УДК 681.3:658.56

А.Н. Глибовец, Я.О. Дмитрук

Эффективность применения языков программирования в фреймворке *Apache Hadoop* с использованием *MapReduce*

Исследована эффективность использования различных языков для фреймворка *Apache Hadoop* с целью обработки больших коллекций данных на базе модели *MapReduce*. Акцент сделан на анализе скорости выполнения программ в *Hadoop*-кластере. Проведено сравнение различных проектов по экосистеме *Hadoop* для распределенных вычислений. Описанные эксперименты подтвердили преимущество использования *Apache Spark*. Установлено, что преимущество в скорости *MapReduce*-программ, написанных на *Java*- или другом *JVM*-языке, существенны.

Ключевые слова: *BigData*, *MapReduce*, *Apache Hadoop*, *Spark*, *Java*, *Pyton*, *Scala*.

Досліджено ефективність використання різних мов програмування у фреймворку *Apache Hadoop* для обробки великих колекцій даних з використанням моделі *MapReduce*. Акцент зроблено на аналізі швидкості виконання програм у *Hadoop*-кластері. Проведено порівняння різних проектів із екосистеми *Hadoop* для розподілених обчислень. Описано експерименти, які підтвердили переваги використання *Apache Spark*. Встановлено, що перевага у швидкості *MapReduce*-програм, написаних на *Java*- або іншій *JVM*-мові над іншими, є суттєвою.

Ключові слова: *BigData*, *MapReduce*, *Apache Hadoop*, *Spark*, *Java*, *Pyton*, *Scala*.

Введение. Потребность в оптимизации обработки больших объемов данных (*BigData*) привела к разработке новых программных решений. Одним из таковых стала модель распределенных вычислений *MapReduce*, обеспечивающая эффективную обработку и генерирование больших наборов данных [1]. Цель настоящей статьи – выбор языка программирования и инструментрия, которые будут использованы в задачах построения поисковых систем, в часности при индексировании массивов данных.

MapReduce (*MR*) называют подходом к обработке данных, моделью распределенных вычислений, парадигмой обработки данных, парадигмой программирования [2]. Будем использовать следующее определение: *MR* – модель распределенных вычислений, разработанная *Google*, используемая для организации параллельных вычислений с очень большими наборами данных в компьютерных кластерах.

Эта модель базируется на идеях функционального программирования. Двумя часто применяемыми функциями высшего порядка есть *map* и *fold*. Первая функция в качестве аргументов принимает некоторую функцию *m* и список *L*. Далее, к каждому элементу списка *L* применяется функция *m*, и как результат возвращается модифицированный список. Вторая функция приводит структуру данных к единому атомарному значению. *MR* использует подобную аналогию: фаза *Map* соответствует функции *map*, а фаза *Reduce* – функции *fold*. Конечно, *MR* работает в кластере, где операции распределяются между многими машинами, и является более сложной, но корреляцию с функциональным подходом можно увидеть. Вычисления выполняются над множеством входных пар *ключ–значение*, в результате каждого из них образуется новое множество из пар *ключ–значение*. Для вычислений используются две функции: *Map* и *Reduce*. Они явно определяются разработчиком.

Пары *ключ–значение* формируют базовую структуру данных в *MR*. Ключи и значения могут быть примитивами, например натуральными числами, числами с плавающей точкой, строкой или сложными структурами – списки, массивы и др. Программист может также определить и собственную структуру данных. На-

ложение структуры *ключ–значение* на произвольную коллекцию данных есть одним из существенных этапов проектирования *MR*-алгоритма. Например, для коллекции *Web*-страниц ключом могут быть *URL*-страницы, а значением *HTML* – содержимое страницы [3].

В функции распределения (*partitioner*) разработчик может явно указать необходимое число редюсеров (задач *Reduce*). Данные распределяются между этими задачами с использованием некоторой функции разделения от значений промежуточного ключа. По умолчанию используется функция хеширования (например, *hash* (*key*) *mod R*). Однако пользователи *MR* могут специфицировать и собственные функции разделения [2].

В некоторых случаях в результатах задачи *Map* содержится много одинаковых значений промежуточного ключа, а определенная разработчиком задача *Reduce* является коммутативной и ассоциативной. В таких случаях пользователь может определить дополнительную функцию-комбинатор (*combiner*), выполняющую частичную агрегацию таких данных до их передачи по сети. Функция *combiner* выполняется на той же машине, что и задачи *Map*. Обычно для ее реализации используется тот же код, что и для реализации функции *reduce*. Единственное различие между функциями *combiner* и *reduce* состоит в способе работы с их результирующими данными. Результаты функции *reduce* записываются в окончательный файл результатов. Результаты же функции *combiner* помещаются в промежуточные файлы, впоследствии пересылаемые в задачи *Reduce*. Применение *MR* предусматривает использование распределенной файловой системы.

Более детальное описание выполнения *MR*-программы представлено в [2].

Сначала *MR* разбивает исходный файл на *M* частей заданного размера. Далее запускается основная программа сразу на нескольких узлах кластера. Один из узлов становится *распорядителем* (*master*), а остальные – *исполнителями* (*worker*). Распорядитель назначает работу исполнителям, для выполнения *M* задач *Map* и *R* задач *Reduce*.

Исполнитель, которому назначена задача *Map* (*mappers*), читает содержаимое соответствующей группы, разбирает пары *ключ–значение* входных данных. Промежуточные пары *ключ–значение*, образованные функцией *Map*, собираются в буфере основной памяти. Периодически буферизированные пары, разделенные на *R* областей функцией распределения, записываются в локальную дисковую память исполнителя. Координаты этих сохранившихся буферизуемых пар направляются распорядителю для передачи исполнителям задачи *Reduce* (*reducers*). *I*-й редюсер получает координаты всех *i*-х областей буферизованных пар, образованных всеми *Map*-исполнителями.

После получения этих координат от распорядителя *Reduce*-исполнитель с использованием механизма отдаленных вызовов процедур переписывает данные с локальных дисков *Map*-исполнителей в свою память или на жесткий диск (в зависимости от объема данных). После переписи всех промежуточных данных выполняется их сортировка по значениям промежуточного ключа для образования групп с одинаковым значением ключа. Если объем промежуточных данных слишком велик для выполнения сортировки в основной памяти, используются внешние сортировки.

Далее *Reduce*-исполнитель организует цикл по отсортированным промежуточным данным и для каждого уникального значения ключа вызывает функцию пользователя *Reduce*, которая получает в качестве аргументов ключ и связанные с ним значения. Результирующие пары функции *Reduce* добавляются в окончательный результирующий файл данного *Reduce*-исполнителя.

После успешного завершения выполнения задания *MR*-результаты размещаются в *H* файлах распределенной файловой системы. Обычно нет необходимости объединять их в один файл, так как часто полученные файлы используются как входные для запуска следующей *MR*-задачи или в каком-нибудь другом распределенном приложении, которое может получать входные данные из нескольких файлов.

Apache Hadoop и связанные с ним проекты

Apache Hadoop – это фреймворк с открытым кодом, написанный на *Java* для распределенного хранения и вычисления больших коллекций данных в кластере [4]. Базовыми модулями *Apache Hadoop* являются: *Hadoop Common* – модуль, который содержит библиотеки и утилиты для других модулей; *HDFS* (*Hadoop Distributed File System*) – распределенная файловая система; *Hadoop Yarn* – модуль управления ресурсами кластера для выполнения приложений пользователя; *Hadoop MR* – модель программирования для обработки больших коллекций данных.

Кроме упомянутых составляющих в *Hadoop* могут входить и другие модули. Часто говорят, что эти модули входят в «экосистемы» *Hadoop*, и устанавливаются на его верхнем уровне. Примерами проектов, входящих в экосистемы *Hadoop*, являются *Apache Hive, Apache Pig, Apache HBase, Apache Spark* и др.

Apache Hadoop имеет много связанных проектов в своей «экосистеме», созданных для упрощения работы в этом фреймворке. Все имеют преимущества и недостатки. Далее будут рассмотрены только те проекты экосистемы *Hadoop*, которые используются в этой статье.

Apache Hive – распределенное хранилище данных, построенное на вершине *Hadoop*, которое облегчает процесс управления данными для разработчика. *Hive* предоставляет язык для написания запросов *HiveQL*, подобный *SQL*. Во время выполнения *HiveQL*-запрос переводится в несколько *MR*-работ (*job*) [5].

HiveQL – декларативный язык, упрощающий процесс программирования. Однако разработчик может только надеяться, что его запрос будет эффективно реализован. Именно поэтому в данной статье проверяется, будет ли время выполнения *MR*-программы на *Hive* таким же, как программы на *Java*.

Apache Spark – высокопроизводительный фреймворк для обработки данных, хранящихся в кластере *Hadoop*. Он может выполняться на узлах кластера *Hadoop* как с помощью *Hadoop YARN*, так и в самостоятельном режиме. Поддерживается обработка данных в хранилищах *HDFS*, *HBase*, *Cassandra*, *Hive* и в любом формате ввода *Hadoop* (*InputFormat*) [6].

В сравнении с предоставленным в *Hadoop* механизмом *MR*, продуктивность работы *Spark* ощутимо зависит от сложности задачи и размера данных и может обеспечивать в 100 раз большую производительность при обработке данных в оперативной памяти и в 10 раз – при размещении данных на дисках. Это связанно с тем, что *Spark* строит граф выполнения сложных задач и позволяет с меньшими потерями восстанавливаться после сбоев, а также эффективно использует кеширование промежуточных результатов. *Spark* может использоваться как в типовых сценариях обработки данных в архитектуре *MR*, так и для реализации специфических методов, таких как потоковая обработка *SQL*, интерактивные и аналитические запросы, решение задач машинного обучения и работа с графами [6]. В 2014 г. *Apache Spark* установил рекорд при сортировке 100 Тб данных [7].

Hadoop может быть развернут как на персональных компьютерах, так и в виде кластера в облаке. *HDInsight* – это сервис развертывания *Hadoop*-кластера в облаке *Microsoft Azure*. *HDInsight* в настоящее время предоставляет дистрибутив *Hadoop*, основанный на *Windows Server*, разработанный совместно с *Hortonworks*. *HDInsight* разворачивается несколькими нажатиями мыши, и уже через 15–25 мин кластер будет готов к использованию. Разработчик может выбрать необходимое количество узлов и их конфигурацию. Следует отметить, что *Microsoft* работает в режиме предоплаты. Деньги будут сниматься за время, пока кластер работает (даже просто создан и не занимается вычислениями), а также за трафик. Чтобы остановить кластер, его придется удалять, а потом заново создавать. Преимущество – то, что *HDInsight* использует хранилища, существующие независимо, т.е. при удалении кластера данные хранилищ сохраняются.

Эксперименты с реализацией программ

Кластер с одним узлом. Для ознакомления с *Hadoop* часто рекомендуют использовать кластер с одним узлом. Проанализированы доступ-

ные дистрибутивы виртуальных машин с готовой конфигурацией: *Cloudera*, *Hortonworks*, *MapR*. Выбран образ виртуальной машины *Cloudera Quickstart VM*. Удобство использования этой виртуальной машины заключается в том, что в ней уже установлены *Hadoop*, *HDFS* и все необходимые для проведения экспериментов сервисы. Характеристики физической машины, на который запускался узел виртуальной машины: процесор – *Intel Core i*5-2410*M* (2.3 Ггц), 3 Гб оперативной памяти. Характеристики виртуальной машины: 2 Гб оперативной памяти, ОС – *Ubuntu*.

Предметом сравнения было время выполнения программ на языках *Java*, *Scala* и *Python*, реализовавших решения простой задачи: сколько раз встречается каждое слово во входной коллекции текстовых документов. Заметим, что все три программы были написаны в одном стиле, чтобы результаты сравнения были объективными. Например, текст программы *WordCount* на *Java* был взят из [7].

Скомпилируем *WordCount.java* и создадим *jar* (*Java ARchive*) (учитывая, что *HADOOP_HOME* – директория, куда установлен *Hadoop*, а *HADOOP_VERSION* – его версия):

$ *mkdir wordcount_classes*

$ *javac – classpath* ${*HADOOP_HOME*}/*hadoop*-${*HADOOP_VERSION*}-*core.jar – d wordcount_ classes WordCount.java*

$ *jar – cvf /usr/joe/wordcount.jar – C wordcount_classes/ .*

Запускаем программу в *Hadoop* так:

$ hadoop jar wordcount.jar org.myorg.WordCount myinput wordcount_java_output.

Для написания и запуска *WordCount* на *Scala* проделана следующая работа. Пусть файл *pom.xml* – основной файл описания проекта. Откроем командную строку и перейдем в директорию проекта, где хранится файл. С помощью *Maven* выполним команду *mvn clean package*. Выполнение программы инициируется запуском в командной строке виртуальной машины команды: *hadoop jar target / wordcount*-1.0-*job.jar / myinput / wordcount_scala_ output*.

Запуск программы на *Python* опишем более детально. Создадим файл *mapper.py* для хранения кода *Map*-функции:

```
#!/usr/bin/env python
import sys
def read_input(file):
    # Split each line into words
  for line in file:
    yield line.split()

def main(separator='\t'):
  data = read_input(sys.stdin)
  for words in data:
    # Process each word
    for word in words:
      print '%s%s%d' % (word, separator, 1)
  if __name__ == «__main__»:
      main()
```

Создадим файл *reducer.py*, в котором хранится код *Reduce*-функции:

```
#!/usr/bin/env python

# import modules
from itertools import groupby
from operator import itemgetter
import sys

def read_mapper_output(file, separator='\t'):
  for line in file:
    yield line.rstrip().split(separator, 1)

def main(separator='\t'):
    data = read_mapper_output(sys.stdin, separator=separator)
    # Group words and counts into 'group'
    #   Since MR is a distributed process, each word
    #   may have multiple counts. 'group' will have all counts
    #   which can be retrieved using the word as the key.
    for current_word, group in groupby(data, itemgetter(0)):
      try:
        total_count = sum(int(count) for current_word, count in group)
```

```
        print «%s%s%d» % (current_word, separator, total_count)
    except ValueError:
        # Count was not a number, so do nothing
        pass

if __name__ == «__main__»:
    main()
```

Запустим программу: *hadoop jar /usr/lib/hadoop*-0.20-*MR*/*contrib*/*streaming*/*hadoop-streaming*-2.0.0-*mr*1-*cdh*4.1.1. *jar -mapper mapper.py -reducer reducer.py -file mapper.py –file reducer.py – input <path> -output <path>*

Каждая из программ запущена для разных размеров входных данных: 8 Мб, 34 Мб, 61 Мб, 106 Мб и 203 Мб.

Эксперимент показал, что *MR*-программа на *Python* в *Hadoop* работает медленнее программы на *Java* или *Scala*. Время выполнения программы на *Java* и *Scala* почти одинаково (рис. 1). Следовательно, выбор языка программирования существенно влияет на время выполнения.

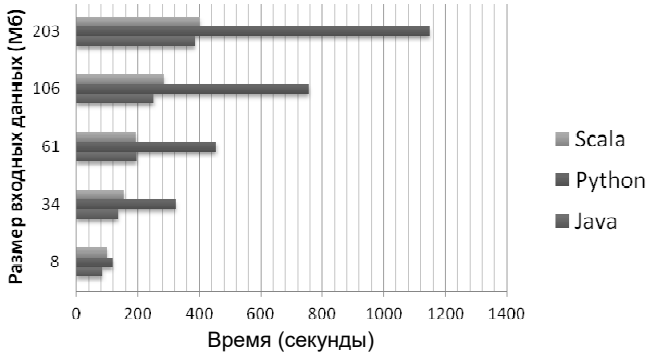

Рис. 1. Время выполнения програмы *WordCount*

Программы, написанные на *JVM*-не совместимых языках в *Hadoop*, будут работать дольше. Это объясняется тем, что во время потокового процесса (*Streaming*) *Java*-процесс (*task JVM*) передает входные пары *ключ–значение* в некоторый внешний процесс, где выполняется *map*- или *reduce*-функция, и по завершении возвращаются выходные пары *ключ–значение* в *Java*-процесс, что занимает определенное время.

Кластер с несколькими узлами. Для исследования создан кластер из трех узлов (виртуальных машин). Был избран *CDH* (*Cloudera Distribution Including Apache Hadoop*) как дистрибутив *Apache Hadoop* и связанных с ним проектов. Опустим настройки кластера, подробности ее – в открытом доступе в Интернет [8, 9].

Кластер состоял из трех узлов: *CDH*1, *CDH*2 и *CDH*3. Каждый узел – это отдельная виртуальная машина, для которой выделены определенные ресурсы, установлены различные сервисы и настроена сеть. Для каждого узла выделена оперативная память: *CDH*1 – 6 Гб; *CDH*2 – 4 Гб; *CDH*3 – 2 Гб. Сервисы, установленные на каждом узле, можно увидеть в таблице.

Название сервиса	Узлы		
	*CDH*1	*CDH*2	*CDH*3
Cloudera Manager Server	+		
Cloudera Manager Agent	+	+	+
Standby Name Node		+	
Name Node	+		
Data Node	+	+	+
Task Tracker	+	+	+
Job Tracker		+	
Node Manager	+	+	+
Resource Manager	+	+	
Balancer	+		
ZooKeeper Server	+	+	+
Job History Server		+	
Journal Node	+	+	+
ZooKeeper Failover controller	+	+	

Конфигурация машины, на которой был развернут кластер, имела следующие характеристики: *CPU – Intel Core i*7-3632*QM* (2.2 *GHz*), *RAM* – 16.0 *Gb* (*DDR*3), *Hard disk* – 1 Tb (*HDD*), *OS – Windows* 8 (64-*bit*). Узлы *CDH*1, *CDH*2 и *CDH*3 имели следующие *IP* адреса 192.168.1.201, 192.168.1.202 и 192.168.1.203 соответственно.

Для анализа была выбрана следующая задача. Необходимо посчитать количество сообщений для каждого *IP*-адреса по каждому дню, часу и уровню лога. Программа должна принимать параметры *start_timestamp* и *end_timestamp* в формате «*YYYYMMDD HH*», определяющие временной интервал логов, которые будут обрабатываться. Например, *start_timestamp* = «20150301 00:00» и *end_timestamp* = «20150401 23:59».

Коллекция системных логов *Syslog* генерировалась самостоятельно. Для создания логов

использовалась программа *Kiwi Syslog Generator*. Также, чтобы собирать созданные логи в *HDFS*-директорию, использовался *Apache Flume* [10]. Запускался *Flume*-агент по хосту *CDH*3 на *TCP*-порту 27001, который получал системные сообщения в режиме *real-time*. Входные и выходные файлы хранились в директории / *home* / *devclient* / *dev* / *flume* на хосте *CDH*3. Установленные настройки программы *Kiwi Syslog Generator* показаны на рис. 2.

Размер коллекции *Syslog* составил 1689173800 байт.

Выходные данные имели формат <Дата> <Время> <*IP* адрес> <Уровень лога> <Количество сообщений>. Пример выходных данных: 20150303 10 192.168.1.1 1 54.

Исследовались программные реализации решения поставленной задачи на *Java* (*MR*), *Hive*, *Spark* (v. 2.10).

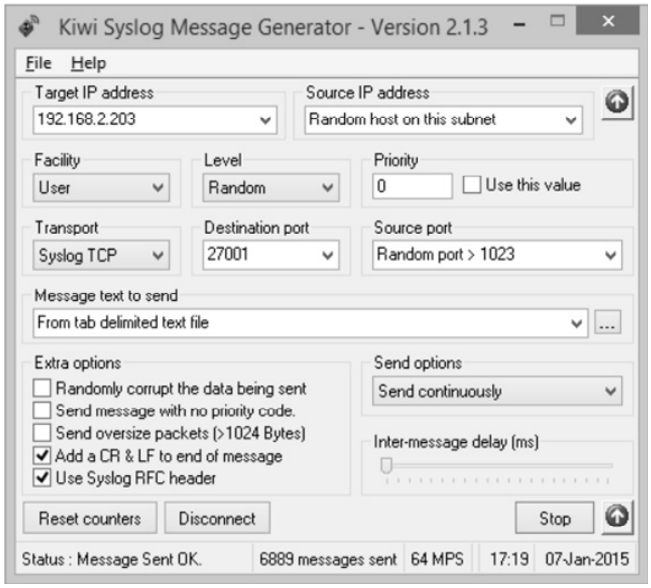

Рис. 2. Настройка программы *Kiwi Syslog Generator*

Рассмотрим реализацию данной *MR*-программы на *Hive*. В базе данных *training* создадим таблицу *syslog*, поддерживающую динамическое разделение (*partitioning*) по дате, и использующую *org.apache.hadoop.hive*. *serde*2.*RegexSerDe* для считывания системных логов. Сгенерированные системные логи содержатся в / *input_data* / *flume* / *syslog*. Следующий код выполняет описанное выше:

```
    USE training;
CREATE EXTERNAL TABLE IF NOT EXISTS
training.syslog(
  priority STRING,
  month STRING,
  day STRING,
  time STRING,
  host STRING,
  process STRING,
  tag STRING,
  message STRING)
    PARTITIONED BY (dt STRING)
ROW FORMAT SERDE 'org.apache.hadoop.
hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  «input.regex» = «<(\\d+)>(\\w{3}) (\\d+)
(\\d+:\\d+:\\d+) (\\S+) (\\S+) Original Ad-
dress=(\\d+\\.\\d+\\.\\d+\\.\\d+) (.*)» )
    LOCATION '/input_data/flume/syslog';
```

Напишем сначала простой запрос, который считает количество системных логов по разделу:

```
    USE training;
SELECT dt, COUNT(*)
FROM syslog
WHERE syslog.dt >= '2015-03-15' AND syslog.dt
<= '2015-03-16'
GROUP BY dt;
```

На выходе получим данные в формате <дата> <количество сообщений>. Очевидно сходство с *SQL*-запросами.

Код для реализации поставленного условия задачи (файл *hive_task*_2_3_1.*hql*):

```
    USE training;
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
    FROM syslog log
INSERT OVERWRITE TABLE syslog_count
PARTITION (date, hour)
  SELECT COUNT(*),
    log.tag,
    ((CAST(log.priority AS INT))-8),
    log.dt,
    substr(log.time, 1,2)
  WHERE
     unix_timestamp(concat(log.dt, ' ', log.time),
'yyyy-MM-dd HH:mm:ss') >=  unix_timestamp
```

(*concat*('${*hiveconf*:*start_timestamp*}', ':00'), *'yyyyMMdd HH*:*mm*:*ss'*)
AND
unix_timestamp(*concat*(*log.dt*, ' ', *log.time*), *'yyyy-MM-dd HH*:*mm*:*ss'*) <=
unix_timestamp(*concat*('${*hiveconf*:*end_timestamp*}', ':00'), *'yyyyMMdd HH*:*mm*:*ss'*)
GROUP BY log.tag, ((*CAST*(*log.priority AS INT*))-8), *log.dt*, *substr*(*log.time*, 1,2);

Результат записывается в таблицу *syslog count*, созданную следующим образом:

USE training;
CREATE TABLE IF NOT EXISTS syslog_count (
message_count INT,
ip_address STRING,
log_level INT)
PARTITIONED BY (*date STRING*, *hour STRING*);

Наконец, для запуска скрипта с файла *hive_2_3_1.hql* создадим *shell* файл *hive_task_2_3_2.sh*:

#!/*bin*/*bash*
hive --hiveconf start_timestamp=«$1» *--hiveconf end_timestamp*=«$2» *-f hive_task_2_3_1.hq*l

Запуск программы выглядит так:

$./*hive_task_2_3_2.sh*'2015030300:00'20150426 23:59'

В ходе эксперимента все три программы выполняли одну и ту же задачу в равных условиях. Каждая программа запускалась независимо. Размер выходной коллекции составил 4203326 б.

Каждую программу запускали дважды. Отметим, что во всех случаях программа успешно завершала свою работу и давала правильные результаты. На рис. 3 приведено лучшее время выполнения каждой программы. В результате эксперимента быстрее выполнила работу программа на *Spark* (за 482 с), а программа, написанная на *Hive*, выполнилась на 11 с медленнее, чем программа, написанная на *MR* с одним редюсером, которая завершилась через 606 с.

Однако в ходе экспериментов программа на *Spark* демонстрировала нестабильное время выполнения работы. На рис. 4 показана гистограмма, которая демонстрирует время выполнения программы на *Spark* в двух случаях, которые назовем *запуск* 1 и *запуск* 2. В первом случае программа выполнялась почти в 1,5 раза дольше, чем во втором. При первом запуске менеджер ресурсов много раз прекращал работу из-за недостатка оперативной памяти. Сначала прекращал работу менеджер ресурсов на *CDH*1; тогда менеджером становился *CDH*2, поскольку он настроен как *active-standby*; дальше *CDH*2 тоже аварийно завершал свою работу. Таким образом, менеджер ресурсов переключался между *CDH*1 и *CHD*2 несколько раз. Во втором случае был перезагружен компьютер хост-кластера и максимально выключены лишние процессы, чтобы освободить больше оперативной памяти. На этот раз менеджер ресурсов работал стабильнее. Отметим, что программа на *Spark* требует значительно большего количества оперативной памяти в сравнении с *MR*.

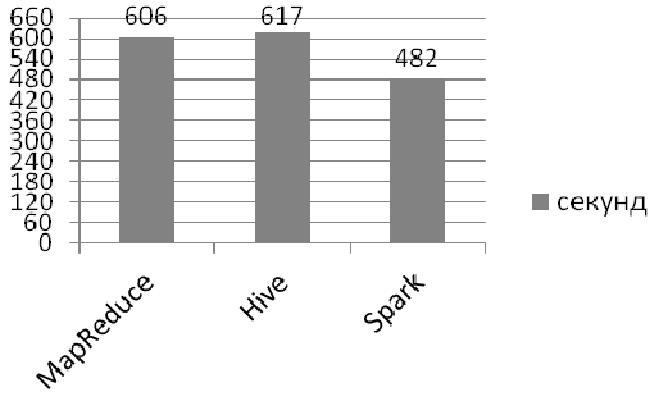

Рис. 3. Лучшее время выполнения каждой программы

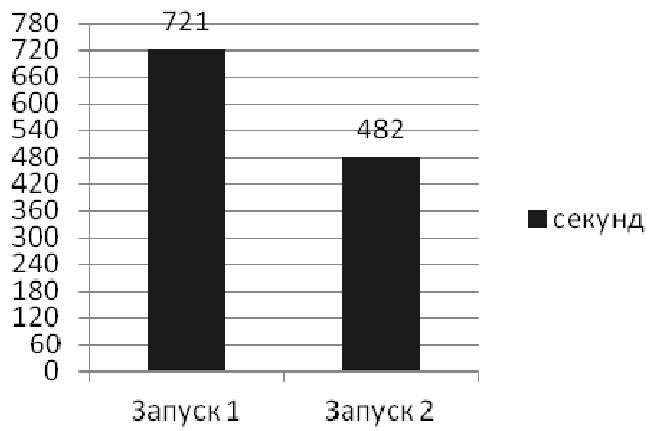

Рис. 4. Гистограмма времени выполнения программы на *Spark*

Исследовано также влияние количества редьюсеров на время выполнения *MR*-программы. На рис. 5 изображена гистограмма времени выполнения программы на *MR* с использованием одного редюсера и пяти редюсеров во время второго запуска. На примере этой задачи можно увидеть, что программа *MR*, написанная на *Java*, в которой разработчик поставил пять редюсеров, работала дольше на 186 с, чем программа с одним редюсерем, и на 175 с дольше программа на *Hive*.

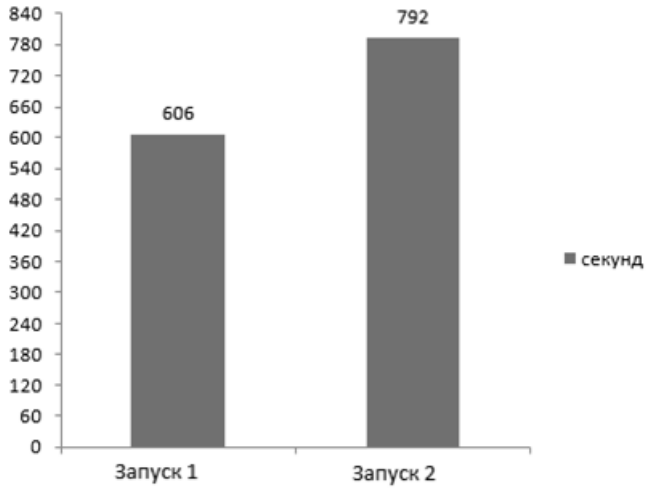

Рис. 5. Гистограмма времени выполнения программы на *MR*

Руководствуясь результатами экспериментов, можно сделать следующие выводы. Программа на *Spark* работает быстрее программ, использующих чистый фреймворк *MR* или *Hive*. При больших входных коллекциях *Spark* может быть даже в несколько раз быстрее, чем его конкуренты. *Spark* максимально использует оперативную память, благодаря чему достигается лучшая скорость. С другой стороны, если кластер имеет мало ресурсов, то могут выходить из строя некоторые сервисы, восстановления которых необходимо будет ждать для продолжения программы. *MR* имеет высокую отказоустойчивость. *MR*-программы, написанные на *Java*, быстрее работают, чем программы на *Hive*. Написание программ на *Hive* не требует глубоких знаний программирования, и выглядит лаконичнее. Однако *Hive* стоит использовать для написания запросов над структурированными данными (аналогично реляционным базам данных). В *Spark* много уже реализованных функций. При написании *MR* на *Java* придется писать гораздо больше кода. Количество редьюсеров влияет на скорость выполнения *MR*-программы. По умолчанию используется один редьюсер. Общих рекомендаций нет, разработчик часто подбором определяет их количество.

Использование кластера в облаке. Развернем кластер *HDInsight* в облаке *Microsoft Azure* со следующими характеристиками: *Data Node* – 4 по 7 Гб оперативной памяти и четыре ядра; *Head Node* (предоставляются по умолчанию) – 2 по 7 Гб оперативной памяти и четыре ядра.

Одна из программ, запущенная на этом кластере, была уже упомянута – программа *WordCount*, написанная на *Java*. Скорость выполнения этой программы для разного размера входных данных показана на рис. 6. Если сравнить скорость выполнения программы *WordCount* на этом кластере и кластере с одним узлом, который упоминался ранее, то увидим, что за примерно аналогичное время (385 с) данный *HDInsight*-кластер обработал коллекцию размером 3891 Мб, а кластер с одним узлом – лишь 203 Мб.

Рис. 6. Скорость выполнения программы *Word Count*

Эксперименты подтвердили и известное положение, что *Hadoop* следует использовать для обработки очень больших данных.

Заключение. Исследовано влияние выбора языка и использование дополнительных фреймворков на скорость выполнения программ в экосистеме *Apache Hadoop*.

Эксперементально установленно, что лучшие результаты показал *Apache Spark*, что *MR*-программы в *Apache Hadoop* лучше писать на *Java* или другом *JVM*-совместимом языке, чем на *Python*. Преимущество в скорости может быть в несколько раз. При проведении экспериментов

хотелось бы оперировать большими обьемами данных, но ограничения были связаны с высокими затратами на аренду облака *Microsoft Azure*. Но, несмотря на это, скорость обработки данных будет больше при больших входных коллекциях, т.е. *Hadoop* не целесообразно использовать для работы с небольшими обьемами данных.

1. *Dean J.*, *Ghemawat S.* MR: Simplified Data Processing on Large Clusters. Retrieved 2004. – http://static.googleusercontent.com/media/research.google.com/ru//archive/MR-osdi04.pdf
2. *Кузнецов С. MR*: внутри, снаружи или сбоку от параллельных СУБД, 2010. – http://citforum.ru/database/articles/ dw_appliance_and_mr/2.shtml#2.1
3. *Lin J.*, *Dyer C.* Data-Intensive Text Processing, 2010. MR: University of Maryland, College Park. – https://**lin**tool.github.io/**MR**Algorithms/**MR**-book-final.pdf
4. *White T.* Hadoop: The Definitive Guide. O'Reilly, 2012. – http://cdn.oreillystatic.com/oreilly/booksamplers/9781449311520_sampler.pdf
5. *Miner D.*, *Shook A.* MR: Design Patterns. O'Reilly, 2012. – http://www.nataraz.in/data/ebook/ hadoop/MR_design_patterns.pdf
6. *Apache* Spark. – http://uk.wikipedia.org/wiki/Apache_Spark
7. *Xin R.* World record set for 100 TB sort by open source and public cloud team, 2015. – http://opensource.com/business/15/1/apache-spark-new-world-record
8. *Hadoop*, Ч. 1: развертывание кластера. – https://habrahabr.ru/company/selectel/ blog/198534/
9. *Установка* кластера *Hadoop* (*CDH*) на *Debian*, Ч. 1 – https://bigdata-intips.blogspot.com/2015/10/hadoop-cdh-debian-1.html
10. *Apache* Flume. – //flume.apache.org/

Поступила 25.08.2016

Тел. для справок: +38 044 425-0245 (Киев)
E-mail: *andriy@glybovets.com.ua*, *yaroslav.dmytruk@gmail.com*

UDC 681.3:658.56

Glybovets A.N., Dmytruk Ya.O.

The Effectiveness of Programming Languages in the Apache Hadoop MapReduce Framework

Keywords: BigData, MapReduce, Apache Hadoop, Spark, Java, Pyton, Scala.

The effectiveness of the different languages for Apache Hadoop framework to process large data collections based on the MapReduce model is discussed.

Apache Hadoop is used in many industrial projects all over world such as Facebook and Yahoo!. It provides the ability to process different tasks effectively and reliably on the cluster to handle the huge amounts of data. MR model allows the developers to ignore the complex architectures by cluster management, and immediately to develop a program.

This work investigates the influence of the programming language on the speed of the program in the Apache Hadoop framework.

The subject of comparison is the execution of programs in Java, Scala and Python that implements the solution of the simple problem: how long each word in the input collection of text documents is searched. All three programs, in spite of the language, is written in the same style, so that the comparison results are objective.

For the experiments, we have chosen the image of ClouderaQuickstart VM virtual machine. The easy use of this virtual machine is that it is already established Hadoop, HDFS, and other services. Also, a cluster of three nodes is created for the study. CDH is elected as the distribution of Apache Hadoop and related projects. The desired configuration on each node is set.

Each program is ran for the different size input: 8Mb, 34Mb, 61Mb, 106Mb and 203Mb.

During the experiments, the best results is showed by the program that is written in the Apache Spark. In addition, it is found that the MR program in the Apache Hadoop is better to write in Java or any other JVM languages than Python. An advantage in speed is obvious. Also, experiments shows that the processing speed is larger at higher input collections. So, it is not necessary to use Hadoop to work with small data.

●