**Harvard University Extension School**

**"Principles of Big Data Processing"**

**CSCI E-88, Fall 2022**

**Final Project**

**by Asma Naeem**

## Project Goal and Problem Statement

This project aims to set up a data processing pipeline for ingesting Twitter streaming data filtered by some criteria and move it into Elasticsearch for visualization and insights.

## YouTube Video URL

<https://youtu.be/ijskuAePeLs>

## Big Data Source

Twitter streaming data.

Describe the data, size, schema and any other details that are needed to understand the data.

|  |
| --- |
| PUT /tweets  {  "mappings": {  "properties": {  "id": {  "type": "long"  },  "tweet": {  "type": "keyword"  },  "createdAt": {  "type": "date"  },  "createdDateHour": {  "type": "date"  },  "location": {  "type": "keyword"  },  "geoLocation": {  "type": "geo\_point"  },  "sentimentType": {  "type": "keyword"  },  "hashTag": {  "type": "keyword"  }  }  }  } |
| **public** **class** TweetEvent {  Long id;  String tweet;  String createdAt;  String createdDateHour;  String location;  String geoLocation; //geopoint  String sentimentType;  String hashTag;    **public** String getTweet() {  **return** tweet;  }  **public** **void** setTweet(String tweet) {  **this**.tweet = tweet;  }  **public** Long getId() {  **return** id;  }  **public** **void** setId(Long id) {  **this**.id = id;  }  **public** String getCreatedAt() {  **return** createdAt;  }  **public** **void** setCreatedAt(String createdAt) {  **this**.createdAt = createdAt;  }  **public** String getCreatedDateHour() {  **return** createdDateHour;  }  **public** **void** setCreatedDateHour(String createdDateHour) {  **this**.createdDateHour = createdDateHour;  }  **public** String getGeoLocation() {  **return** geoLocation;  }  **public** **void** setGeoLocation(String geoLocation) {  **this**.geoLocation = geoLocation;  }  **public** String getLocation() {  **return** location;  }  **public** **void** setLocation(String location) {  **this**.location = location;  }    **public** String getSentimentType() {  **return** sentimentType;  }  **public** **void** setSentimentType(String sentimentType) {  **this**.sentimentType = sentimentType;  }  **public** String getHashTag() {  **return** hashTag;  }  **public** **void** setHashTag(String hashTag) {  **this**.hashTag = hashTag;  }    **public** TweetEvent(Long id, String tweet, String createdAt, String createdDateHour) {  **super**();  **this**.tweet = tweet;  **this**.id = id;  **this**.createdAt = createdAt;  **this**.createdDateHour = createdDateHour;  }  **public** TweetEvent() {  **super**();  }  @Override  **public** String toString() {  **return** "TweetEvent [id=" + id + ", createdAt=" + createdAt + ", tweet=" + tweet + ", createdDateHour=" + createdDateHour  + ", location=" + location+ ", geoLocation=(" + geoLocation + ")]";  }  } |
| The average Tweet size is 28 characters, and the maximum limit is 280 characters. |

## Expected results

The outcome of this processing pipeline is to demonstrate graphs and maps for visualizing tweets data to compare the most popular video-on-demand over-the-top streaming services like Netflix, Amazon Prime Video, Disney Plus, and Hulu. We will pass each tweet message through the StanfordCoreNLP library to conduct sentiment analysis that scores it from 0 to 4 based on whether the analysis comes back with Very Negative, Negative, Neutral, Positive, or Very Positive, respectively.

## Processing Pipeline

Diagram

Description automatically generated

* Collection Tier: We will use Apache Flume distributed, reliable service for collecting and moving large amounts of data.
  + Flume Twitter Data Source fetches Twitter streaming data filtered by video-on-demand service tags like Netflix, Amazon Prime Video, Disney Plus, and Hulu.
* Messaging Tier: Apache Kafka
  + Flume will transport collected data to a Kafka topic for further processing
* Stream Processing Tier: Kafka Elasticsearch Sink Connector
  + Kafka Elasticsearch sink service will write data from a topic in Apache Kafka to an index in Elasticsearch
* Visualization Tier: Kibana source-available data visualization dashboard software for Elasticsearch.
  + Kibana user interface will allow us to explore which video-on-demand services receive the most positive or negative tweets on a given day/ hour.

## Implementation

This is the main section. It should contain all implementation details for each tier. Copy all relevant configuration and code. Code should be well documented.

|  |
| --- |
| **public** **class** FlumeTwitterSource **extends** AbstractSource  **implements** EventDrivenSource, Configurable {    StringBuffer queryCriteria = **new** StringBuffer();  String[] keywords = **null**;    Twitter twitterIns = **null**;    **public** **void** configure(Context cntxt) {    ConfigurationBuilder configBuilder = **new** ConfigurationBuilder();  configBuilder.setOAuthConsumerKey(cntxt.getString(System.*getProperty*(AppConstants.***CONSUMER\_KEY***, AppConstants.***CONSUMER\_KEY\_VAL***)));  configBuilder.setOAuthConsumerSecret(cntxt.getString(AppConstants.***CONSUMER\_SECRET***, AppConstants.***CONSUMER\_SECRET\_VAL***));  configBuilder.setOAuthAccessToken(cntxt.getString(AppConstants.***ACCESS\_TOKEN***, AppConstants.***ACCESS\_TOKEN\_VAL***));  configBuilder.setOAuthAccessTokenSecret(System.*getProperty*(AppConstants.***ACCESS\_TOKEN\_SECRET***, AppConstants.***ACCESS\_TOKEN\_SECRET\_VAL***));  configBuilder.setJSONStoreEnabled(**true**);  keywords = System.*getProperty*(AppConstants.***KEYWORDS***, AppConstants.***KEYWORDS\_VAL***).split(AppConstants.***COMMA***);  **for** (String ht: keywords) {  **if**(queryCriteria.toString().length()>0)queryCriteria.append(AppConstants.***CHAR\_SPACE*** + AppConstants.***OR*** + AppConstants.***CHAR\_SPACE***);  queryCriteria.append(AppConstants.***CHAR\_HASH*** +ht);  }    twitterIns = **new** TwitterFactory(configBuilder.build()).getInstance();    SentimentsAnalysisService.*initialize*();  }  **public** **void** start() {  // The Channel is a passive store that holds the Event until that Event is consumed by a Sink.  **final** ChannelProcessor channelProcessor = getChannelProcessor();  List<Status> tweetsStatus = **null**;    **try** {  ObjectMapper o = **new** ObjectMapper();  Random random = **new** Random();  **int** rand = 0;  Query qry = **null**;  QueryResult rslt;  **do** {  qry = **new** Query(queryCriteria.toString());  **do** {  rslt = twitterIns.search(qry);  tweetsStatus = rslt.getTweets();  **for** (Status tweetStatus: tweetsStatus)  {  LocalDateTime createdAt = LocalDateTime.*ofInstant*(tweetStatus.getCreatedAt().toInstant(),  ZoneId.*systemDefault*());  String createdAt\_fmt = createdAt.format(DateTimeFormatter.***ISO\_LOCAL\_DATE\_TIME***).substring(0,AppConstants.***DATE\_LEN***);  String createdDateHr = (createdAt.format(AppConstants.***HOUR\_FORMATTER***).replace(AppConstants.***HOUR\_FORMAT\_COLON***,  AppConstants.***ISO\_HOUR\_FORMAT\_T***)+AppConstants.***ISO\_HOUR\_FORMAT\_APPEND***).substring(0, AppConstants.***DATE\_LEN***);  TweetEvent te = **new** TweetEvent(tweetStatus.getId(), tweetStatus.getText(), createdAt\_fmt, createdDateHr);  **if**(tweetStatus.getPlace()!=**null**)te.setLocation(tweetStatus.getUser().getLocation());  **if**(tweetStatus.getGeoLocation()!=**null**)te.setGeoLocation(tweetStatus.getGeoLocation().getLatitude()  +AppConstants.***COMMA***+tweetStatus.getGeoLocation().getLatitude());  **if**(tweetStatus.getPlace()!=**null**) {  **if**(tweetStatus.getPlace().getCountry()!=**null**)te.setLocation(tweetStatus.getPlace().getCountry());  **if**(tweetStatus.getPlace().getGeometryCoordinates()!=**null**) {  GeoLocation[][] lo = tweetStatus.getPlace().getGeometryCoordinates();  te.setGeoLocation(lo[0][0].getLatitude()+AppConstants.***COMMA***+lo[0][0].getLongitude());  }  **if**(tweetStatus.getPlace().getBoundingBoxCoordinates()!=**null**) {  GeoLocation[][] lo = tweetStatus.getPlace().getBoundingBoxCoordinates();  te.setGeoLocation(lo[0][0].getLatitude()+AppConstants.***COMMA***+lo[0][0].getLongitude());  }  }  **for**(HashtagEntity ht:tweetStatus.getHashtagEntities()) {**for**(**int** i=0; i<keywords.length;i++){  **if**(keywords[i].equalsIgnoreCase(ht.getText())) {te.setHashTag(keywords[i]);**break**;}}}  te.setSentimentType(SentimentsAnalysisService.*analyseSentiment*(te.getTweet()));    rand = random.nextInt(10);    String str = o.writeValueAsString(te);  Event tweetEvent = EventBuilder.*withBody*( str.getBytes());  channelProcessor.processEvent(tweetEvent);  }  } **while** ((qry = rslt.nextQuery()) != **null**);  } **while** (**true**);  }**catch** (TwitterException extw) {  extw.printStackTrace();  System.***out***.println("Exception searching tweets - " + extw.getMessage());  }**catch** (Exception ex) {  ex.printStackTrace();  }  }    **public** **void** stop() {  **super**.stop();  }  } |
|  |
| Flume Kafla Sink: -  **public** **class** FlumeKafkaSink **extends** AbstractSink **implements** Configurable, BatchSizeSupported {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(FlumeKafkaSink.**class**);  **private** **final** Properties kafkaProps = **new** Properties();  **private** KafkaProducer<String, **byte**[]> producer;  **private** String topic;  **private** **int** batchSize;  **private** List<Future<RecordMetadata>> kafkaFutures;  **private** KafkaSinkCounter counter;  **private** **boolean** useAvroEventFormat;  **private** String partitionHeader = **null**;  **private** Integer staticPartitionId = **null**;  **private** **boolean** allowTopicOverride;  **private** String topicHeader = **null**;  **private** String timestampHeader = **null**;  **private** Map<String, String> headerMap;  **private** **boolean** useKafkaTransactions = **false**;  **private** Optional<SpecificDatumWriter<AvroFlumeEvent>> writer = Optional.*absent*();  **private** Optional<ByteArrayOutputStream> tempOutStream = Optional.*absent*();  **private** BinaryEncoder encoder = **null**;  **public** String getTopic() {  **return** topic;  }  **public** **long** getBatchSize() {  **return** batchSize;  }  @Override  **public** Status process() **throws** EventDeliveryException {  Status result = Status.***READY***;  Channel channel = getChannel();  Transaction transaction = **null**;  Event event = **null**;  String eventTopic = **null**;  String eventKey = **null**;  **try** {  **long** processedEvents = 0;  transaction = channel.getTransaction();  transaction.begin();  **if** (useKafkaTransactions) {  producer.beginTransaction();  }  kafkaFutures.clear();  **long** batchStartTime = System.*nanoTime*();  **for** (; processedEvents < batchSize; processedEvents += 1) {  event = channel.take();  **if** (event == **null**) {  **if** (processedEvents == 0) {  result = Status.***BACKOFF***;  counter.incrementBatchEmptyCount();  } **else** {  counter.incrementBatchUnderflowCount();  }  **break**;  }  counter.incrementEventDrainAttemptCount();  **byte**[] eventBody = event.getBody();  Map<String, String> headers = event.getHeaders();  **if** (allowTopicOverride) {  eventTopic = headers.get(topicHeader);  **if** (eventTopic == **null**) {  eventTopic = BucketPath.*escapeString*(topic, event.getHeaders());  ***logger***.debug("{} was set to true but header {} was null. Producing to {}" +  " topic instead.",  **new** Object[]{AppConstants.***ALLOW\_TOPIC\_OVERRIDE\_HEADER***,  topicHeader, eventTopic});  }  } **else** {  eventTopic = topic;  }  eventKey = headers.get(AppConstants.***KEY\_HEADER***);  **if** (***logger***.isTraceEnabled()) {  **if** (LogPrivacyUtil.*allowLogRawData*()) {  ***logger***.trace("{Event} " + eventTopic + " : " + eventKey + " : "  + **new** String(eventBody, StandardCharsets.***UTF\_8***));  } **else** {  ***logger***.trace("{Event} " + eventTopic + " : " + eventKey);  }  }  ***logger***.debug("event #{}", processedEvents);  // create a message and add to buffer  **long** startTime = System.*currentTimeMillis*();  Integer partitionId = **null**;  **try** {  ProducerRecord<String, **byte**[]> record;  **if** (staticPartitionId != **null**) {  partitionId = staticPartitionId;  }  //Allow a specified header to override a static ID  **if** (partitionHeader != **null**) {  String headerVal = event.getHeaders().get(partitionHeader);  **if** (headerVal != **null**) {  partitionId = Integer.*parseInt*(headerVal);  }  }  Long timestamp = **null**;  **if** (timestampHeader != **null**) {  String value = headers.get(timestampHeader);  **if** (value != **null**) {  **try** {  timestamp = Long.*parseLong*(value);  } **catch** (Exception ex) {  ***logger***.warn("Invalid timestamp in header {} - {}", timestampHeader, value);  }  }  }  List<Header> kafkaHeaders = **null**;  **if** (!headerMap.isEmpty()) {  List<Header> tempHeaders = **new** ArrayList<>();  **for** (Map.Entry<String, String> entry : headerMap.entrySet()) {  String value = headers.get(entry.getKey());  **if** (value != **null**) {  tempHeaders.add(**new** RecordHeader(entry.getValue(),  value.getBytes(StandardCharsets.***UTF\_8***)));  }  }  **if** (!tempHeaders.isEmpty()) {  kafkaHeaders = tempHeaders;  }  }  **if** (partitionId != **null**) {  record = **new** ProducerRecord<>(eventTopic, partitionId, timestamp, eventKey,  serializeEvent(event, useAvroEventFormat), kafkaHeaders);  } **else** {  record = **new** ProducerRecord<>(eventTopic, **null**, timestamp, eventKey,  serializeEvent(event, useAvroEventFormat), kafkaHeaders);  }  kafkaFutures.add(producer.send(record, **new** SinkCallback(startTime)));  } **catch** (NumberFormatException ex) {  **throw** **new** EventDeliveryException("Non integer partition id specified", ex);  } **catch** (Exception ex) {  **throw** **new** EventDeliveryException("Could not send event", ex);  }  }  **if** (useKafkaTransactions) {  producer.commitTransaction();  } **else** {  producer.flush();  **for** (Future<RecordMetadata> future : kafkaFutures) {  future.get();  }  }  // publish batch and commit.  **if** (processedEvents > 0) {  **long** endTime = System.*nanoTime*();  counter.addToKafkaEventSendTimer((endTime - batchStartTime) / (1000 \* 1000));  counter.addToEventDrainSuccessCount(processedEvents);  }  transaction.commit();  } **catch** (Exception ex) {  String errorMsg = "Failed to publish events";  ***logger***.error("Failed to publish events", ex);  counter.incrementEventWriteOrChannelFail(ex);  **if** (transaction != **null**) {  **try** {  kafkaFutures.clear();  **try** {  **if** (useKafkaTransactions) {  producer.abortTransaction();  }  } **catch** (ProducerFencedException e) {  ***logger***.error("Could not rollback transaction as producer fenced", e);  } **finally** {  transaction.rollback();  counter.incrementRollbackCount();  }  } **catch** (Exception e) {  ***logger***.error("Transaction rollback failed", e);  **throw** Throwables.*propagate*(e);  }  }  **throw** **new** EventDeliveryException(errorMsg, ex);  } **finally** {  **if** (transaction != **null**) {  transaction.close();  }  }  **return** result;  }  @Override  **public** **synchronized** **void** start() {  // instantiate the producer  producer = **new** KafkaProducer<>(kafkaProps);  **if** (useKafkaTransactions) {  ***logger***.info("Transactions enabled, initializing transactions");  producer.initTransactions();  }  counter.start();  **super**.start();  }  @Override  **public** **synchronized** **void** stop() {  producer.close();  counter.stop();  ***logger***.info("Kafka Sink {} stopped. Metrics: {}", getName(), counter);  **super**.stop();  }  @Override  **public** **void** configure(Context context) {  String topicStr = context.getString(AppConstants.***TOPIC\_CONFIG***);  **if**(topicStr == **null** || topicStr.trim().isEmpty())  topicStr = System.*getProperty*(AppConstants.***TOPIC\_CONFIG***, AppConstants.***TOPIC\_CONFIG\_VAL***);  **if** (topicStr == **null** || topicStr.isEmpty()) {  topicStr = AppConstants.***DEFAULT\_TOPIC***;  ***logger***.warn("Topic was not specified. Using {} as the topic.", topicStr);  } **else** {  ***logger***.info("Using the static topic {}. This may be overridden by event headers", topicStr);  }  topic = topicStr;  timestampHeader = context.getString(AppConstants.***TIMESTAMP\_HEADER***);  headerMap = context.getSubProperties(AppConstants.***KAFKA\_HEADER***);  batchSize = context.getInteger(AppConstants.***BATCH\_SIZE***, AppConstants.***DEFAULT\_BATCH\_SIZE***);  **if** (***logger***.isDebugEnabled()) {  ***logger***.debug("Using batch size: {}", batchSize);  }  useAvroEventFormat = context.getBoolean(AppConstants.***AVRO\_EVENT***,  AppConstants.***DEFAULT\_AVRO\_EVENT***);  partitionHeader = context.getString(AppConstants.***PARTITION\_HEADER\_NAME***);  staticPartitionId = context.getInteger(AppConstants.***STATIC\_PARTITION\_CONF***);  allowTopicOverride = context.getBoolean(AppConstants.***ALLOW\_TOPIC\_OVERRIDE\_HEADER***,  AppConstants.***DEFAULT\_ALLOW\_TOPIC\_OVERRIDE\_HEADER***);  topicHeader = context.getString(AppConstants.***TOPIC\_OVERRIDE\_HEADER***,  AppConstants.***DEFAULT\_TOPIC\_OVERRIDE\_HEADER***);  String transactionalID = context.getString(AppConstants.***TRANSACTIONAL\_ID***);  **if** (transactionalID != **null**) {  **try** {  context.put(AppConstants.***TRANSACTIONAL\_ID***, InetAddress.*getLocalHost*().getCanonicalHostName() +  Thread.*currentThread*().getName() + transactionalID);  useKafkaTransactions = **true**;  } **catch** (UnknownHostException e) {  **throw** **new** ConfigurationException("Unable to configure transactional id, as cannot work out hostname", e);  }  }  **if** (***logger***.isDebugEnabled()) {  ***logger***.debug(AppConstants.***AVRO\_EVENT*** + " set to: {}", useAvroEventFormat);  }  kafkaFutures = **new** LinkedList<Future<RecordMetadata>>();  String bootStrapServers = context.getString(AppConstants.***BOOTSTRAP\_SERVERS\_CONFIG***);  **if**(bootStrapServers == **null** || bootStrapServers.trim().isEmpty())  bootStrapServers = System.*getProperty*(AppConstants.***BOOTSTRAP\_SERVERS\_CONFIG***, AppConstants.***BOOTSTRAP\_SERVERS\_CONFIG\_VAL***);  **if** (bootStrapServers == **null** || bootStrapServers.isEmpty()) {  **throw** **new** ConfigurationException("Bootstrap Servers must be specified");  }  setProducerProps(context, bootStrapServers);  **if** (***logger***.isDebugEnabled() && LogPrivacyUtil.*allowLogPrintConfig*()) {  ***logger***.debug("Kafka producer properties: {}", kafkaProps);  }  **if** (counter == **null**) {  counter = **new** KafkaSinkCounter(getName());  }  }    **private** **void** setProducerProps(Context context, String bootStrapServers) {  kafkaProps.clear();  kafkaProps.put(ProducerConfig.***ACKS\_CONFIG***, AppConstants.***DEFAULT\_ACKS***);  kafkaProps.put(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***, AppConstants.***DEFAULT\_KEY\_SERIALIZER***);  kafkaProps.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, AppConstants.***DEFAULT\_VALUE\_SERIAIZER***);  kafkaProps.putAll(context.getSubProperties(AppConstants.***KAFKA\_PRODUCER\_PREFIX***));  kafkaProps.put(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, bootStrapServers);  **if** (*isSSLEnabled*(kafkaProps) && "true".equalsIgnoreCase(kafkaProps.getProperty(***SSL\_DISABLE\_FQDN\_CHECK***))) {  kafkaProps.put(SslConfigs.***SSL\_ENDPOINT\_IDENTIFICATION\_ALGORITHM\_CONFIG***, "");  }  KafkaSSLUtil.*addGlobalSSLParameters*(kafkaProps);    String jass\_config = context.getString(AppConstants.***JAAS\_CONFIG***);  **if**(jass\_config == **null** || jass\_config.trim().isEmpty())  jass\_config = System.*getProperty*(AppConstants.***JAAS\_CONFIG***, AppConstants.***JAAS\_CONFIG\_VAL***);  **if**(jass\_config!=**null** && !jass\_config.isEmpty()) {  kafkaProps.put(AppConstants.***JAAS\_CONFIG***, jass\_config);  kafkaProps.put(AppConstants.***SASL\_MECHANISM***, System.*getProperty*(AppConstants.***SASL\_MECHANISM***, AppConstants.***SASL\_MECHANISM\_PLAIN***));  kafkaProps.put(CommonClientConfigs.***SECURITY\_PROTOCOL\_CONFIG***, System.*getProperty*(CommonClientConfigs.***SECURITY\_PROTOCOL\_CONFIG***,  AppConstants.***SECURITY\_PROTOCOL\_SASL\_SSL***));  }  // Required for correctness in Apache Kafka clients prior to 2.6  kafkaProps.put(AppConstants.***CLIENT\_DNS\_LOOKUP***, System.*getProperty*(AppConstants.***CLIENT\_DNS\_LOOKUP***, AppConstants.***USE\_ALL\_DNS\_IPS***));  // Best practice for higher availability in Apache Kafka clients prior to 3.0  kafkaProps.put(AppConstants.***SESSION\_TIMEOUT\_MS***, Integer.*parseInt*(System.*getProperty*(AppConstants.***SESSION\_TIMEOUT\_MS***, AppConstants.***SESSION\_TIMEOUT\_MS\_DEFAULT***)));  }  **protected** Properties getKafkaProps() {  **return** kafkaProps;  }  **private** **byte**[] serializeEvent(Event event, **boolean** useAvroEventFormat) **throws** IOException {  **byte**[] bytes;  **if** (useAvroEventFormat) {  **if** (!tempOutStream.isPresent()) {  tempOutStream = Optional.*of*(**new** ByteArrayOutputStream());  }  **if** (!writer.isPresent()) {  writer = Optional.*of*(**new** SpecificDatumWriter<AvroFlumeEvent>(AvroFlumeEvent.**class**));  }  tempOutStream.get().reset();  AvroFlumeEvent e = **new** AvroFlumeEvent(*toCharSeqMap*(event.getHeaders()),  ByteBuffer.*wrap*(event.getBody()));  encoder = EncoderFactory.*get*().directBinaryEncoder(tempOutStream.get(), encoder);  writer.get().write(e, encoder);  encoder.flush();  bytes = tempOutStream.get().toByteArray();  } **else** {  bytes = event.getBody();  }  **return** bytes;  }  **private** **static** Map<CharSequence, CharSequence> toCharSeqMap(Map<String, String> stringMap) {  Map<CharSequence, CharSequence> charSeqMap = **new** HashMap<CharSequence, CharSequence>();  **for** (Map.Entry<String, String> entry : stringMap.entrySet()) {  charSeqMap.put(entry.getKey(), entry.getValue());  }  **return** charSeqMap;  }  }  **class** SinkCallback **implements** Callback {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(SinkCallback.**class**);  **private** **long** startTime;  **public** SinkCallback(**long** startTime) {  **this**.startTime = startTime;  }  **public** **void** onCompletion(RecordMetadata metadata, Exception exception) {  **if** (exception != **null**) {  ***logger***.warn("Error sending message to Kafka {} ", exception.getMessage());  }  **if** (***logger***.isDebugEnabled()) {  **long** eventElapsedTime = System.*currentTimeMillis*() - startTime;  **if** (metadata != **null**) {  ***logger***.debug("Acked message partition:{} ofset:{}", metadata.partition(),  metadata.offset());  }  ***logger***.debug("Elapsed time for send: {}", eventElapsedTime);  }  }  } |
| Flume.conf:  # The configuration file needs to define the sources,  # the channels and the sinks.  # Sources, channels and sinks are defined per agent,  # in this case called 'FlumeTwitAgent'  # Other config values specific to each type of channel(sink or source)  # can be defined as well  # In this case, it specifies the capacity of the memory channel  FlumeTwitAgent.sources = Twitter  FlumeTwitAgent.channels = FlumeTwitAgentChannel  FlumeTwitAgent.sinks = kafka  FlumeTwitAgent.sources.Twitter.type = cscie88.assignments.FlumeTwitterSource  FlumeTwitAgent.sources.Twitter.channels.selector.type = replicating  FlumeTwitAgent.sources.Twitter.channels = FlumeTwitAgentChannel  FlumeTwitAgent.sources.Twitter.consumerKey = <API Key>  FlumeTwitAgent.sources.Twitter.consumerSecret = <API Key Secret>  FlumeTwitAgent.sources.Twitter.accessToken = <Access Token>  FlumeTwitAgent.sources.Twitter.accessTokenSecret = <Access Token Secret>  FlumeTwitAgent.sources.Twitter.keywords = netflix,disneyPlus,hulu,primeVideo,vudu,hbo  FlumeTwitAgent.channels.FlumeTwitAgentChannel.type = memory  FlumeTwitAgent.channels.FlumeTwitAgentChannel.capacity = 10000  FlumeTwitAgent.channels.FlumeTwitAgentChannel.transactionCapacity = 1000  FlumeTwitAgent.sinks.kafka.type = cscie88.assignments.FlumeKafkaSink  FlumeTwitAgent.sinks.kafka.topic = quickstart-evnts  FlumeTwitAgent.sinks.kafka.brokerList = <bootstrap.servers>  FlumeTwitAgent.sinks.kafka.security.protocol = SASL\_SSL  FlumeTwitAgent.sinks.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username='<Kafka API Key>' password='<Kafka API Secret>';  FlumeTwitAgent.sinks.sasl.mechanism=PLAIN  # Required for correctness in Apache Kafka clients prior to 2.6  FlumeTwitAgent.sinks.client.dns.lookup=use\_all\_dns\_ips  # Best practice for higher availability in Apache Kafka clients prior to 3.0  FlumeTwitAgent.sinks.session.timeout.ms=45000  ## Best practice for Kafka producer to prevent data loss  FlumeTwitAgent.sinks.acks=all  FlumeTwitAgent.sinks.kafka.api.key = <Kafka API Key>  FlumeTwitAgent.sinks.kafka.api.secret = <Kafka API Secret>  FlumeTwitAgent.sinks.kafka.requiredAcks = 1  FlumeTwitAgent.sinks.kafka.batchSize = 20  FlumeTwitAgent.sinks.kafka.channel = FlumeTwitAgentChannel |
| Kafka Elasticsearch Connector : - |

## Results

Demonstrate your results. Provide all relevant screenshots of the log files, data stored in DB and GUI tier if applicable.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

## Conclusions and Lesson Learned

Describe what you have learned during this project:

* What issues did you have?

|  |
| --- |
| I wanted to generate a graph of tweets’ geolocation and perform an aggregation, but most tweets did not show location information. This is because the tweet location is off by default, and users must opt-in to the service to attach it. Our geolocation dataset got better after collecting 28000 tweets: |

* What limitations, if any, did you run into with the technologies used?

|  |
| --- |
| Twitter API has a rate limit of up to 300 requests per app and up to 900 requests per user over any 15-minute interval.  When these limits are exceeded, an error is returned. |

* What would you do differently next time?

|  |
| --- |
| I would like to filter streams on movies and shows and analyze tweets data to recommend movies. |

* What alternatives to technologies you used you might consider ?

|  |
| --- |
| Many real-time pipelines built over the past few years use Apache Kafka for large-scale distributed messaging.  Apache Kafka follows a publish-subscribe model, so multiple subscribers can be created for the same topic. Subscribers are assigned a partition for better scalability.  Partitions are directly stored in the Leader Node, and data is replicated to the Replica Node for fault tolerance. The **biggest challenge in this approach** is that the partition is stored on a local disk with limited space. Another challenge is that incoming messages will halt once the Replica node size is filled, leading to data loss. Kafka Brokers aren’t stateless, which means another Broker must synchronize the state from the current broker if it fails.  The Apache Pulsar Messaging System was contributed to the Apache Software Foundation in 2016. It aims to fill in the gaps of existing open-source messaging solutions like**multi-tenancy, geo-replication, and durability.**  **Apache Pulsar’s cloud-native design** separates the storage of messages from their delivery, resulting in a system that can s**cale up**or dow**n** dynamically without causing downtime.  In Apache Pulsar, partitions are subdivided into segments evenly distributed across Bookies, eliminating the need to replicate content when the memory is maxed out. Also, brokers are stateless. Pulsar maintains the state, but the data is stored in Apache Bookkeeper rather than brokers. Therefore, no physical limit exists to how much data can be retained and ingested.  **Pulsar supports Geo-replication,** allowing messages published to a Topic to be automatically replicated to the configured remote Geo-location without any complicated setup.  Apache Pulsar was created as a ground-up multi-tenant event streaming system. Tenants can be spread across clusters, and each can have its own authentication and authorization scheme applied to them. They are also the administrative unit where storage quotas, message TTL, and isolation policies can be managed.  The Pulsar software comes with a **preinstalled schema registry**. Producers and consumers inform the system about data types transmitted via the topic, ensuring type safety and synchronization.  Pulsar features serverless functions to natively process data, eliminating the need to run an external stream processing system to perform data transformations, filtering, and aggregation.  Apache Pulsar is a distributed messaging and streaming solution that features advantages over Kafka in native geo-replication, scalability, all-in-one messaging, and zero data loss. |

* Where would you take your project next ?

|  |
| --- |
| Creating personalized movie recommendations based on User tweets is a very interesting research subject. I would like to analyze users' posted movie tweets to understand their sentiments. We will process movie tweets using tokenization, porter stemming, stop word removal, and a Word-Net dictionary to match synonyms. Then, we apply the Latent Semantic Indexing technique, which involves Singular Value Decomposition, to this processed data and predict the genre based on IMDb movie genre categorization. |