Анализ сообщений социальной сети Twitter с использованием систем обработки потоковых данных Apache Spark и Apache Storm

Горшков Н.А, Денисов В.С.

***Аннотация* — В статье обсуждается сравнение систем обработки потоковых данных Apache Storm и Apache Spark в задаче анализа сообщений социальной сети Twitter. Сначала описываются основные концепции движков, особенности их настройки и запуска приложений. Затем рассматриваются конкретная задача анализа твитов, а также структура кластера, на котором проводился тест производительности. В заключении были сделаны выводы о применимости Storm и Spark для рассмотренной задачи.**

***Ключевые слова*— Apache Storm, Apache Spark, Apache Kafka, большие данные, потоковые данные, Twitter, AWS**

I. Введение

В последние несколько лет термин “большие данные” все больше набирает популярность. Множество источников, такие как Интернет, социальные сети, сенсоры устройств из сети интернета вещей, а также некоторые научные исследования в области физики элементарных частиц [1], биоинформатики [2] и астрономии [3], создают огромные объемы как структурированной, так и неструктурированной информации.

Как указано в [4], самой известной моделью для управления и обработки больших объемов данных является модель MapReduce. Основными ее особенностями являются простая модель программирования, линейная масштабируемость и встроенная отказоустойчивость. В работе [5] проведено подробное сравнение различных реализаций этой модели, где заключается, что самой популярной и наиболее адаптированной под цели разработчиков является Apache Hadoop [6]. Apache Hadoop - это фреймворк, с помощью которого можно делать распределенные вычисления больших объемов данных на кластере компьютеров с использованием простой программной модели. Он спроектирован для пакетной обработки огромного объема данных. Это значит, что для получения результата требуются часы, или даже дни, но иногда ответ необходим в течение минуты или нескольких секунд.

Последнее время появляется все больше систем, выполняющих обработку больших потоков данных в реальном времени. В [7] сказано, что самыми распространенными являются Apache Spark [8] и Apache Storm [9]. Поэтому в этой статье пойдет речь о сравнении этих движков потоковой обработки. Это сравнение будет включать в себя обзор основных концепций обоих движков, рассмотрение сложности развертывания и настройки приложений, модели программирования под каждый из них и тест производительности для конкретной задачи.

Статья получена 13 сентября 2016.

Горшков Н.А., студент 2го курса магистратуры факультета ВМК МГУ им. М.В.Ломоносова (email: gor-nikita93@yandex.ru)

Денисов В.С., МГУ имени М.В. Ломоносова, (email: vdenisov@plukh.org)

II. Постановка задачи

Главной задачей этой работы является выбор одного из представленных движков обработки больших потоков данных для разработки эффективной системы кластеризации входящих сообщений по темам. Так как сообщения должны быть написаны реальными людьми, то в качестве источника данных была выбрана социальная сеть Twitter, которая предоставляет удобный общедоступный API [11] для получения потока опубликованных сообщений с возможностью фильтрации по языку, местоположению и другим параметрам. Данные из Twitter будут поступать в промежуточный кластер, работающий под управлением Apache Kafka [10]. Apache Kafka – это распределенный программный брокер сообщений, который распределенно сохраняет поступающие сообщения на заданный промежуток времени для последующей их обработки.

Для решения этой задачи были сформулированы следующие критерии:

- Простота развертывания приложения
- Простота настройки кластера
- Потоковая модель
- Модель программирования
- Гарантированность доставки сообщений
- Масштабируемость
- Отказоустойчивость
- Удобство пользовательского интерфейса
- Модель обработки данных
- Максимальная скорость обработки потока
- Величина задержки результата

Тест производительности в настоящей работе было решено провести с помощью облачной платформы Amazon Elastic Compute Cloud (Amazon EC2) [15] на 2 инстансах-обработчиках и задачи из области обработки естественного языка, так как будущая система кластеризации будет развернута именно на серверах AWS.

III. Обзор предметной области

По теме сравнения движков Apache Storm и Apache Spark в данный момент существует достаточно мало работ из-за относительной новизны области. Одной из таких работ является [18]. Но в этой работе, во-первых, для сравнения используются задачи не из области обработки естественного языка, а, во-вторых, не предоставляются исходные тексты программ, поэтому проверить корректность достигнутых результатов невозможно. В работе [19] инженеры из Yahoo! сравнили производительность таких систем потоковой обработки данных, как Apache Spark, Apache Storm и Apache Flink [20]. Исходные коды тестов производительности выложены на Github [21], в отличие от предыдущей работы, но запускались эти программы с помощью физических серверов Yahoo! с 24GiB оперативной памяти и 16-ядерными процессорами (8 физических). Из-за отсутствия доступа к таким ресурсам в точности повторить их тест невозможно. Так же, как и в [18] задача, которая выполнялась при сравнении производительности, не связана с обработкой естественного языка. Более того, статья была написана в 2014 году, поэтому тесты проводились на более старых версиях движков (0.10.0 для Storm и 1.5.1 для Spark). Настоящая работа использует версии 1.0.2 и 1.6.1 соответственно.

IV. Основные концепции

A. *Apache Spark Streaming*

Spark Streaming – это расширение основного ядра Spark API, которое позволяет осуществлять масштабируемую, отказоустойчивую обработку потока данных с высокой пропускной способностью [8]. Данные в Spark могут поступать из таких источников как Apache Kafka [10], Twitter Streaming API [11], Apache Flume [12], Amazon Kinesis Streams [13] и другие.

Модуль Spark Streaming получает данные, делит их на пакеты и отправляет в основной движок Spark, который их обрабатывает и генерирует конечный поток результирующих пакетов. Такая потоковая модель называется микро-пакетированием (micro-batching). Spark Streaming манипулирует абстракцией, называемой DStream (discretized stream, дискретизированный поток), который представляет из себя непрерывную последовательность RDD (resilient distributed dataset, распределенный набор данных). RDD – отказоустойчивая и неизменяемая коллекция элементов, которую можно обрабатывать параллельно. Каждый RDD в DStream содержит данные из определенного интервала времени. Каждый такой распределенный набор делится на разделы (partitions) и распределяется по кластеру, где каждый раздел ждет обработки. Такая модель обработки называется параллелизмом данных.

DStream поддерживает различные преобразования, которые можно выполнять над ним. Это такие функции как `map(func)` (возвращение нового DStream с помощью пропускания каждого элемента через функцию func), `filter(func)` (возвращение нового DStream с помощью отбора элементов через функцию-предикат func), `repartition(numPartitions)` (изменение степени параллелизма потока), `union(otherStream)` (объединение двух потков) и другие. Также DStream поддерживает выходные операции, самой важной из которых является `foreachRDD(func)`, которая применяет func к каждому RDD, сгенерированному DStream.

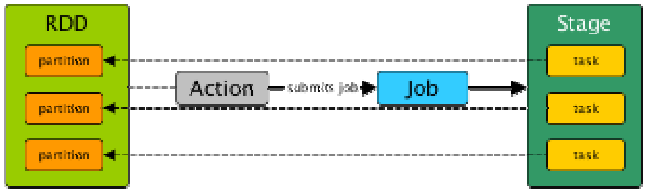

Рисунок 1 – Создание задания для Spark Streaming

Spark имеет декларативную модель программирования, то есть программист в функциональном стиле описывает, что программа должна выполнять для обработки каждой RDD, а Spark сам создаст код, непосредственно выполняющийся на машинах-обработчиках. Но начинает выполняться этот код только в момент вызова выходной операции (такой, как `foreachRDD`). После выполнения таких операций создается задание (Job), которое может содержать несколько этапов (Stage). Существует 2 вида этапов: этап перемешивания (shuffle map stage) и результирующий этап. Этап перемешивания создается функциями, требующими перемещения данных по кластеру. Такими функциями являются `repartition`, `groupByKey`, `reduceByKey`, `join`. Результирующий этап – это этап, завершающий вычисления, создающийся выходными операциями. Каждая стадия создает несколько задач (Task), выполняющих одну и ту же работу, но над разными данными. Каждая задача ассоциируется с одним или несколькими разделами RDD и отправляется на машину к этим разделам, где их и обрабатывает. На рисунке 2 представлено создание задания с одним этапом и дальнейшим распределением задач по разделам RDD.

Для гарантирования доставки сообщений Spark Streaming использует семантику доставки ровно один раз (exactly-once delivery). Это позволяет сделать модель микро-пакетирования, так как необходимо отслеживать только состояние пакетов, которые могут быть либо обработаны, либо нет. Поэтому ни один пакет не будет обработан дважды, и никакие данные не потеряются.
Spark является отказоустойчивым и может восстанавливать состояние после выхода из строя процессов-воркеров (slave-процессов) и машин-обработчиков. Master-процесс ответственен за публикацию приложения и перераспределение задач между обработчиками, поэтому пока этого не требуется, выход из строя этого процесса не повлияет на работу кластера.

B. *Apache Storm*

В отличие от Spark Streaming Storm использует нативную потоковую модель (native streaming), то есть поток не делится на пакеты, а каждое событие обрабатывается отдельно. Storm использует композиционную модель программирования, с помощью которой программист реализует определенные интерфейсы для создания “строительных блоков” программы, а затем соединяет их между собой, создавая топологию (topology). Топология – это граф, в

узлах которого находятся источники (Spouts) и обработчики (Bolts), которые будут рассмотрены ниже. Основной абстракцией, которой оперирует Storm, является поток (Stream). Поток – это бесконечная последовательность кортежей (Tuples), которые распределенно обрабатываются. Потоки объявляются с помощью схемы, которая определяет имена полей кортежей.

Spout – это источник потоков в топологии. Обычно данные читаются из внешних источников таких, как Apache Kafka [10], Twitter Streaming API [11], Apache Flume [12], Amazon Kinesis Streams [13] и других. Источники могут быть надежными и ненадежными. Надежные заново отправляют на обработку кортежи, которые не были корректно отработаны, ненадежные забывают о кортеже сразу после передачи в следующее звено топологии. Spout может испускать несколько потоков.

Bolt – это обработчик кортежей. С помощью них можно делать любую обработку потоков. Для сложных многоэтапных вычислений можно создавать несколько обработчиков. Обработчики также, как и источники могут испускать несколько потоков.

После создания всех источников и обработчиков необходимо создать топологию с помощью механизма группировки. После создания топологии ее необходимо опубликовать на сервере с помощью класса StormSubmitter.

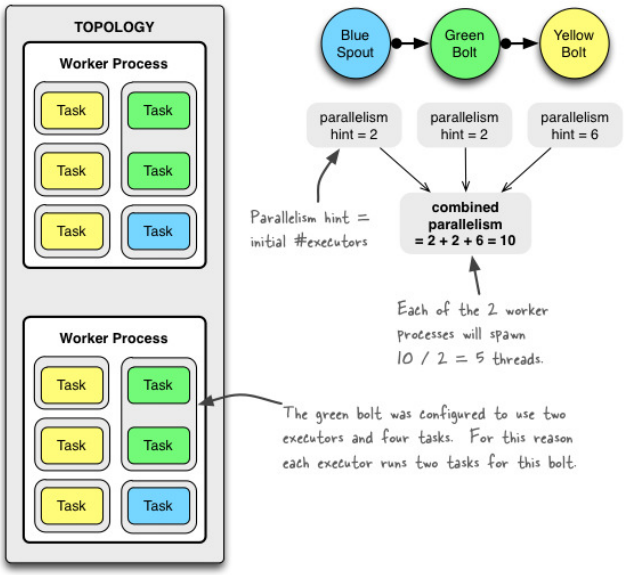

Рисунок 2 – Запущенная Storm топология

Каждая топология выполняется на нескольких рабочих процессах, распределенных по кластеру. Каждый рабочий процесс создает один или несколько исполнителей (Executors). Executor – поток выполнения, связанный с определенным компонентом топологии (Bolt или Spout). Для каждого компонента может быть создано несколько исполнителей для возможности выполнять обработку параллельно. Реальную обработку выполняют задания (Tasks), которые создаются внутри исполнителя. На рисунке 2 приведен пример топологии, состоящей из 1го источника и 2х обработчиков, запущенной на 2 рабочих процессах. Storm использует модель параллелизма задач для обработки событий.

В отличие от Spark Storm для гарантирования доставки сообщений использует семантику отправки как минимум один раз (at least one delivery). Так как в данном случае обрабатываются отдельные сообщения, а не пакеты, нужно отслеживать состояние каждого события. В Storm для этого используется система подтверждений (acknowledgements). Каждая задача, выделенная на Spout, к испущенному сообщению добавляет уникальный идентификатор и хранит их копии. Обработчики после обработки каждого сообщения посылают подтверждение того, что данное событие успешно прошло всю топологию. После этого соответствующие задачи источников вызывают метод `ack` и удаляют копию сообщения, уведомляя при это источник данных (в данной работе этим источником является Kafka). При неудаче в обработке Bolt уведомляет соответствующий Spout, который вызывает метод fail и кладет копию сообщения с этим id обратно в очередь на обработку. Если в течение времени, которое определяется настройкой `Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS`, сообщение не было подтверждено, то в этом случае источник отправляет его на повторную обработку. Из-за сложности отслеживания большого количества сообщений некоторые подтверждения могут не дойти до источника, поэтому может произойти повторная обработка сообщения. Такая ситуация делает невозможным семантику доставки ровно один раз, которая используется в Apache Spark.

Storm, так же, как и Spark, является отказоустойчивым. С помощью механизма, описанного выше, при выходе из строя любой машины-обработчика или процесса-обработчика, все неподтвержденные сообщения будут обработаны после перезапуска вышедшей из строя машины/процесса. При выходе из строя машины все задачи, которые были ей выданы, отдаются другой машине. При завершении процесса-обработчика он перезапускается специальным демоном Storm, который называется Supervisor. Все состояние кластера Storm хранится в кластере Zookeeper [14] (система управления, синхронизации и конфигурации серверов кластера), который реплицирует все полученные данные между своими серверами, что практически исключает возможность потери этих данных. Процесс Nimbus отвечает за публикацию топологии, перераспределение задач между обработчиками. При остановке Nimbus-процесса кластер продолжит функционировать до тех пор, пока не потребуется перераспределять задачи.

V. Особенности первоначальной настройки и запуска приложений

A. *Apache Spark Streaming*

Программное обеспечение Spark должно быть распространено на узлы кластера, после чего необходимо запустить master-процесс, который заодно запустит веб-интерфейс для просмотра статистики приложений. Далее запускаются slave-процессы, которые будут присоединены к основному процессу с помощью URL типа `spark://host:port`. Для управления кластером используется встроенный менеджер; также доступна интеграция с внешними менеджерами, такими как YARN [16] или Mesos [17].

Приложение Spark отправляется на сервер с помощью специального скрипта, которому нужно указать main класс и путь к Jar-файлу приложения.

Теперь рассмотрим простейшее приложение для Spark Streaming. Код инициализации приложения Spark представлен на рисунке 3.

```
SparkConf conf = new SparkConf()
        .setAppName("twitter-test")
        .set("spark.default.parallelism", "2");
JavaStreamingContext ssc =
        new JavaStreamingContext(conf, Durations.seconds(1));
```

Рисунок 3 - Код инициализации приложения Spark

На рисунке 4 показан код создания входящего потока данных из Kafka.

```
Set<String> topics = new HashSet<>();
topics.add(Config.KAFKA_TOPIC);
Map<String, String> kafkaParams = new HashMap<>();
kafkaParams.put("group.id", "spark-consumer");
kafkaParams.put("auto.offset.reset", "smallest");
kafkaParams.put("metadata.broker.list", Config.KAFKA_BROKER_LIST);

JavaPairInputDStream<String, String> messages =
        KafkaUtils.createDirectStream(ssc,
                String.class, String.class,
                StringDecoder.class, StringDecoder.class,
                kafkaParams, topics);
```

Рисунок 4 - Код создания входящего потока данных из Kafka

После создания контекста и источника можно приступать к обработке. Код простейшей обработки представлен на рисунке 5. Это приложение получает сообщения из Kafka в виде пары (id, content), с помощью функции `map` трансформирует каждое сообщение к виду обычной строки, в которой записано содержание. После этого каждое сообщение выводится в стандартный поток вывода.

```
JavaDStream<String> statuses = messages.map((status) -> {
    return status._2();
});

statuses.foreachRDD((rdd) -> {
    rdd.foreach(System.out::println);
});
```

Рисунок 5 – Простейшее Spark-приложение

Обработка может проходить в несколько этапов. После каждого этапа при необходимости можно увеличивать или уменьшать количество потоков, которыми обрабатывается RDD. Также можно соединять и разводить потоки. Таким образом с помощью Spark возможно создать что-то похожее на топологии Storm.

B. Apache Storm

Настройка Storm очень похожа на ту, что была описана для Spark. Также необходимо скачать скомпилированную версию Storm и положить ее на все узлы кластера. Но у Storm есть серьезное отличие, ему для работы необходим Zookeeper, который отвечает за координацию всех серверов кластера. Из-за этого необходимо скачать Zookeeper и разместить его на одном или нескольких серверах. Первым делом необходимо запустить Zookeeper. После этого запускается мастер-процесс Storm, который называется Nimbus, указав ему расположение серверов Zookeeper. На этой же машине можно запустить веб-интерфейс, который будет отображать информацию о рабочих машинах и приложениях. Теперь необходимо запустить процессы, называющиеся supervisors, которые ответственны за запуск и перезапуск процессов-обработчиков (executors).

Для публикации приложения в кластер нужно выполнить следующий специальный скрипт, передав в него в качестве параметров jar-файл приложения, имя класса-точки входа и параметры самого приложения.

Теперь рассмотрим, из чего состоит приложение. Сначала необходимо настроить интеграцию с Kafka. Для этого создается специальный KafkaSpout, код которого представлен на рисунке 6, который будет отправлять данные из Kafka в топологию.

```
TopologyBuilder topologyBuilder = new TopologyBuilder();
String topicName = Config.KAFKA_TOPIC;
BrokerHosts hosts = new ZkHosts(
        Config.ZOOKEEPER_IP + ":" + Config.ZOOKEEPER_PORT
);
SpoutConfig spoutConfig = new SpoutConfig(
        hosts, topicName, "/" + topicName, "kafkastorm"
);
spoutConfig.ignoreZkOffsets = true;
spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
topologyBuilder.setSpout("spout", kafkaSpout, 1);
```

Рисунок 6 – Код создания источника KafkaSpout

После того, как источник данных создан, нужно объявить обработчик (Bolt), исходный код которого представлен на рисунке 7, который извлекает контент сообщения и отправляет его дальше по топологии.

```
public class StormBolt extends BaseBasicBolt {
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String content = (String) tuple.getValueByField("content");
        collector.emit(new ArrayList<Object>() {{ add(content); }});
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {
        ofd.declare(new Fields("status", "msgid"));
    }
}
```

Рисунок 7 – Исходный код простого обработчика

А теперь этот Bolt нужно прицепить к уже созданному Spout, указав последним параметром количество исполнителей (степень параллелизма) для этого обработчика. На рисунке 8 показано, как это можно сделать.

```
topologyBuilder.setBolt("bolt", new StormBolt(), 2)
        .shuffleGrouping("spout");
```

Рисунок 8 – Присоединение обработчика к источнику

Теперь эту топологию, состоящую из одного Spout и одного Bolt можно опубликовать, указав сначала количество рабочих процессов, которые будут исполняться в кластере. Код публикации топологии показан на рисунке 9.

```
Config conf = new Config();
conf.setDebug(false);
StormTopology topology = topologyBuilder.createTopology();
conf.setNumWorkers(numWorkers);
conf.setMaxSpoutPending(4000);
StormSubmitter.submitTopology("mytopology", conf, topology);
```

Рисунок 9 – Публикация топологии

VI. Сравнительный анализ производительности

A. *Выбор задачи для сравнения производительности*

Для того, чтобы сравнить два движка обработки потоковых данных, необходимо выбрать достаточно ресурсоемкую задачу. Поскольку в дальнейшем предполагается решение задач, связанных с обработкой естественного языка, то в качестве модельной задачи было решено выбрать одну из типовых задач этой области. При этом ее решение должно достаточно сильно занимать процессор для того, чтобы данные сравнения были наиболее приближены к данным, которые получились бы на реальной системе обработки большого объема данных.

В качестве такой задачи для данной работы была выбрана проблема поиска всех N-грамм в твите. N-грамма – это последовательность N слов, расположенных друг за другом. N-граммная модель используется в основном для моделирования естественного языка [22]. При этом предполагается, что появление каждого слова зависит только от предыдущих слов. N-граммная модель рассчитывает вероятность последнего слова N-граммы, если известны все предыдущие. Другим применением N-грамм является выявление плагиата [23]. Если разделить текст на несколько небольших фрагментов, представленных N-граммами, их легко сравнить друг с другом, и таким образом получить степень сходства контролируемых документов. N-граммы часто успешно используются для категоризации текста и языка [24]. Используя N-граммы, можно эффективно найти кандидатов, чтобы заменить слова с ошибками правописания [25].

Для каждого сообщения искались все N-граммы, где `N = 1,..., msg.wordsCount.` На вход подается любая строка, состоящая из слов, разделенных пробелами. В случае данной работы этой строкой будет твит на английском языке. Исходный код программы, выполняющей эту задачу, заключен в классе `NGrams` репозитория [29]. Функция `allNGrams` этого класса будет использована при обработке сообщений для каждого из движков.

B. *Структура кластера*

Определившись с задачей, необходимо развернуть кластер, на котором будут тестироваться Storm и Spark. Для данной работы был развернут кластер, состоящий из пяти виртуальных машин Amazon EC2 типа t2.micro, предоставляемых на бесплатном аккаунте AWS, работающих под управлением операционной системы Red Hat Enterprise Linux 7.2. Такие инстансы предоставляют 1 ядро процессора и 1 Гб оперативной памяти. Все виртуальные машины были добавлены в одну зону доступности сети для того, чтобы они могли обмениваться между собой данными.

На рисунках 10 и 11 приведены структуры кластеров для Apache Storm и Apache Spark. В этой структуре представлены все процессы, работающие в системе независимо от решаемой задачи. Как было сказано выше, для того, чтобы приложение заработало в этой системе, необходимо выполнить команду submit для обоих движков. Можно заметить, что структуры очень похожи. Единственным отличием является то, что процесс Spark Master не взаимодействует с Zookeeper, так как сам отвечает за координацию всех своих серверов.

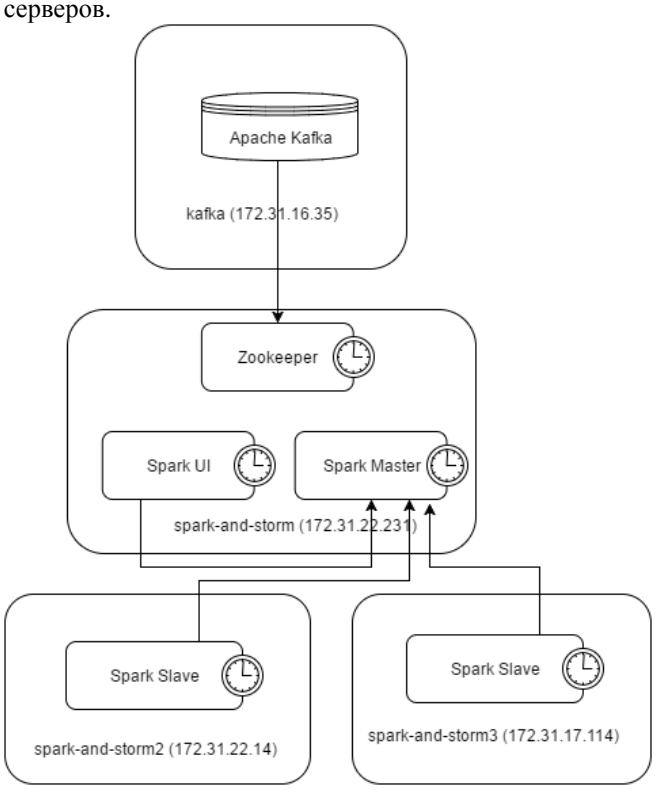

Рисунок 10 – Структура кластера Apache Spark Streaming

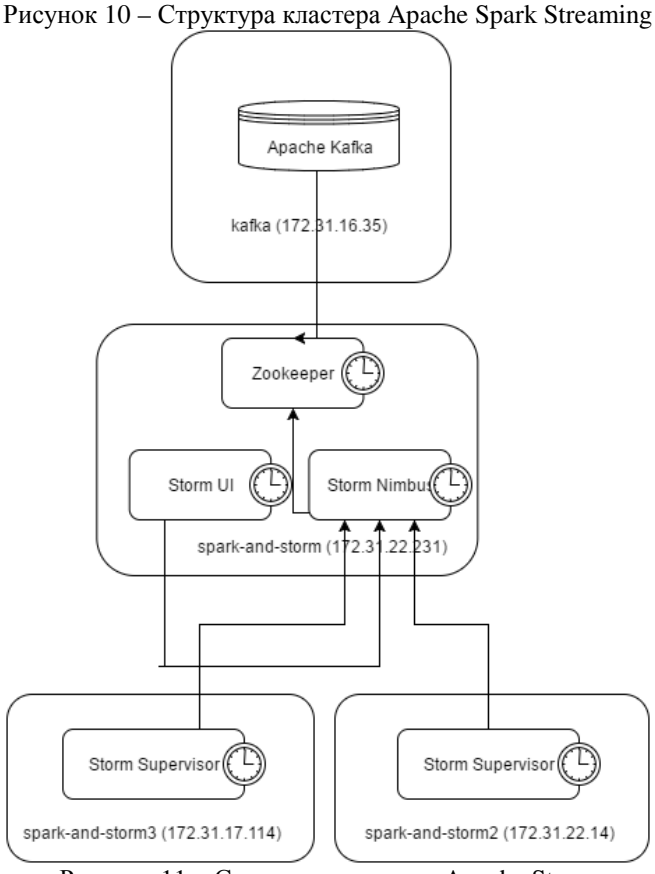

Рисунок 11 – Структура кластера Apache Strom

На центральной машине всегда работает мастер-процесс движков, его интерфейс и процесс zookeeper. Zookeeper необходим для координации брокеров Apache Kafka. Также Apache Storm использует Zookeeper для координации работы всех своих серверов. В данном случае хватает одного брокера, который работает на

отдельной машине. Брокеры хранят сообщения, поступающие с еще одного инстанса, на котором лежит файл с собранными твитами и работает программа-продюсер. На оставшихся двух машинах работают процессы-обработчики движков, которые непосредственно выполняют работу по процессингу поступающих данных.

C. Получение данных из Twitter

Для того, чтобы результат работы был повторяем необходимо получить какой-то набор твитов и постоянно с ним работать. Для решения этой задачи в данной работе была использована библиотека Hosebird Client (hbc) [26]. Это HTTP клиент для работы со Streaming Twitter API. Для работы с этой библиотекой на сайте Twitter в разделе для разработчиков [27] было создано приложение, которому был предоставлен доступ к Streaming API с помощью нескольких токенов. Далее было написано приложение, получающее твиты на английском языке и отправляющее их в кластер Apache Kafka, в котором сообщения сохранялись и потом потреблялись приложениями, написанными для Apache Storm и Apache Spark. Код получения твитов находится в методе `sendFromTwitterToFile` класса `Main` в репозитории [28].

Далее все полученные твиты записывались в файл, чтобы для всех тестов использовался один и тот же набор данных. При каждом запуске задачи все твиты читались из этого файла и отправлялись в кластер Apache Kafka. Все твиты передаются в формате json, а преобразовываются в нужную форму уже при работе движков. Для большей приближенности к реальным системам поток твитов, поступающий в Kafka ограничивался с помощью переменной rate. В нее записывалось число отправляемых в Kafka сообщений в секунду. Исходный код продюсера, отправляющего твиты в кластер представлен в методе `sendFromFileToKafka` класса `Main` в репозитории [28].

Для теста производительности было собрано и записно в файл 4500000 твитов на английском языке.

D. Приложение для Apache Spark

Для написания приложения для Apache Spark необходимо создать несколько этапов обработки сообщения из Kafka. Для начала нужно создать DStream для получения сообщений, который создается также, как было описано в разделе V.A. Далее из полученного потока json-строк получаем твиты, как указано на рисунке 12.

```
JavaDStream<Status> statuses = messages.map((status) -> {
    try {
        return TwitterObjectFactory.createStatus(status._2());
    } catch (TwitterException ex) {
        return null;
    }
});
```

Рисунок 12 – Создание твита из json для Spark

В поток твитов попадают сообщения об их удалении. Такую информацию нужно отфильтровать, чтобы получить только опубликованные сообщения. В предыдущем листинге при выбросе исключения при преобразовании возвращается null. Все null-объекты нужно отфильтровать с помощью кода, представленного на рисунке 13.

```
JavaDStream<Status> filteredStatuses =
        statuses.filter((status) -> status != null);
```

Рисунок 13 – Фильтрация null-объектов для Spark

Последним этапом обработки является получения всех N-грамм каждого твита. Исходный код для выполнения этой задачи показан на рисунке 14.

```
JavaDStream<String> ngrams = filteredStatuses.flatMap(
        (status) -> NGrams.allNGrams(status.getText())
);
```

Рисунок 14 – Код получения всех N-грамм для Spark

Функция flatMap используется, когда возвращается несколько объектов за один вызов, как в этом случае, так как в каждом твите множество N-грамм.

E. Приложение для Apache Storm

Для написания приложения для Apache Storm необходимо создать Spout для получения данных из Kafka, Bolt для преобразования полученного json в объект твита, а также Bolt для поиска всех N-грамм. KafkaSpout был взят из специальной библиотеки storm-kafka, а его настройка была описана в разделе V.B. Теперь покажем, как выглядят обработчики.

Обработчик преобразования из json в объект, передающий дальше по топологии только текст твита представлен на рисунке 15.

```
public class StatusFilterBolt extends BaseBasicBolt {
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        Status status;
        try {
            status = TwitterObjectFactory.createStatus(
                    tuple.getStringByField("str")
            );
        }
        catch (TwitterException e) {
            status = null;
        }
        if (status != null) {
            final Status finalStatus = status;
            collector.emit(new ArrayList<Object>()
                {{ add(finalStatus.getText());}}
            );
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {
        ofd.declare(new Fields("statusText"));
    }
}
```

Рисунок 15 – Код получения твита из json для Storm

Обработчик поиска всех N-грамм показан на рисунке 14.

```
public class NGramDetectionBolt extends BaseBasicBolt {
    @Override
    public void execute(Tuple input, BasicOutputCollector collector) {
        List<String> ngrams = NGrams.allNGrams(
                (String) input.getValueByField("statusText")
        );
        for (String ngram : ngrams)
            collector.emit(new ArrayList<Object>() {{ add(ngram); }});
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("ngram"));
    }
}
```

Рисунок 16 – Получение всех N-грамм для Storm

Далее строим топологию, присоединяя, друг к другу KafkaSpout, StatusFilterBolt, NGramDetectionBolt и публикуем ее также, как это было показано в разделе V.B. Каждому обработчику необходимо поставить степень параллелизма, равную 2, так как тестирование будет проходить на 2х серверах.

F. Приложение без использования движков

Для того, чтобы показать преимущества движков Apache Storm и Apache Strom было написано приложение, которое выполняет те же самые задачи на одной машине без использования каких-либо движков потоковой обработки. Код этого приложения представлен в классе `DefaultEngine` репозитория [29].

G. Сравнение производительности

Для того, чтобы сравнить производительность необходимо найти такое максимальное число твитов, поступающих в секунду, при котором время обработки каждого сообщения практически не изменяется со временем. В какой-то момент система достигает точки насыщения, когда сообщения поступают быстрее, чем они успевают обрабатываться. В таком состоянии обработка может завершиться с ошибкой, поэтому нужно найти такую скорость, при которой такое состояние не достижимо, но при этом система работает с максимально возможной скоростью. На рисунке 17 представлен график зависимости времени обработки сообщений от скорости входящего потока, показывающий описанную ситуацию. Искомое состояние находится около отметки 2500 операций в секунду. Этот график не привязан к каким-то конкретным измерениям, а просто показывает, как будет вести себя любая система при достижении насыщения.

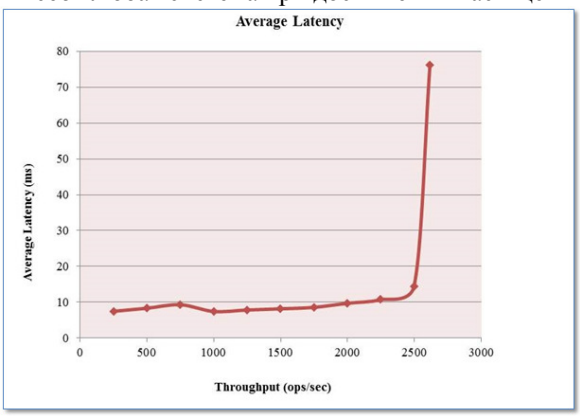

Рисунок 17 – График зависимости времени обработки каждого сообщения от скорости входящего потока

Теперь приведем результаты проведенных исследований. Сначала задача поиска всех N-грамм в каждом твите была запущена на программе без использования движков. Эта программа просто брала сообщения из Kafka с максимально возможной скоростью и обрабатывала их так же, как и в движках. Средняя скорость обработки получилась около 3650 твитов в секунду. При написании этой программы необходимо правильно подобрать такие настройки потребителя Kafka, как receive.buffer.bytes и max.partition.fetch.bytes. При достаточно маленьких значениях, которыми являются значения по умолчанию, эти настройки могут сильно уменьшить производительность приложения.

Приложение на кластере Apache Spark запускалось на 11 минут, а потом считалось количество сообщений, обработанных за последние 10 минут. Приложение было запущено 3 раза с частотой отправки 5675 сообщений в секунду с перерывом в 30 минут между запусками без перезагрузки виртуальных машин. Были получены результаты, представленные в таблице 1.

Таблица 1. Результаты теста приложения для Spark

Попытка	1	2	2
Средняя задержка при обработке, мс	20665	27725	23365
Количество обработанных сообщений	3279768	3234185	3284447

Spark предоставляет достаточно удобный интерфейс, который позволяет быстро понять выдерживает ли он заданную скорость. Этот интерфейс с результатами одного из запусков представлен на рисунке 18. Как видно из рисунка поток твитов поступал с частотой около 5675 сообщений в секунду. В каждую RDD попадали все сообщения, пришедшие за текущую секунду. Далее эта RDD обрабатывалась, занимая примерно 1 секунду времени, то есть Spark выдерживал заданную скорость. Это показывает и время, проведенное каждым сообщением в очереди (Scheduling Delay), которое через какое-то время стабилизировалось и было постоянным (около 20с). Увеличение скорости потока твитов приводило к тому, что время обработки каждой RDD было больше секунды, а Scheduling Delay постоянно увеличивалось. Из этого можно заключить, что Apache Spark с данной конфигурацией кластера и задачей может обрабатывать 5675 сообщений в секунду. Можно заметить, что это число заметно больше, чем при обработке не распределенной программой, что показывает целесообразность применения Spark. При этом при необходимости программу движка можно распараллелить на любое количество машин, в отличие от обычной программы.

Apache Storm имеет менее богатый интерфейс, чем Spark, поэтому делать выводы о том, выдерживает он заданный темп или нет, было сложнее. UI предоставляет количество обработанных событий за определенные промежутки времени (10 минут, 3 часа, 1 день и все время). Этот интерфейс представлен на рисунке 20. Для того, чтобы определить способен ли Storm обрабатывать заданное количество твитов, необходимо было запускать топологию, ждать около 14 минут, выбирать 10-минутный промежуток времени и делить полученное количество обработанных сообщений на 600 секунд (10 минут), чтобы понять выдерживает ли движок заданный темп. Если после деления получается число близкое к указанной скорости при запуске отправки сообщений в Kafka, то можно заключить, что Storm способен обработать это количество событий. Приложение было запущено 3 раза с частотой отправки 5725 сообщений в секунду с перерывом в 40 минут между запусками без перезагрузки виртуальных машин. Были получены результаты, представленные в таблице 2.

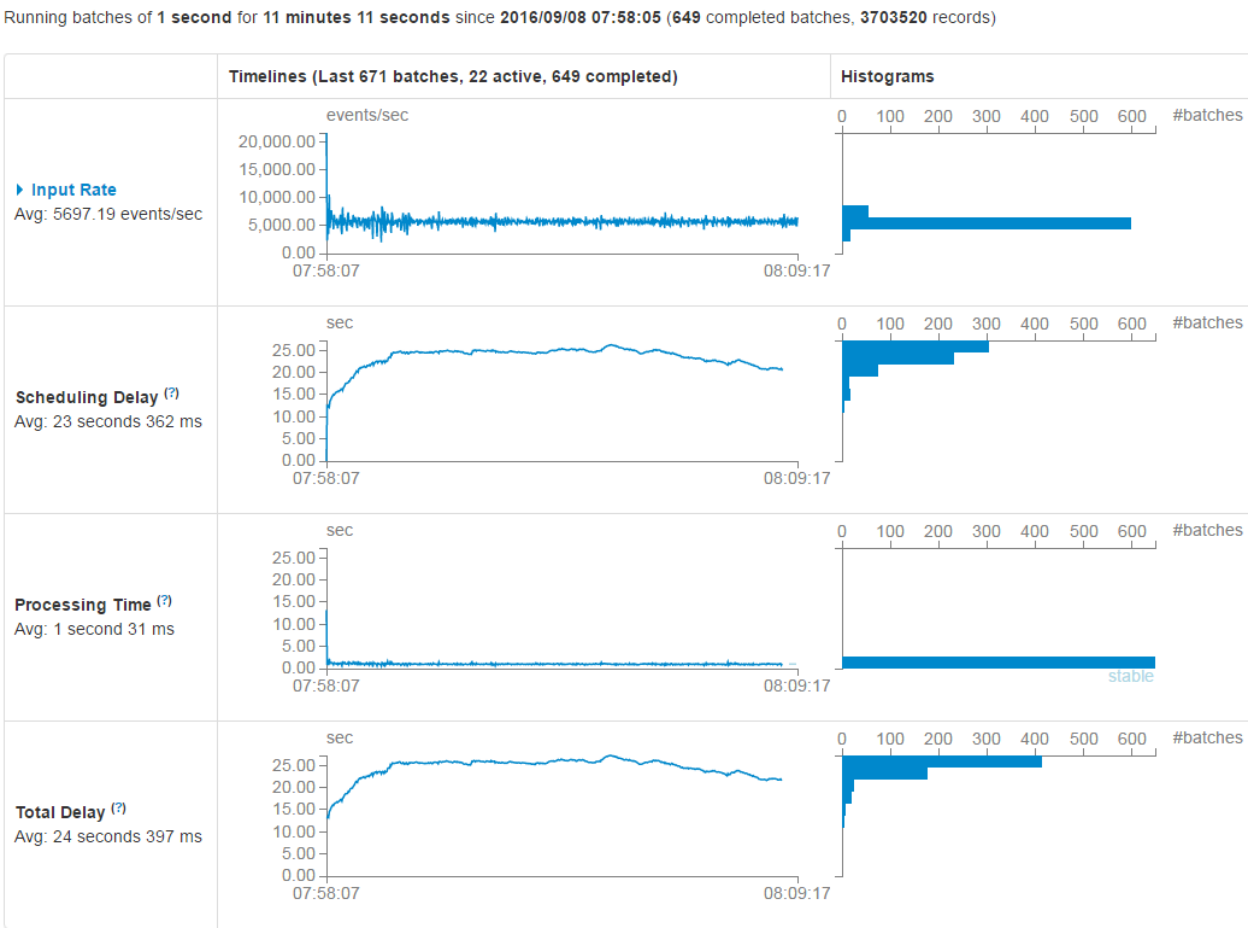

Рисунок 18 – Интерфейс Apache Spark с результатами

Таблица 2. Результаты теста приложения для Storm

Попытка	1	2	2
Средняя задержка при обработке, мс	827	796	827
Количество обработанных сообщений	3423055	3432100	3449140

Таким образом, можно сказать, что приложение выдерживало темп 5725 сообщений в секунду. При увеличении скорости входящего потока количество обрабатываемых сообщений не увеличивалось, что означает, что 5725 – искомая максимальная пропускная способность. Также нужно сказать, что при работе со Storm нужно аккуратно выставлять свойство `topology.max.spout.pending`, которое обозначает максимальное количество сообщений, уже выпущенных spout, но еще не полностью обработанных. Слишком маленькое значение может привести к ограничению обрабатываемого потока, даже несмотря на то, что Storm может его выдержать, а слишком большое значение может привести к тому, что обработка сообщений будет теперь неудачу по таймауту. Эту настройку нужно выставлять экспериментальным путем, начиная с небольшого значения (около 1000) и увеличивая ее с каждым запуском топологии. Значение, подходящее для топологии можно определить в тот момент, когда его увеличение не приводит к изменениям в работе топологии. В данном случае эта настройка была выставлена на 4000. В отличие от Storm, у Spark есть очень удобная настройка `spark.streaming.kafka.maxRatePerPartition`, которая четко ограничивает скорость получения твитов из кластера Kafka.

Во время проведения тестов производительности было замечено, что для машин AWS EC2 типа t2.micro спустя минут 15 и примерно 25 гигабайт трафика, пересылаемого по сети, этот трафик достаточно жестко ограничивается, поэтому тесты приходилось делать после того, как этот лимит снимется. Эта ситуация показана на рисунке 19. Также приходилось делать перерывы после того, как выданные CPU Credits на машинах обработчиках (на остальных они практически не тратились) заканчивались из-за особенностей инстансов t2, которые имеют базовую производительность и возможность ее увеличения на короткий промежуток времени до исчерпания CPU Credits.

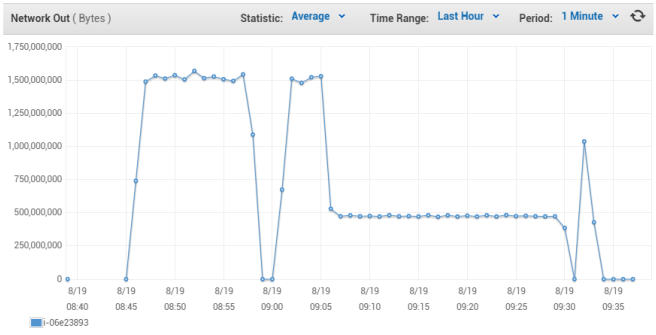

Рисунок 19 – Ограничение исходящего трафика на AWS EC2 t2.micro

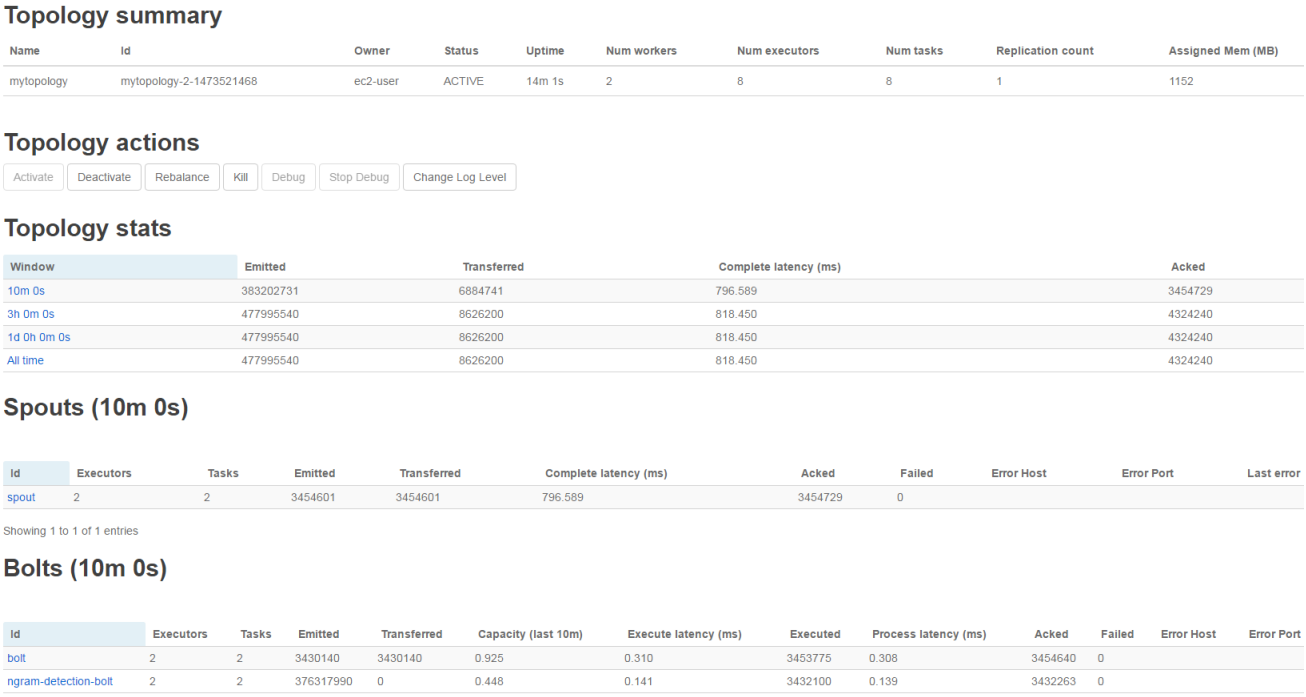

Рисунок 20 – Интерфейс Storm с результатами

H. Результаты проведенного сравнительного анализа

Общие результаты сравнительного анализа приведены в таблице 3.

Таблица 3. Результаты сравнения Spark и Storm

Критерий	Spark	Storm
Простота развертывания приложения	+	+
Простота настройки кластера	Средняя	Высокая
Потоковая модель	Нативная	Микро-пакетирование
Модель программирования	Декларативная	Композиционная
Гарантированность доставки сообщений	Exactly-once	At-least-once
Масштабируемость	+	+
Отказоустойчивость	+	+
Удобство пользовательского интерфейса	+	-
Модель обработки данных	Параллелизм данных	Параллелизм задач
Максимальная скорость обработки входящего потока, соб/сек	5675	5725
Величина задержки результата, мс	~817	~23919

По итогам проведенного сравнительного анализа двух движков потоковой обработки данных Apache Spark Streaming и Apache Storm, можно сказать, что на задаче поиска всех n-грамм в твите Storm показывает более высокую максимальную скорость обработки потока, а также, за счет использования нативной потоковой модели, более низкую задержку результата. Именно благодаря этим показателям Apache Strom был выбран для дальнейшей разработки системы кластеризации, несмотря на не совсем удобный интерфейс.

VII. Заключение

В данной работе было проведено подробный сравнительный анализ двух потоковых систем обработки данных Apache Storm и Apache Spark. Это сравнение включало в себя обзор основных концепций обоих движков, рассмотрение сложности развертывания и настройки приложений, модели программирования под каждый из них и тест производительности для конкретной задачи. По итогам проделанной работы Apache Storm был выбран для дальнейшего использования в разработке системы кластеризации сообщений социальной сети Twitter. Кроме разработки самой системы планируется подключение системы мониторинга приложений для движка Apache Storm для устранения его основного недостатка, связанного с достаточно бедным интерфейсом, предоставляющий маленький набор статистики.

Библиография

[1] Leah Hesla, “Particle physics tames big data” http://www.symmetrymagazine.org/article/august-2012/particle-physics-tames-big-data
[2] Hirak Kashyap, Hasin Afzal Ahmed, “Big Data Analytics in Bioinformatics: A Machine Learning Perspective” http://arxiv.org/pdf/1506.05101.pdf
[3] Eric D. Feigelson and G. Jogesh Babu, “Big data in astronomy” http://astrostatistics.psu.edu/2012Significance.pdf
[4] Saeed Shahrivari and Saeed Jalili, “Beyond Batch Processing: Towards Real-Time and Streaming Big Data” https://arxiv.org/ftp/arxiv/papers/1403/1403.3375.pdf
[5] Zeba Khanam and Shafali Agarwal, “Map-Reduce Implementations: Survey and Performance Comparison” http://airccse.org/journal/jcsit/7415ijcsit10.pdf
[6] Apache Hadoop http://hadoop.apache.org/

[7] Andrew C.Oliver, “Storm or Spark: Choose your real-time weapon” http://www.infoworld.com/article/2854894/application-development/spark-and-storm-for-real-time-computation.html
[8] Документация Apache Spark http://spark.apache.org/docs/latest/
[9] Документация Apache Storm http://storm.apache.org/releases/current/index.html
[10] Документация Apache Kafka http://kafka.apache.org/documentation.html
[11] Twitter Streaming API https://dev.twitter.com/streaming/overview
[12] Apache Flume https://flume.apache.org/
[13] Amazon Kinesis Streams https://aws.amazon.com/ru/kinesis/streams/
[14] Apache Zookeeper https://zookeeper.apache.org/
[15] Документация AWS EC2 https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
[16] Apache Hadoop YARN https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html
[17] Apache Mesos http://mesos.apache.org/
[18] Matei Zaharia, Tathagata Das, et al., “Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing” https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.pdf
[19] Sanket Chintapalli, Derek Dagit, Bobby Evans, et al., “Benchmarking Streaming Computation Engines at Yahoo!” https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
[20] Apache Flink https://flink.apache.org/
[21] Исходные коды теста производительности от Yahoo! https://github.com/yahoo/streaming-benchmarks
[22] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, et al., “Class-Based n-gram Models of Natural Language” http://www.aclweb.org/anthology/J92-4003
[23] Alberto Barr´on-Cede˜no and Paolo Rosso, “On Automatic Plagiarism Detection Based on n-Grams Comparison” http://users.dsic.upv.es/~prosso/resources/BarronRosso_ECIR09.pdf
[24] William B. Cavnar and John M. Trenkle, “N-Gram-Based Text Categorization” http://odur.let.rug.nl/~vannoord/TextCat/textcat.pdf
[25] David Sundby, “Spelling correction using N-grams” http://fileadmin.cs.lth.se/cs/education/EDA171/Reports/2009/david.pdf
[26] Hosebird Client https://github.com/twitter/hbc
[27] Twitter Apps https://apps.twitter.com/
[28] Исходный код программы-продюсера, отправляющей твиты в Kafka https://github.com/GorshkovNikita/kafka-test
[29] Исходный код программ для движков Spark и Storm https://github.com/GorshkovNikita/streaming-engines-comparison
[30] Jonathan Leibiusky, Gabriel Eisbruch and Dario Simonassi, “Getting Started with Storm”
[31] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, “Learning Spark”

Analysis posts of the social network Twitter with the stream processing systems Apache Spark and Apache Storm

Gorshkov N.A, Denisiov V.S.

Abstract — **The article discusses the comparison streaming processing systems Apache Storm and Apache Spark in the problem analysis the social network Twitter posts. At first, it describes the basic concepts of engines, their settings and launching applications. Then specific problems of tweets analysis are considered, as well as the structure of the cluster on which the performance test was carried out. In conclusion, the findings were made on the applicability of Storm and Spark for the considered problems.**

Keywords — **Apache Storm, Apache Spark, Apache Kafka, big data, stream processing, Twitter, AWS**