**ЛАБОРАТОРНАЯ РАБОТА №2**

**MAPREDUCE**

**Цель работы:** формирование практических навыков использования парадигмы MapReduce для обработки больших данных.

**Задачи:**

1. Изучить подход MapReduce.
2. Изучить принципы работы Hadoop MapReduce.
3. Получить практические навыки реализации MapReduce задач.
4. Уметь обрабатывать большие текстовые файлов с помощью MapReduce.

Результатами работы являются:

* Входные файлы с данными
* MapReduce-программа, обрабатывающая данные согласно варианту задания
* Выходные файлы с результатами вычислений
* Подготовленный отчет

**КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ**

**Определение MapReduce**

В современном мире все большую роль играют технологии, обеспечивающие эффективную обработку больших массивов данных. Мировой объем информации увеличивается более чем в 2 раза каждые два года, что говорит о лавинообразном росте общего количества данных. Современные программные средства предъявляют серьезные требования к вычислительным ресурсам, значительно превышающие возможности отдельных компьютеров. Особое значение уделяется алгоритмам и методам, применяемым для обработки и анализа данных с использованием компьютерных кластеров, состоящих из сотен или тысяч узлов. Однако реализация процедур обработки данных на кластерных системах сопряжена с решением таких задач, как разбиение и распределение данных между процессорами, балансировка нагрузки, обработка отказов, сбор и агрегация промежуточных результатов.

Одним из самых эффективных методов обработки больших объемов данных в распределенных средах является парадигма MapReduce, предложенная компанией Google в начале 2000-х для сканирования и обработки большого количества страниц из сети Интернет. Впервые такая парадигма была реализована в составе распределенной файловой системы GFS (Google File System) и в высокопроизводительной нереляционной базе данных Big Table.

Данная модель отличается простотой и удобством использования, скрывает от пользователя детали организации вычислений на кластерной системе. Преимущество MapReduce заключается в том, что она позволяет распределенно выполнять операции предварительной обработки и свертки. Операции предварительной обработки работают независимо друг от друга и могут производиться параллельно. Аналогичным образом множество рабочих узлов осуществляют операцию свертки – для этого необходимо, чтобы все результаты предварительной обработки с одним конкретным значением ключа обрабатывались одним рабочим узлом в один момент времени.

Параллелизм также дает некоторые возможности восстановления после частичных сбоев серверов: если в рабочем узле, производящем операцию предварительной обработки или свертки, возникает сбой, то его работа может быть передана другому рабочему узлу (при условии, что входные данные для проводимой операции доступны). Пользователю достаточно описать процедуру обработки данных в виде нескольких функций, после чего система автоматически распределяет вычисления по кластеру, обрабатывает отказы машин, балансирует нагрузку и координирует взаимодействия между машинами.

**Цель MapReduce**

Входные данные обычно имеют большой объём, и вычисления приходится распределять по сотням или даже тысячам машин, чтобы иметь возможность решить задачу в разумный промежуток времени. Проблемы распараллеливания вычислений, распределения данных и обработки сбоев заставляют отказаться от простой модели вычислений с большим объёмом сложного кода. В Google отреагировали на эту сложность и разработали новую абстракцию, которая позволяет легко производить необходимые вычисления, но скрывает все детали распараллеливания, отказоустойчивости, распределения данных и балансировки нагрузки в своих библиотеках.

Вдохновением для этой абстракции были функции map и reduce, представленные во многих функциональных языках программирования. В Google осознали, что две эти операции применяются в большинстве задач обработки больших данных.

**Принцип работы модели**

В рамках концепции MapReduce предполагается, что данные организованы в виде некоторого набора упорядоченных записей, а их обработка происходит в три стадии: Map, Shuffle и Reduce (Рис. 1).

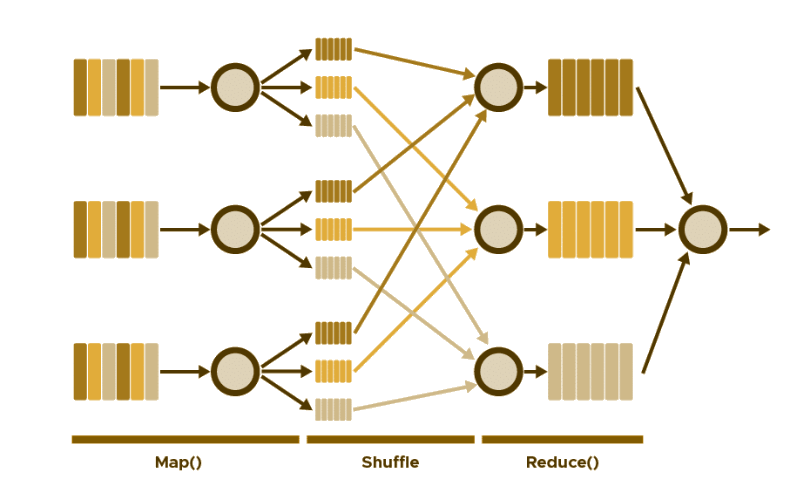


Рис. 1. Стадии MapReduce

*Стадия Map.* На этой стадии выполняется предварительная обработка и фильтрация данных при помощи функции Map, которую определяет пользователь. Принцип работы подобен операции Map в функциональных языках программирования – пользовательская функция применяется к каждой входной записи и возвращает множество пар ключ – значение. Все запуски функции работают независимо и могут работать параллельно, в том числе на разных машинах кластера. Функция Map, как правило, применяется на той же машине, на которой хранятся данные. Это позволяет снизить передачу данных по сети (принцип локальности данных).

*Стадия Shuffle*. На этой стадии вывод функции Map разбивается на специальные секции (корзины). Каждая корзина соответствует одному ключу вывода стадии Map. Кроме того, она принимает на вход совокупность записей, соответствующих данному ключу, и общее количество Reduce-задач, а возвращаемым значением является номер задачи, в которой обрабатывалась каждая запись. Каждая секция формируется на основе функции хеширования, которая вызывается для каждого ключа и зависит от определенных критериев, например, от номера задачи. Для ускорения процесса обработки информации очень часто на данной стадии применяют алгоритмы параллельной сортировки. В первую очередь они требуются в тех случаях, когда разные атомарные обработчики возвращают наборы с одинаковыми ключами, при этом правила сортировки на этой фазе могут быть заданы программно и использовать какие-либо особенности внутренней структуры ключей разделения (partition key).

*Стадия Reduce.* Каждая корзина со значениями, сформированная на стадии Shuffle, попадает на вход функции Reduce. Эта функция вычисляет финальный результат для каждой отдельной секции. Все запуски Reduce, как и функция Map, работают независимо и могут работать параллельно, в том числе на разных машинах кластера. Для некоторых видов обработки свертка не требуется, и каркас возвращает в этом случае набор отсортированных пар, полученных базовыми обработчиками.

Многие реализации MapReduce также поддерживают операцию combine. Combine, как и reduce, получает на вход записи с одним и тем же ключом, но не обязательно все. Запускается на каждом хосте перед reduce, чтобы уменьшить объем участвующих в сетевом обмене данных. Иногда в качестве combine можно использовать reduce (например, в wordcount).

**Применение MapReduce**

Парадигма MapReduce достаточна гибкая и может легко адаптироваться под разные типы задач, включая в себя дополнительные стадии обработки информации. Например, стадия Combine применяется в тех случаях, когда в результатах функции Map содержится значительное число повторяющихся значений промежуточного ключа, а определенная пользователем задача Reduce является коммутативной и ассоциативной. В таких случаях необходимо осуществить частичную агрегацию данных до их передачи по сети. Функция Combine выполняется на той же машине, что и задача Map. Результаты функции Combine помещаются в промежуточные файлы, которые впоследствии пересылаются в задачи Shuffle или Reduce.

Парадигма распределенных вычислений MapReduce в настоящее время широко используется не только для эффективной обработки больших объемов данных, но и для решения прикладных задач, связанных с расширенной обработкой текста, сортировкой данных, индексированием документов, вычислением индексов цитируемости, статистическим анализом, машинным обучением, обработкой изображений. Классифицировать многообразие этих задач только по области применения не представляется возможным по причине того, что многие области знаний тесно взаимодействуют между собой. Имеет смысл добавить другой критерий, связанный со стадиями обработки данных парадигмой MapReduce. В результате исследования было выделено четыре класса задач, при решении которых применяют данную модель распределенных вычислений.

**Виды MapReduce задач**

К первому классу, называемому MapReduce, необходимо отнести все возможные методы, алгоритмы, использующие в исследуемой парадигме минимум две стадии: Map и Reduce. При решении практических задач зачастую стадии Shuffle или Combine не нужны. В таком случае общая схема работы парадигмы MapReduce упрощается и будет иметь следующий вид:

1. В модель распределенных вычислений подается коллекция документов (записей).

2. Функция Map применяется к каждой паре входных данных и возвращает набор промежуточных пар.

3. MapReduce-контейнер группирует промежуточные значения, связанные с одним ключом, и передает эти значения функции Reduce. Она преобразует промежуточные значения в окончательный набор значений для данного ключа. Как правило, это одно агрегированное значение, например сумма.

К группе *MapOnly* относят задачи, для решения которых можно обойтись только стадией Map (Рис. 2). Примерами таких задач являются фильтрация данных (например, поиск нужной информации в лог-файлах по определенному критерию), преобразование данных (например, удаление определенного свойства в JSON-документе или перевод текста в нижний регистр), загрузка и выгрузка данных из внешнего источника (например, вставка записей в NoSQL-базу данных). В описанных типах задач пользователю требуется получить набор пар ключ – значение, поэтому другие стадии, такие как Combine, Reduce, ему не нужны.

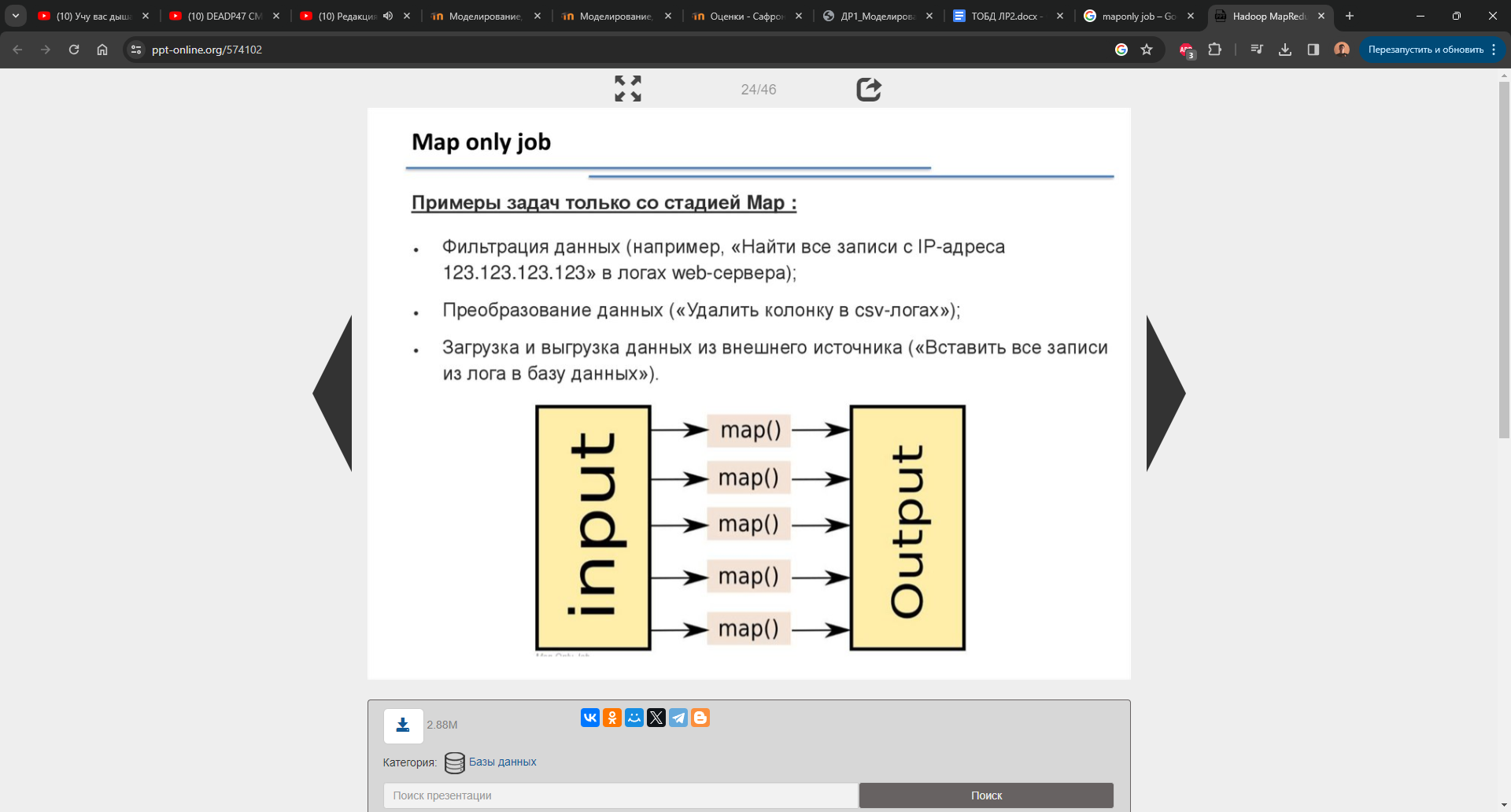


Рис. 2. MapOnly задача

*Цепочки MapReduce*. К данной группе относят ситуации, когда для решения определенных задач реализации одной MapReduce-модели недостаточно. Тогда их объединяют в цепочки, которые могут выполняться либо линейно, либо представлять собой более сложный направленный ациклический граф. Для линейной цепочки проще всего запускать задания одно за другим, дожидаясь успешного завершения задания перед запуском следующего. Если одно из заданий завершается неудачно, то последующие задания в конвейере выполняться не будут. Когда их последовательность сложнее линейной цепочки, то необходимо определенным образом организовать поток операций, учитывая информацию о зависимости между вызовами MapReduce-задач. Если одно из заданий завершается с ошибкой, планировщик должен прекратить анализ зависимостей и прервать выполнение всей цепочки задач (Рис. 2.3).

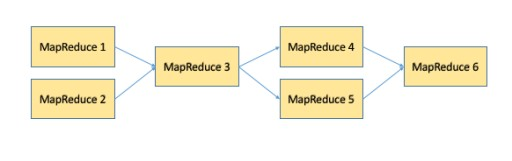


Рис. 3. Цепочка MapReduce

*ReduceJoin.* К данному классу относят задачи, в которых необходимо объединить содержимое нескольких документов по некоторому ключу в выходном потоке данных. Результат работы этих задач очень похож на принцип работы с реляционными базами данных, где часто используют очень удобную операцию Join, позволяющую совместно обрабатывать содержание некоторых таблиц, объединив их по некоторому ключу. Примером таких задач является объединение двух или более логфайлов сервера в один итоговый документ либо определение, на какой из двух серверов пользователь чаще заходит по его IP-адресу.

Модель ReduceJoin функционирует, используя следующий алгоритм (Рис. 4):

1. На вход поступает две коллекции документов (записей).

2. Каждая из коллекций запускает отдельную MapOnly-задачу, преобразующую входные данные к паре ключ – значение. В качестве ключа используется поле, по которому нужно объединять записи коллекций, а в качестве значений выступает пара Type (тип коллекции) и Value (любые дополнительные данные, привязанные к ключу).

3. Результат работы MapOnly подается на вход следующей модели MapReduce. Эта цепочка должна содержать пустую функцию Map, которая копирует входные данные. Далее на стадии Shuffle данные разделяются по ключам и подаются на вход функции Reduce в виде пары, где в качестве значения используется массив элементов Type и Value.

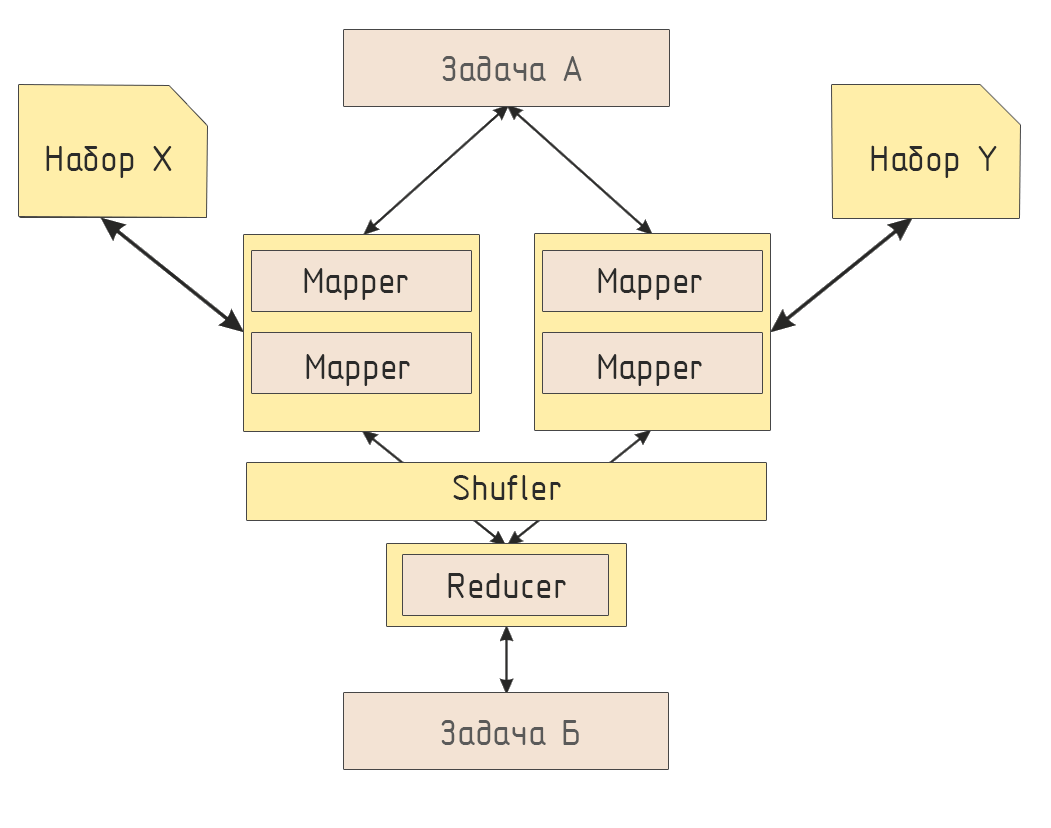


Рис. 4. ReduceJoin задача

**Особенности модели**

MapReduce накладывает ряд ограничений на реализующее её программное средство в связи с необходимостью автоматизировать распараллеливание, запуск и управление вычислениями на кластере. Кроме того, данная парадигма всегда требует полного сканирования данных, поэтому использование индексов здесь недопустимо. Это означает, что подход MapReduce плохо применим и требует дополнительных оптимизаций, когда необходимо получить ответ в режиме реального времени.

Эффективная реализация MapReduce невозможна без эффективной организации способа хранения данных на кластерной системе. Для этой цели используются распределенные файловые системы (DFS), обеспечивающие высокую производительность, масштабируемость, надежность и доступность данных. Эти файловые системы должны быть оптимизированы для хранения файлов большого размера, эффективного использования сетевых ресурсов и оптимизации под высокую агрегированную пропускную способность, нестандартный интерфейс файловой системы, а также ослабленную модель целостности данных, связанную с хранением слабоструктурированной или не структурированной информации.

Запуском MapReduce-задач на кластере должен управлять планировщик, который отслеживает состояние всех узлов и подбирает группу машин для выполнения задания. Вызовы функции Map распределяются между несколькими машинами путем автоматического разбиения входных данных, хранящихся в распределенной файловой системе, на несколько частей. Полученные порции данных могут обрабатываться параллельно различными машинами. Вызовы Reduce распределяются путем разбиения пространства промежуточных ключей на совокупность частей, определяемых с помощью заданной функции разбиения. Каждый из Reduce-процессов загружает со всех Map-процессов порции обработанных данных с соответствующими значениями промежуточных ключей, производит сортировку и объединение этих данных, после чего выполняет функцию Reduce. Результаты вычислений записываются в виде файлов в DFS.

Парадигма MapReduce используется многими компаниями, такими как: Google (служит для параллельных вычислений над очень большими, несколько петабайт, наборами данных в компьютерных кластерах), CouchDB (использует MapReduce для определения представлений поверх распределенных документов), MongoDB (позволяет применять MapReduce для параллельной обработки запросов на нескольких серверах), Apache Hadoop (фреймворк для разработки и выполнения распределенных программ), Nvidia (распараллеливание вычислений на видео-ядрах с использованием технологий CUDA), Яндекс (обработка и анализ данных Интернет-сайтов). Каждая компания имеет свои закрытые реализации моделей MapReduce, позволяющие выполнять задачи, написанные на языках Java, C++, C, Python, JavaScript, C#, Perl, Erlang, Ruby.

**ПРИМЕР MAPREDUCE ЗАДАЧИ**

Классическим примером MapReduce задачи является подсчет слов в большом количестве документов (Word count).

Задача формулируется следующим образом: имеется большой корпус документов. Задача – для каждого слова, хотя бы один раз встречающегося в корпусе, подсчитать суммарное количество раз, которое оно встретилось в корпусе.

Входной записью для MapReduce задачи будет являться каждый отдельный документ. Функция map обрабатывает каждое слово в документе и превращает один входной документ в набор пар (word, 1). Стадия shuffle прозрачно для пользователя группирует данные пары по словам (word, [1,1,1,1,1,1]). Функция reduce суммирует единицы, возвращая финальный результат для каждого слова (word, count).

**Способ №1. Hadoop Streaming**

Самый простой способ запустить MapReduce-программу на Hadoop –  воспользоваться streaming-интерфейсом Hadoop. Streaming-интерфейс предполагает, что map и reduce реализованы в виде программ, которые принимают данные с *stdin* и выдают результат на *stdout.*

Программа, которая исполняет функцию map называется *mapper.* Программа, которая выполняет reduce, называется, соответственно, *reducer.*

Streaming интерфейс предполагает по умолчанию, что одна входящая строка в mapper или reducer соответствует одной входящей записи для map.

Вывод mapper’a попадает на вход reducer’у в виде пар (ключ, значение), при этом все пары соответствующие одному ключу:

* Гарантированно будут обработаны одним запуском reducer’a;
* Будут поданы на вход подряд (то есть если один reducer обрабатывает несколько разных ключей – вход будет сгруппирован по ключу).

**Реализация на языке Python:**

*Файл mapper.py:*

*#!/usr/bin/python3*

import sys

import typing

def map\_line(

\_line: str

) -> typing.Generator[typing.Tuple[str, int]]:

for word in \_line.split():

yield word.lower(), 1

if \_\_name\_\_ == '\_\_main\_\_':

for line in sys.stdin:

for key, value in map\_line(line):

print(f'{key}\t{value}')

*Файл reducer.py:*

*#!/usr/bin/python3*

import sys

import typing

def reduce(

word: str,

values: typing.Iterable[int]

) -> typing.Tuple[str, int]:

return word, sum(values)

if \_\_name\_\_ == '\_\_main\_\_':

previous\_key = None

values = []

for line in sys.stdin:

key, value = line.split("\t")

if previous\_key and key != previous\_key:

result\_key, result\_value = reduce(

previous\_key, values

)

print(f'{result\_key}\t{result\_value}')

values = []

previous\_key = key

values.append(int(value))

if previous\_key is not None:

result\_key, result\_value = reduce(

previous\_key, values

)

print(f'{result\_key}\t{result\_value}')

Данные, которые будет обрабатывать Hadoop должны храниться на HDFS. Необходимо загрузить текст для задачи и загрузить его на HDFS. Для этого нужно воспользоваться командой *hadoop fs*:

hadoop fs -put lenta\_articles

**Запуск streaming-задачи**

Составим скрипт для запуска streaming-задачи.

*Файл mapred.sh:*

**#!/bin/bash**

MAPPER="mapper.py"

REDUCER="reducer.py"

INPUT=$1

OUTPUT="/user/hadoop/lab2\_output"

echo "[SCRIPT] Removing Output $OUTPUT..."

/usr/local/hadoop/bin/hdfs dfs -rm -r -f $OUTPUT

echo "[SCRIPT] Starting MapReduce Job For $INPUT..."

/usr/local/hadoop/bin/mapred streaming -input $INPUT -output $OUTPUT -mapper $MAPPER -reducer $REDUCER

/usr/local/hadoop/bin/hdfs dfs -head "$OUTPUT/part-00000"

Для запуска streaming-задачи выполним bash-скрипт, указав в качестве его аргумента входной файл, находящийся в HDFS. Например, запустим скрипт для заданного входного файла:

mapred.sh user/hadoop/lenta\_articles

Указанный скрипт выполняет последовательно следующие операции:

* удаление предыдущего результата выполнения скрипта;
* запуск streaming-задачи;
* вывод начала файла с результатом в консоль.

В указанном скрипте имеются следующие переменные:

* MAPPER - консольная команда, которая будет использоваться для map-стадии;
* REDUCER - консольная команда, которая будет использоваться для reduce-стадии;
* INPUT - папка с исходными данными на HDFS;
* OUTPUT - папка на HDFS, куда нужно положить результат.

Результат работы после успешного выполнения складывается на HDFS в папку, которую мы указали в поле OUTPUT. Просмотреть её содержание можно при помощи команды «hadoop fs -ls /user/hadoop/lab2\_output».

Сам результат можно получить следующим образом:

hadoop fs -text /user/hadoop/lab2\_output/\* | sort -n -k2,2 | tail -n5

с 41

что 43

на 82

и 111

в 194

**Способ №2. Java-приложение**

*Реализация на языке Java:*

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.

FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.

FileOutputFormat;

public class WordCount {

public static class TokenizerMapper

extends Mapper<Object, Text, Text,

IntWritable>{

private final static IntWritable *one* = new

IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value,

Context context) throws IOException,

InterruptedException {

StringTokenizer itr = new

StringTokenizer(value.toString());

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, *one*);

}

}

}

public static class IntSumReducer extends

Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new

IntWritable();

public void reduce(Text key,

Iterable<IntWritable> values, Context

context) throws IOException,

InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

public static void main(String[] args) throws

Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word

count");

job.setJarByClass(WordCount.class);

job.setMapperClass

(TokenizerMapper.class);

job.setReducerClass

(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass

(IntWritable.class);

FileInputFormat.addInputPath(job, new

Path ("hdfs://localhost/user/

hadoop/lenta\_articles"));

FileOutputFormat.

setOutputPath(job, new

Path("hdfs://localhost/user/hadoop/lent

a\_wordcount"));

System.*exit*(job.waitForCompletion(true)

? 0 : 1);

}

}

Этот класс делает абсолютно то же самое, что и пример на Python. Создаются классы TokenizerMapper и IntSumReducer, наследуемые от классов Mapper и Reducer, соответственно. Классы, передаваемые в качестве параметров шаблона, указывают типы входных и выходных значений. Нативный API подразумевает, что функции map на вход подаётся пара ключ-значение. Поскольку в данном случае ключ пустой – в качестве типа ключа определяется просто Object.

В методе Main резервируется mapreduce-задача и определяются её параметры – имя, mapper и reducer, путь в HDFS, где находятся входные данные и куда загрузить результат.

Для компиляции потребуются библиотеки дополнительные библиотеки hadoop. Для сборки следует использовать Maven. В итоге файл pom.xml (который используется maven’ом для описания сборки проекта) имеют следующий вид:

*<?*xml version="1.0" encoding="UTF-8"*?>*

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd"

>

<modelVersion>4.0.0</modelVersion>

<repositories>

<repository>

<id>cloudera</id>

<url>

https://repository.cloudera.com/artifactory/cloudera-repos/

</url>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-

common

</artifactId>

<version>2.6.0-cdh5.4.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-auth</artifactId>

<version>2.6.0-cdh5.4.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.6.0-cdh5.4.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>

hadoop-mapreduce-client-app

</artifactId>

<version>2.6.0-cdh5.4.2</version>

</dependency>

</dependencies>

<groupId>org.dca.examples</groupId>

<artifactId>wordcount</artifactId>

<version>1.0-SNAPSHOT</version>

</project>

Для запуска приложения необходимо собрать проект в jar-пакет:

mvn clean package

После сборки проекта в jar-файл запуск происходит похожим образом, как и в случае streaming-интерфейса:

yarn jar wordcount-1.0-SNAPSHOT.jar WordCount

Результат выполнения нативного приложения совпадает с результатом streaming-приложения из предыдущего способа:

hadoop fs -text lenta\_wordcount/\* | sort -n -k2,2

| tail -n5

с 41

что 43

на 82

и 111

в 194

**ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

Выполнить задание с помощью подхода MapReduce согласно варианту. В качестве входных текстовых файлов можно использовать книги в txt формате из библиотеки Project Gutenberg:

<https://www.gutenberg.org>.

Список стоп-слов:

<https://github.com/timothywarner/dp900/blob/main/sample-data/stopwords.csv> (вы можете использовать свои стоп-слова, объединив их в подобном CSV-файле).

**ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ**

Программа может быть реализована на любом языке высокого уровня, для которого существует поддержка работы с HDFS (Java, Python, Scala или др.). Имена файлов должны передаваться приложению в качестве ключей при вызове в терминале.

**ВАРИАНТЫ ЗАДАНИЙ**

1. Подсчитать количество строк в файле. Результат должен быть сохранен в файле в виде:

file\_name lines\_count

2. Подсчитать стандартное отклонение в длинах слов в нескольких файлах. Результат должен содержать стандартное отклонение в длинах слов в файле и соответствующее название файла. Сохранить результат в файл в виде:

(7@file1 6@file1 13@file2 22@file2 ...)

3. Реализовать подсчет среднеквадратического отклонения случайно заданной выборки длины N. Входной csv-файл должен содержать N строк выборки. Результатом является полученное значение.

4. Подсчитать количество появлений каждого буквенного символа в файле. Подсчет должен быть регистро-зависимым (т.е. буквы «а» и «A» считаются разными). Результат должен быть сохранен в файле в виде:

((a 484) (b 95) (c 187) ...)

5. Модифицировать программу подсчета слов WordCount. Результат должен содержать 100 самых часто встречающихся слов. Из результата должны быть удалены стоп-слова.

6. Реализовать поиск слова в нескольких файлах. Результат должен содержать номера всех строк в каждом файле, в которых появляется заданное слово. Сохранить результат в файл в виде:

(word (7@file1 46@file1 52@file2 63@file2 ...))

7. Подсчитать средний рейтинг фильма. Входной файл имеет формат:

userId, movieId, rating, timestamp.

Результат должен быть сохранен в файле в формате:

moiveId, av\_rating

Входной файл: rating.csv

8. Сгенерировать массив случайных чисел размера N. N> 100000. Разбить массив на ранжированный ряд, используя полуэмпирическую формулу. Для ранжированного ряда найти выборочное среднее. Входными данными является N. Результатом является полученное значение.

9. Построить индекс файла. Для каждого слова в файле результат должен содержать номера всех строк, в которых появляется данное слово. Индекс должен быть регистро-независимым. Результат должен быть сохранен в файле в виде:

((word1 (1 42 58)), (word2 (34 55 776 3456), …)

10. Модифицировать программу подсчета слов WordCount для удаления стоп-слов, знаков пунктуации и цифр. Список стоп-слов должен находиться в отдельном файле.

11. Реализовать подсчет выборочной дисперсии случайно заданной выборки длины N. Входной csv-файл должен содержать N строк выборки. Результатом является полученное значение.

12. Построить обратный индекс для файлов. Обратный индекс для каждого слова содержит список имен файлов, в которых оно встречается, и количество появлений слова в каждом файле. Результат должен быть сохранен в файле в виде:

(word1 (file1 42), (file2 25)), (word2 (file1 55)), …)

13. Реализовать умножение матриц.

Входной файл имеет формат:

имя\_матрицы, строка, столбец, значение.

Результат выполнения стадии Map представить в виде пар ключ-значение, где ключ – индексы элемента вычисляемой матрицы, а значение – список значений, необходимых для вычисления данного элемента.

14. Модифицировать программу подсчета слов WordCount для подсчета слов, начинающихся с заданной подстроки. Из результата должны быть удалены стоп-слова.

15. Подсчитать среднюю стоимость показа рекламы по городам России. Входной файл имеет формат:

userId, country, city, campaign\_id, creative\_id, payment

Результат должен быть сохранен в файле в формате:

city, av\_av\_payment

Входной файл: [payment.csv](https://github.com/k1noX/bmstu-7th-term/blob/master/hadoop/lab2/data/payment.csv)

16. Используя формулу Бэйли-Боруэйна-Плаффа, вычислите определенное количество знаков числа π. Результатом является полученное значение.

17. Подсчитать среднюю длину слов в нескольких файлах. Результат должен содержать средний размер слов в файле и соответствующее название файла. Сохранить результат в файл в виде:

(7@file1 6@file1 13@file2 22@file2 ...)

18. Подсчитать медианную длину слов в нескольких файлах. Результат должен содержать медианный размер слов в файле и соответствующее название файла. Сохранить результат в файл в виде:

(7@file1 6@file1 13@file2 22@file2 ...)

19. Реализовать нахождение обратной матрицы.

Входной файл имеет формат:

имя\_матрицы, строка, столбец, значение.

Результат выполнения стадии Map представить в виде пар ключ-значение, где ключ – индексы элемента вычисляемой матрицы, а значение – список значений, необходимых для вычисления данного элемента.

20. Реализовать подсчет выборочного среднего случайно заданной выборки длины N. Входной csv-файл должен содержать N строк выборки. Результатом является полученное значение.

**КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ**

1. Дайте определение MapReduce. Опишите её назначение.
2. Опишите преимущества MapReduce перед другими моделями.
3. Перечислите основные стадии решения MapReduce задачи.
4. Опишите стадию Map.
5. Опишите стадию Shuffle.
6. Опишите стадию Reduce.
7. Опишите назначение операции Combine.
8. Перечислите области применения MapReduce модели.
9. Перечислите виды MapReduce задач.
10. Опишите группу MapOnly задач и их назначение.
11. Опишите цепочку MapReduce задач.
12. Опишите группу ReduceJoin задач и их назначение.
13. Перечислите особенности модели MapReduce.
14. Опишите структуру входных и выходных данных классической MapReduce задачи.
15. Приведите пример запуска MapReduce задачи.
16. Опишите способы создания в Java-приложениях mapper- и reducer-классов.

**ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ**

На выполнение лабораторной работы отводится 3 занятия (6 академических часов: 5 часов на выполнение и сдачу лабораторной работы и 1 час на подготовку отчета).

Номер варианта студенту выдается преподавателем.

Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы (со скриншотами), результаты выполнения работы. выводы.