs and result in a purchase.

To achieve this goal, the Machine Learning team at Creative Market (which I am a part of) develops multiple systems and models that power all content delivery on the website, with the goal of optimizing relevance and revenue. While there exists evaluation frameworks across some of these initiatives, visibility on the general content delivery performance is limited.

The goal of this project is to create a stream processing application that will visualize multiple performance metrics, based on various user interaction events with the results returned by the various locations on the website. This will provide a robust understanding of the overall performance of the Machine Learning team's initiatives across the various content delivery systems.

1.2 Data Sources

For every search query performed by a user, a set of events are recorded to track the user's interactions with the results. For instance, if a user performs a search for "business card mockups" and receives a grid of 30 results, an event will be recorded for each of the result returned to the user.

Furthermore, as the user browses through the set of results and interacts with them – views, clicks, adds to cart, and purchases – another event is saved for each, recording the product identifier, the user identifier, as well as a unique identifier representing the search query.

These events are sent to a stream in Amazon Kinesis, and then stored in Amazon S3 in batches of compressed JSON files partitioned by type and date using Amazon Kinesis Data Firehose.

Here is an example payload for an event:

Figure 1: ProductView Event

The types of events recorded are:

- Search: Recorded for each search request made by a user, with the payload including the search parameters, filters, and terms.
- Result: Recorded for each product returned in a search result set.
- ProductView: Recorded every time a product included in a search result is viewed (i.e. enters the user's viewport in their browser window).
- ProductClick: Recorded every time a product included in a search result is clicked on.
- ProductAddtoCart: Recorded every time a product included in a search result is added to a user's cart.
- ProductPurchase: Recorded every time a product included in a search result is purchased.

1.3 Expected Results

The output of this stream processing application should be a set of time-series metrics assessing the performance of content delivery across the website, grouped by various dimensions.

Typical evaluation metrics for information retrieval will be used, such as click- through rate, Success Rate, zero-result rate, and Normalized Discounted Cumulative Gain (nDCG). Here are the definitions for each of the metrics that will be computed:

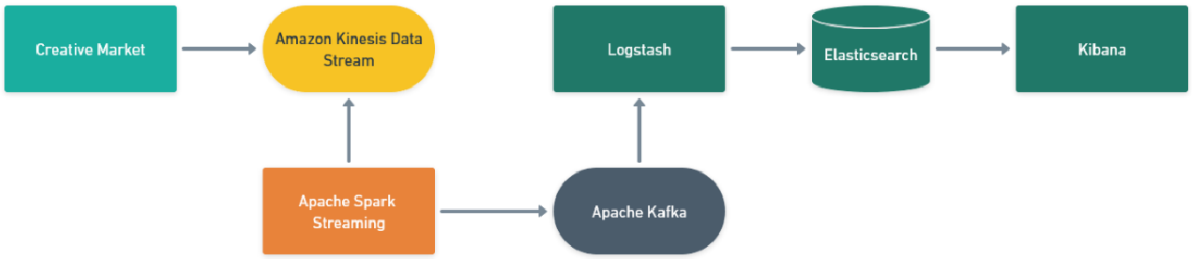
- Click-Through Rate: Ratio of results viewed by a user that were then clicked on.
- Success Rate: Ratio of searches that resulted in a purchase.

- Zero-Result Rate: Ratio of searches that did not return any results.
- nDCG: Score ranging from 0 to 1 that represents how the relevance of actual search results compared to the ideal ranking. Here, click-through rate is used as a proxy for relevance.

The dimensions made available for analysis will include the date, type of page (i.e. Search Engine, Category Page, “You may also like” module), user country, and application version number (i.e. it indicates which ML model and ranking function was used).

These metrics will be stored in Elasticsearch and made available for consumption in Kibana.

1.4 Application Overview and Technologies Used



The streaming application will consume from the stream of events in Amazon Kinesis and leverage Apache Spark Streaming to execute the transformation.

The transformation will be windowed and consist of joining events together on the original result event, to produce a record for each result along with descriptive attributes used to compute the metrics (i.e. isViewed, isClicked, isPurchased).

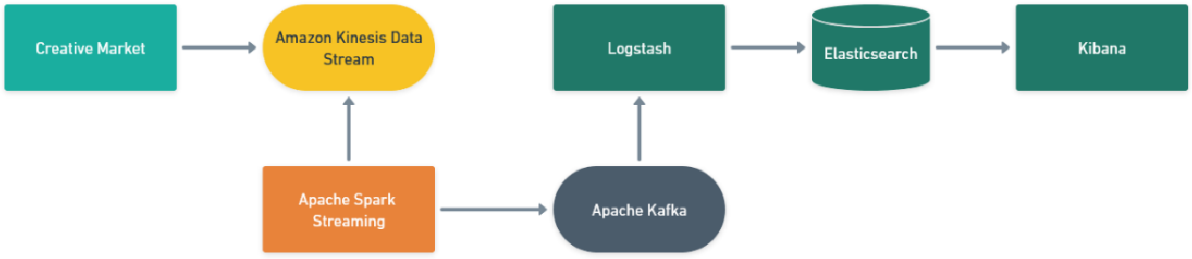
The transformed records will then be written to a Kafka stream which is used to push these to ELK using the Kafka logs integration. Records will be saved in Elasticsearch which will be used to aggregate the metrics and visualize them in Kibana.

1.5 Bonus Options

I will attempt to cover all three bonus options, by writing unit tests on the transformation executed in Spark, executing the streaming application on AWS by consuming from a Kinesis stream, and creating visualizations in Kibana,

2 [50 points] Milestone 1: System Design and Architecture

2.1 System Diagram



2.2 Description of each stage of the processing pipeline

- **Collection**: Events are streamed to Amazon Kinesis by an external system and made available in an Amazon Kinesis Data Stream. Events are made available for consumption within a few milliseconds and are cached for a period of 7 days.
- **Ingestion**: Events will be consumed from the Kinesis data stream using Spark Streaming's Kinesis integration. It allows for obtaining a Dstream from a Kinesis stream by simply providing the necessary configuration.
- **Preparation**: Very limited preparation is needed since the events are formatted as well- formatted JSON objects.
- **Computation**: A multi-stage transformation process is required for this application. Given a stream of ProductSearch, ProductResult, ProductView, ProductClick, ProductAddtoCart, and ProductPurchase, we want to return a set of calculated metrics per window. Computing metrics like Click-Through-Rate, Success Rate, and nDCG will require events to be joined together on the ProductID key, to generate enrich the original ProductResult object with isViewed, isClicked, and isPurchased attributes. Once joined and enriched, metrics such as Click-Through-Rate and Success Rate can be computed per window.
- **Presentation**: The computed metrics will be made available for consumption and visualization by using the ELK stack – Elasticsearch, Logstash, and Kibana. Once computed and logged into the Kafka topic, Logstash's Kafka integration will automatically consume the records and store them into Elasticsearch. These will then be visualized through charts and dashboards using Kibana.

3 [100 points] Milestone 2: Implementation and Results

3.1 a. Code and Configuration Snippets

3.1.1 Configuration

Configuration for the Scala portion of this project is provided through a `.conf` file and Scala case classes.

```
org.cscie88c {
    kafka {
        bootstrap-server = "localhost:9092"
    }
    spark {
        master-url = "local[*]"
    }
    event-generator {
        name = "product-event-generator"
        topic = "product-events"
        searches-per-minute = 6
        view-rate = 0.7
        click-rate = 0.3
        purchase-rate = 0.1
    }
    count-aggregator {
        name = "count-aggregator"
        topic = "product-event-counts"
    }
    zero-result-searches-aggregator {
        name = "zero-result-searches-aggregator"
        topic = "product-zero-result-searches"
    }
}
```

```
final case class KafkaConfig(bootstrapServer: String)
final case class SparkConfig(masterUrl: String)
final case class EventGeneratorConfig(
  name: String,
  topic: String,
  viewRate: Float,
  clickRate: Float,
  purchaseRate: Float,
  searchesPerMinute: Int
)
final case class CountAggregatorConfig(name: String, topic: String)
final case class ZeroResultSearchesAggregatorConfig(name: String, topic: String
    )
```

These configuration case classes are then loaded using `pureconfig` in each object that requires them. For instance;

```
val KAFKA_CONFIG_PATH = "org.cscie88c.kafka"
implicit val kafkaConf: KafkaConfig = ConfigSource.default.at(KAFKA_CONFIG_PATH
    ).loadOrThrow[KafkaConfig]
```

3.1.2 EventGenerator

While, in practice, the events are generated by an external service and made available in an Amazon Kinesis Data Stream, I decided to instead generate synthetic data for the purpose of this project. This synthetic data is written to a Kafka topic that is then consumed by the Spark Structured Streaming applications.

```
object EventGenerator extends App {
  ...
  val producer = new KafkaProducer[String, String](properties)

  // generate an infinite stream of ProductSearch instances
  val searchStream: Stream[ProductSearch] = Stream.continually(ProductSearch.
      create())

  // for each item, wait, generate records, and send to a Kafka topic
  searchStream
    .map(sleep(_, 60000 / eventConf.searchesPerMinute))
    .map(getRecords)
    .foreach(x => x.map(log).foreach(producer.send))

  def getRecords(productSearch: ProductSearch): List[ProducerRecord[String,
      String]] = {
    // each ProductSearch event generates multiple ProductSearch,
    // ProductView, ProductClick, and ProductPurchase events; each
    // with its own configured probability
    val results = createResults(productSearch)
    val views = results.flatMap(_.createView(eventConf.viewRate))
    val clicks = views.flatMap(_.createClick(eventConf.clickRate))
    val purchase = clicks.flatMap(_.createPurchase(eventConf.purchaseRate))

    // concatenate all events
    val events = List(productSearch) ::: views ::: clicks ::: purchase

    // generate an instance of ProductRecord for each event/message
    events.map(_.toMessage.toRecord(eventConf.topic))
  }
  ...
}
```

3.1.3 Events

Each of the events generated by this class are structured as Scala case classes with a companion object used as an instance factory. For example, the `ProductResult` event is implemented as such:

```scala
case class ProductResult(id: String, searchId: String, productId: String, rank: Int, recordedAt: String) extends Messageable {
  override def key: String = this.id
  override def timestamp: String = this.recordedAt
  override def toJson: Json = this.asJson

  def createView(probability: Float): Option[ProductView] = {
    // create a ProductView with a certain probability
    if (Random.nextFloat() < probability) {
      Option(ProductView.create(searchId = this.searchId, productId = this.productId))
    } else {
      None
    }
  }
}

object ProductResult extends Helper {
  def create(searchId: String, rank: Int): ProductResult = {
    // create a ProductResult instance with random attributes
    ProductResult.apply(
      id = randomId,
      searchId = searchId,
      productId = randomId,
      rank = rank,
      recordedAt = currentDate
    )
  }
}
```

This object takes advantage of a few powerful Scala features, such as Option, Trait, and Implicits.

3.1.4 Message / Messageable

To standardize the structure of messages sent to Kafka, and to avoid duplicating logic in each of the multiple `Product*` case classes, a `Message` class and `Messageable` trait were created.

```scala
case class Message(key: String, eventType: String, payload: String, recordedAt: String) {
  def toRecord(topic: String): ProducerRecord[String, String] = {
    new ProducerRecord[String, String](topic, key, this.asJson.noSpaces)
  }
}
```

```scala
trait Messageable {
  def key: String
  def eventType: String = {
    this.getClass.getSimpleName
  }
  def toJson: Json
  def timestamp: String
  def toMessage: Message = {
```

```
    // create an instance of Message from the current class
    Message(
      key = this.key,
      eventType = this.eventType,
      payload = this.toJson.noSpaces,
      recordedAt = this.timestamp
    )
  }
}
```

3.1.5 SparkStreamingAggregator

With the synthetic data available in a Kafka topic, it is now possible to run our Spark Structured Streaming applications. Given that the scope of this project includes multiple types of aggregations, a `SparkStreamingAggregator` Trait was created to faciliate the creation of multiple Spark Structured Streaming applications that rely on Kafka by reducing code duplication.

```
trait SparkStreamingAggregator extends App {
  val SPARK_CONFIG_PATH: String = "org.cscie88c.spark"
  val KAFKA_CONFIG_PATH: String = "org.cscie88c.kafka"

  // load config
  implicit val sparkConfig: SparkConfig = loadConfig[SparkConfig](
      SPARK_CONFIG_PATH)
  implicit val kafkaConfig: KafkaConfig = loadConfig[KafkaConfig](
      KAFKA_CONFIG_PATH)

  // create spark seassion
  implicit val spark: SparkSession = SparkSession
    .builder()
    .master(sparkConfig.masterUrl)
    .getOrCreate()

  // import spark implicits
  import spark.implicits._
  spark.sparkContext.setLogLevel("ERROR")
  ...
```

While this Trait initilizes the required config and `SparkSession`, it also provides a few powerful utility methods that can be used by downstream classes.

For example, it provides a `getEvents` method that returns a `DataFrame` from the Kafka topic that contains our synthetic data, and unpacks the JSON payload using the schema of the `Message` class defined above (i.e. `Encoders.product[Message].schema`)

```
  def getEvents(topic: String)(implicit spark: SparkSession, kafkaConf:
      KafkaConfig): DataFrame = {
    val df = spark
      .readStream
      .format("kafka")
```

```
      .option("kafka.bootstrap.servers", kafkaConf.bootstrapServer)
      .option("subscribe", topic)
      .option("startingOffsets", "earliest") // From starting
      .load()

    // decode key/value into string
    df.select(
        col("key").cast("string"),
        col("value").cast("string")
      )
      // extract JSON payload using Message schema
      .select(
        from_json(
          col("value"),
          Encoders.product[Message].schema
        ).as("parsed")
      )
      // unpack parsed JSON into columns
      .select("parsed.*")
      .withColumn("recordedAt", col("recordedAt").cast("timestamp"))
  }
```

3.1.6 EventTypeAggregator

Given the logic contained in the Trait defined above, our Spark Structured Streaming application that is responsible for the actual business logic is realtively slim. This application counts the number of events by 1-minute window and type of event.

```
object EventTypeAggregator extends SparkStreamingAggregator {
  val COUNT_AGGREGATOR_CONFIG: String = "org.cscie88c.count-aggregator"
  val EVENT_GENERATOR_CONFIG: String = "org.cscie88c.event-generator"
  implicit val config: CountAggregatorConfig = loadConfig[CountAggregatorConfig
      ](COUNT_AGGREGATOR_CONFIG)
  implicit val generatorConfig: EventGeneratorConfig = loadConfig[
      EventGeneratorConfig](EVENT_GENERATOR_CONFIG)

  // get events from Kafka topic
  val events = getEvents(generatorConfig.topic)

  // count by window and eventType
  val counts = events
    .withWatermark("recordedAt", "1 minute")
    .groupBy(
      window(col("recordedAt"), "1 minute"),
      col("eventType")
    )
    .count()

  // format as Kafka messages
  val df = toKafkaDataFrame(counts, col("window").cast("string"))

  // write to Kafka topic
```

```
23    writeKafka(df, config.topic)
24  }
```

3.1.7 ZeroResultSearchesAggregator

The following application filters and parses events of type `ProductSearch` from the topic that contains all types of events, and then counts the number of `ProductSearch` where `results=0`.

```
1  object ZeroResultSearchesAggregator extends SparkStreamingAggregator {
2    val ZERO_RESULT_AGGREGATOR_CONFIG: String = "org.cscie88c.zero-result-
         searches-aggregator"
3    val EVENT_GENERATOR_CONFIG: String = "org.cscie88c.event-generator"
4    implicit val config: ZeroResultSearchesAggregatorConfig = loadConfig[
         ZeroResultSearchesAggregatorConfig](ZERO_RESULT_AGGREGATOR_CONFIG)
5    implicit val generatorConfig: EventGeneratorConfig = loadConfig[
         EventGeneratorConfig](EVENT_GENERATOR_CONFIG)
6
7    // get events from Kafka topic
8    val events = getEvents(generatorConfig.topic)
9    // parse JSON into columns using schema from case class
10   val searches = getEventsOfType[ProductSearch](events)
11
12   val zeroResultSearches = searches
13     .filter(col("results").equalTo(0))
14
15   // count number of rows where 'results' = 0
16   val counts = zeroResultSearches
17     .withWatermark("recordedAt", "1 minute")
18     .groupBy(
19       window(col("recordedAt"), "1 minute"),
20     )
21     .count()
22
23   // write to Kafka topic
24   val df = toKafkaDataFrame(counts, col("window").cast("string"))
25   writeKafka(df, config.topic)
26 }
```

This application takes advantage of the `getEventsOfType` polymorphic method defined in the `SparkStreamingAggregator` tait. This method allows for "automatically" parsing and unpacking a JSON payload into DataFrame columns with the correct types.

```
1    def getEventsOfType[T <: Product : ClassTag : TypeTag](events: DataFrame):
         DataFrame = {
2      events
3        // filter rows where 'type' is equal to the name of the provided class
4        .filter(col("eventType").equalTo(lit(classTag[T].runtimeClass.
             getSimpleName)))
5        // extract JSON payload using T schema
6        .select(
7          from_json(
```

```
            col("payload"),
            Encoders.product[T].schema
          ).as("parsed")
        )
        // unpack parsed JSON into columns
        .select("parsed.*")
        .withColumn("recordedAt", col("recordedAt").cast("timestamp"))
    }
```

3.1.8 Logstash

Finaly, since the Spark streaming applications above write their output to a Kafka stream, we can take advantage of the ELK stack's integration with Kafka to automatically import messages into Elasticsearch using Logstash.

The logstash pipeline that imports messages output by `EventTypeAggregator` is as follows:

```
input {
    kafka {
        codec => json
        bootstrap_servers => "kafka:29092"
        topics => ["product-event-counts"]
    }
}
output {
    elasticsearch {
          hosts => ["elasticsearch:9200"]
          user => "elastic"
          password => "changeme"
          index => "product-event-counts"
      }
}
```

3.2 Screenshots of run commands and processing results

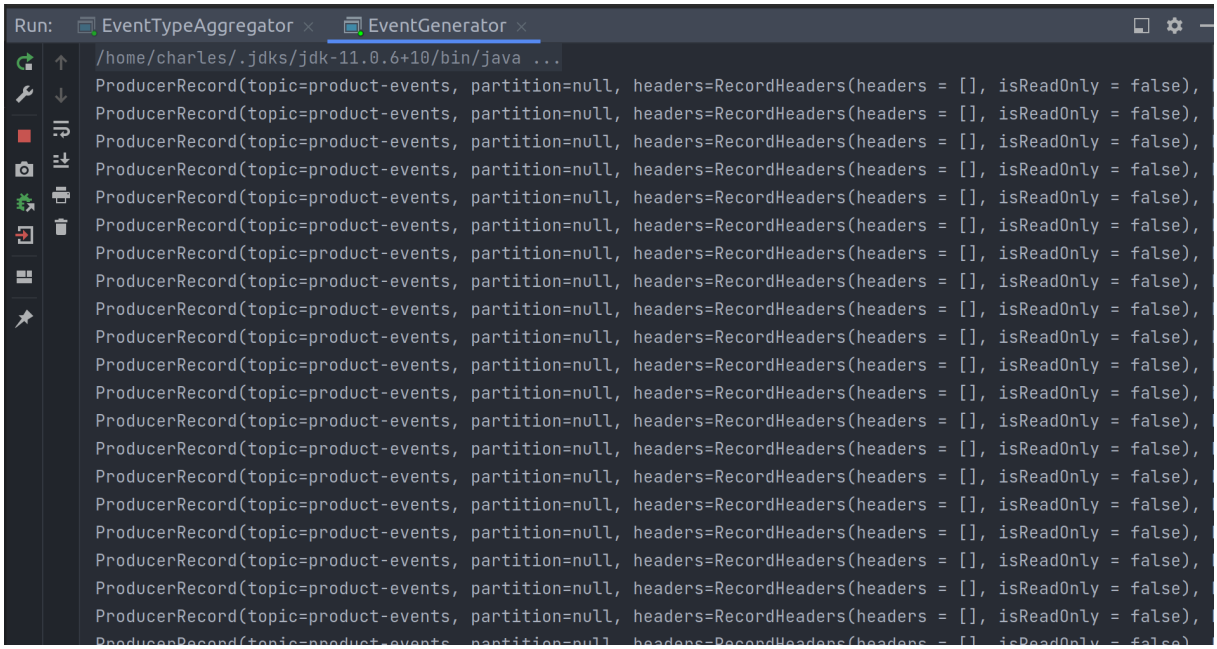


Figure 2: ProducerRecord generated by EventGenerator

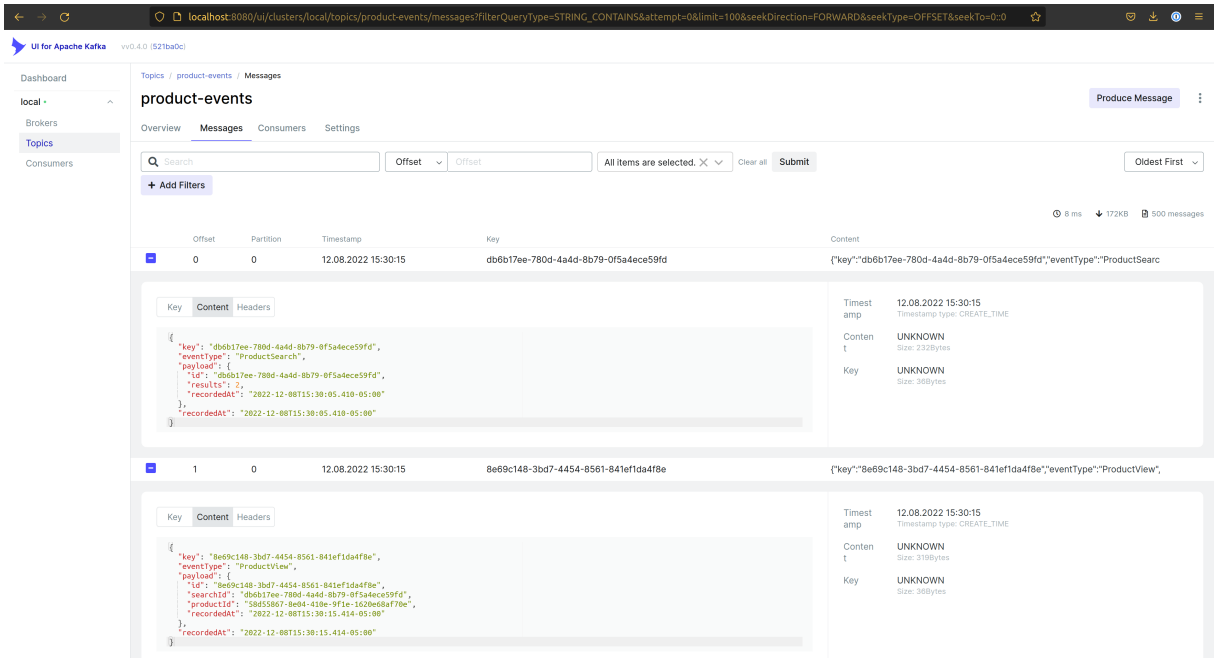


Figure 3: Output of EventGenerator in Kafka

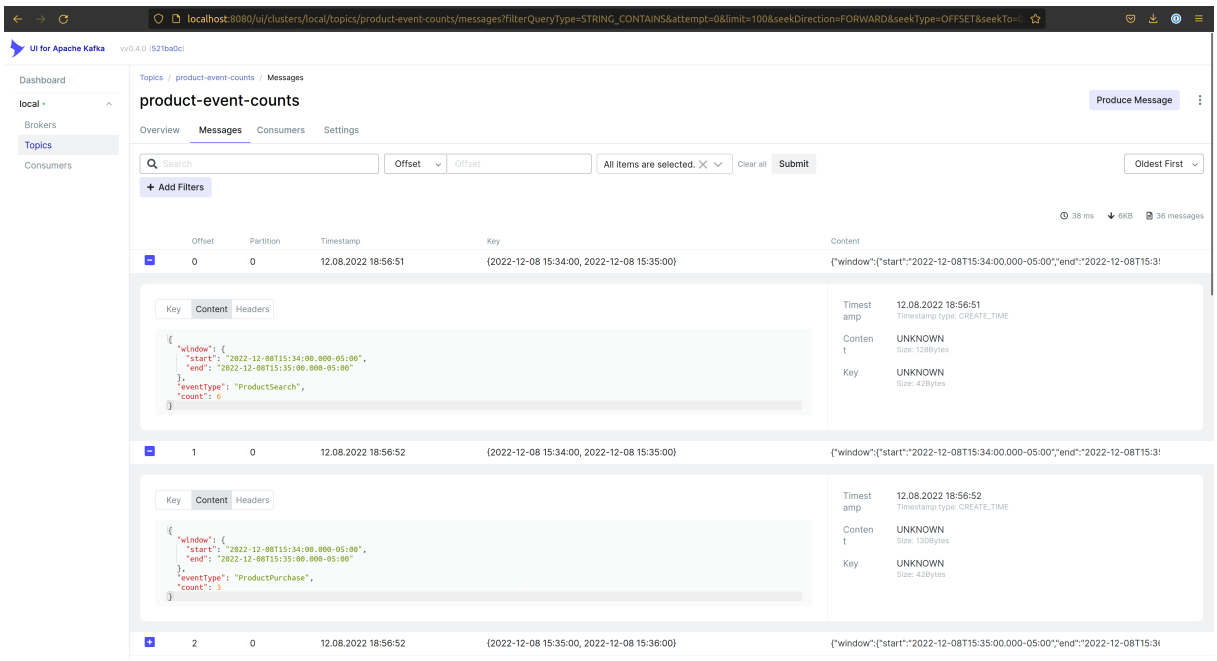


Figure 4: Output of EventTypeAggregator in Kafka

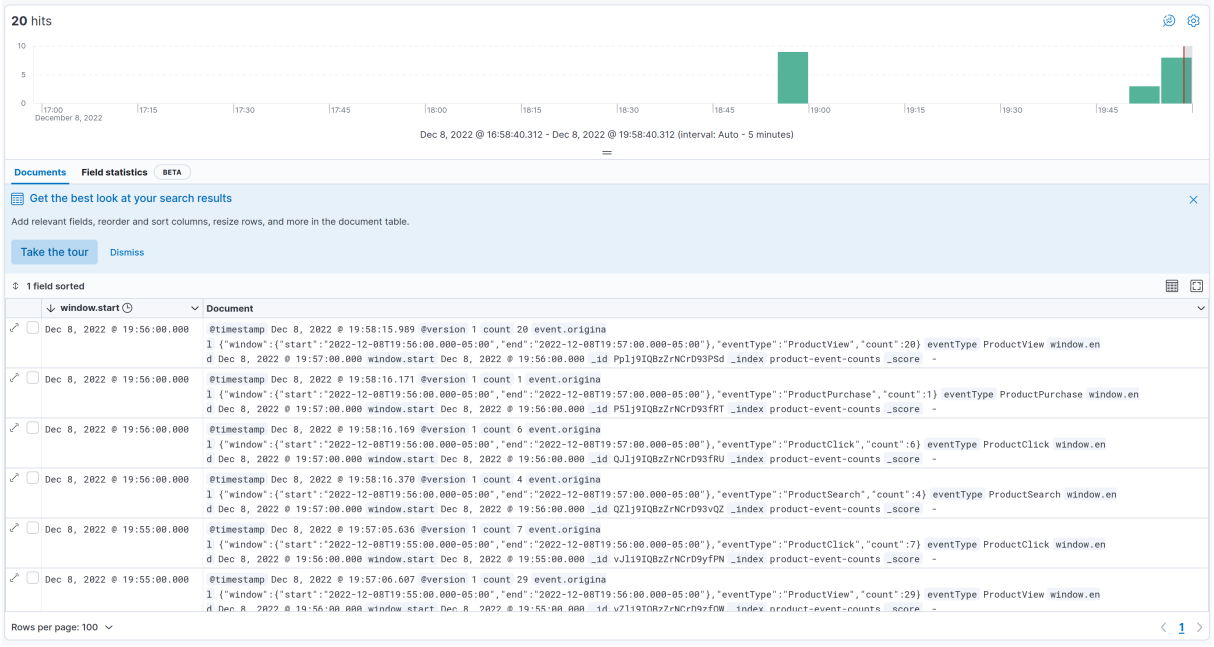


Figure 5: Output from EventTypeAggregator in Kibana