МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование кафедры)
02.03.03 Математическое обеспечением и администрирование
информационных систем
(код и наименование направления подготовки, специальности)
Технология программирование
(направленность (профиль)/специализация)

БАКАЛАВРСКАЯ РАБОТА

на тему «Разработка алгоритма построения инвертированного индекса на основе технологии MapReduce»

Студент А.П. Токарев
(И.О. Фамилия) (личная подпись)
Руководитель А.В. Очеповский
(И.О. Фамилия) (личная подпись)
Консультант А.В. Москалюк
(И.О. Фамилия) (личная подпись)

Допустить к защите

Заведующий кафедрой к.т.н., доцент кафедры ПМИ, А.В. Очеповский
(ученая степень, звание, И.О. Фамилия)

«_____»_______________________20_____г. _________________
(личная подпись)

Тольятти 2018

Аннотация

Тема бакалаврской работы «Разработка алгоритма построения инвертированного индекса на основе технологии MapReduce».

Объектом исследования является технология MapReduce.

Предметом исследования является алгоритм построения инвертированного индекса на основе технологии MapReduce.

Бакалаврская работа посвящена разработке алгоритма построения инвертированного индекса на основе технологии MapReduce. В работе представлено сравнение MapReduce с альтернативными технологиями обработки больших данных и выявлено для каких задач данная парадигма подходит лучше всего.

Цель бакалаврской работы - разработка алгоритма построения инвертированного индекса на основе технологии MapReduce.

Для достижения поставленной цели были выполнены следующие задачи: рассмотрены принципы работы различных алгоритмов построения инвертированного индекса, а также принцип работы парадигмы Mapreduce, произведено сравнение существующих технологий для работы с большими данными, проанализированы результаты сравнения и выбран наиболее эффективный способ реализации программы, проектирование программы, разработка программы.

Актуальность работы заключается в том, что с развитием интернета количество данных увеличивается с каждым днем, а распределенные алгоритмы способны выполнять их обработку быстро и эффективно.

Данная работа состоит из 48 страниц, включая в себя введение, три главы, заключение, список используемой литературы из 28 источников, 24 рисунка, 6 таблиц и 4 приложения.

ABSTRACT

The topic of the graduation work is Development of algorithms based on MapReduce technology.

The object of research is MapReduce technology.

The subject of this work is the algorithm for constructing an inverted index based on the MapReduce technology.

Bachelor's work is devoted to the development of algorithms based on MapReduce technology. The paper presents a comparison of MapReduce with alternative technologies for processing large data and identifies which tasks this paradigm implements more effective.

The purpose of the bachelor's work is development an algorithm for constructing an inverted index based on MapReduce technology.

The following tasks were fulfilled: the principles of operation of various algorithms for constructing an inverted index, as well as the principle of the Mapreduce paradigm, were compared, existing technologies were compared to work with large data, the comparison results were analyzed and the most effective method of program implementation, program design, development programs.

The urgency of the work is that with the development of the Internet, the amount of data increases every day, and distributed algorithms can perform their processing quickly and efficiently.

This work consists of 48 pages, including the introduction, three chapters, conclusion, a list of 28 references, 24 figures, 6 tables and 3 applications.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ
1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ ЗАДАЧИ ПОСТРОЕНИЯ ИНВЕРТИРОВАННОГО ИНДЕКСА..6
1.1 Инвертированный индекс в поисковых системах 6
1.2.1 Алгоритмы построения инвертированного индекса...........................7
1.2.2 Инвертированный индекс в MapReduce 10
1.3 Парадигма MapReduce... 12
1.4 Распределенная файловая система HDFS... 20
2. АНАЛИЗ ТЕХНОЛОГИЙ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ 24
2.1 Альтернативы технологии MapReduce... 24
2.2 Сравнительный анализ технологий для обработки больших данных . 29
2.3 Выбор технологии для реализации алгоритма 31
3. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА АЛГОРИТМА ПОСТРОЕНИЯ ИНВЕРТИРОВАННОГО ИНДЕКСА НА МОДЕЛИ MAPREDUCE .... **Ошибка! Закладка не определена.**
3.1 Требования к разрабатываемой программе....... **Ошибка! Закладка не определена.**
3.2 Архитектура разрабатываемой программы .... **Ошибка! Закладка не определена.**
3.3 Разработка программы построения инвертированного индекса 37
3.4 Компиляция и запуск программы построения инвертированного индекса в системе Hadoop.. 40
3.5 Сравнение с последовательной программой 42
ЗАКЛЮЧЕНИЕ.. 44
СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ... 46
ПРИЛОЖЕНИЕ_А. Листинг InvertedIndexDriver 49

ВВЕДЕНИЕ

С появлением вычислительной техники объем информации, с которой может работать человек, значительно увеличился. Еще более 30 лет назад учеными стали разрабатываться алгоритмы, позволяющие упростить работу с данными, а также получить из них новые, ранее неизвестные знания. Однако сегодня эти алгоритмы неэффективны, поскольку объемы информации стали слишком большими.

Для работы с большими объемами данных была разработана специальная система Apache Hadoop [1], Главное место в этой системе занимает технология под названием MapReduce. Реализация MapReduce позволяет производить распределенные вычисления над большими объемами данных в компьютерных кластерах эффективно и безотказно.

Системы, основанные на MapReduce, разработаны таким образом, чтобы параллельность вычислений была реализована не за счет суперкомпьютера, а за счет вычислительных кластеров - наборов стандартных аппаратных средств, соединенных между собой.

Актуальность бакалаврской работы обусловлена тем, что на сегодняшний день объемы хранимых данных становятся слишком большими для того, чтобы была возможность обрабатывать их традиционными алгоритмами. Технология MapReduce позволяет производить эффективные распределенные вычисления за оптимальное время выполнения.

К тому же, технология MapReduce быстро развивается и распространяется. Многие компании используют ее для обработки данных, потому что она имеет открытый исходный код, масштабируема и не требует больших затрат на оборудование.

Новизна исследования состоит в том, что многие проблемы, возникающие, при обработке больших данных могут быть решены за счет высокой отказоустойчивости, масштабируемости и доступности технологии MapReduce.

Целью бакалаврской работы является разработка алгоритма построения инвертированного индекса на основе технологии MapReduce.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Рассмотреть принципы работы различных алгоритмов построения инвертированного индекса, а также принцип работы парадигмы Mapreduce.
2. Сравнить существующие технологии для работы с большими данными.
3. Проанализировать результаты сравнения и выбрать наиболее эффективный способ реализации программы.
4. Проектирование программы.
5. Разработка программы

В первой главе рассматриваются алгоритмы построения инвертированного индекса и принцип работы технологии Mapreduce.

Во второй главе произведено сравнение и анализ технологий для работы с большими данными.

В третьей главе представлены проектирование и разработка алгоритма построения инвертированного индекса на основе модели MapReduce.

В заключении сформированы выводы, полученные в процессе выполнения бакалаврской работы.

1.ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ ЗАДАЧИ ПОСТРОЕНИЯ ИНВЕРТИРОВАННОГО ИНДЕКСА.

1.1 Инвертированный индекс в поисковых системах

Веб-поиск - это одна из главных проблем большого объема данных. Учитывая потребность в информации, выраженную в виде короткого запроса, состоящего из нескольких слов, задача системы состоит в том, чтобы извлекать соответствующие веб-объекты (веб-страницы, документы PDF, слайды PowerPoint и т. Д.) и представлять их пользователю. Насколько велика сеть? Точно неизвестно, но даже по самым заниженным оценкам ее размер составляет не менее нескольких десятков миллиардов страниц, размер которых составляет сотни терабайт, учитывая только текст [4]. В реальных приложениях пользователи запрашивают результаты из поисковой системы и ожидают быстрого результата выполнения их запроса. На сегодняшний день задержки даже в несколько сотен миллисекунд будут испытывать терпение пользователя. Выполнение этих требований происходит путем построения инвертированного индекса.

Почти все поисковые механизмы для полнотекстового поиска сегодня полагаются на структуру данных, называемую инвертированным индексом, которая представлена в виде термина и списка документов, содержащих этот термин [10]. Извлекаемые во время поиска объекты называют документами, хотя на самом деле они могут являться web-страницами, PDF файлами или даже фрагментами кода. Учитывая пользовательский запрос, механизм поиска использует инвертированный индекс для нахождения документов, которые содержат условия запроса.

Информационно-поисковая система включает в себя два этапа: индексирование и поиск. Первым шагом этапа индексации является сбор коллекции документов. Эта коллекция может быть получена несколькими способами, например, путем сканирования страниц в Интернете или путем ручного сбора. После этого, собранная информация индексируется. Индексация информации обеспечивает быстрый поиск. Инвертированный индекс является

наиболее часто используемой структурой для индексирования текстовых данных [4].

Когда информация проиндексирована, она готовы к поиску. Поиск начинается с пользователя, который отправляет запрос, представленный в виде нескольких терминов. Поисковая система обрабатывает запрос и извлекает соответствующие ему документы, после чего предоставляет их пользователю.

Инвертированный индекс занимает самую важную роль в системах, осуществляющих поиск в больших наборах данных. Без индексации документов, системе поиска пришлось поочередно сканировать каждый документ, на что потребовалось бы значительное количество времени и ресурсов.

1.2.1 Алгоритмы построения инвертированного индекса

Инвертированный индекс - структура данных, в которой для каждого слова коллекции документов в соответствующем списке перечислены все документы в коллекции, в которых оно встретилось.

Word	Documents
word_1	Document_2, Document_3, Document 4
word_2	Document_1, Document_3
word_3	Document_4
word_4	Document_1, Document_3, Document 4

Рисунок 1.1 – Инвертированный индекс.

Этот индекс определяет только наличие слова в конкретном документе, то есть такой индекс определяет, какие документы соответствуют запросу, но не определяет степень их соответствия. В некоторых проектах инвертированный индекс содержит дополнительную информацию, такую как частота каждого слова в документе или его позиция. Данная информация позволяет алгоритму поиска определять наиболее подходящие запросу документы.

Задачу построения инвертированного индекса можно разделить на следующие этапы:

- Сбор документов для индексации.
- Разбор и токенизация текста.
- Предварительная обработка терминов
- Присваивание каждому термину документов, в которых он содержится.

Токенизация – это процесс разделения слов текста, обычно с помощью пробела или знаков пунктуации.

К обработке текста можно отнести удаление стоп-слов. Стоп-слова – это очень распространенные слова, которые появляются в тексте, но играют второстепенную роль в поиске. Удаление стоп-слов из процесса индексирования позволяет уменьшить размер конечного индекса.

Нормализация – это процесс приведения слова к начальной форме. Обычно в тексте есть одинаковые слова, представленные в разных формах, но их требуется считать за одно слово. В качестве хорошего примера служит запись U.S.A. и USA. Существует два основных подхода к решению этой проблемы. Наиболее часто используемый подход заключается в создании классов эквивалентности, которые приводят слова к общей форме. В данном примере слова U.S.A. и USA будут преобразованы в USA. Второй подход заключается в поддержании отношений между ненормализованными словами. Мы можем расширить этот подход, добавив список синонимов, таких как автомобиль и машина. Этот метод менее эффективен, чем классификация эквивалентности.

Индексирование на основе сортировки блоков (BSBI) – это алгоритм, который хранит словарь терминов в памяти, а документы на диске. Словарь должен помещаться в памяти, а свободное место на диске по крайней мере быть вдвое больше, чем коллекция документов. По соображениям эффективности алгоритм преобразует термин в идентификатор. Обычно это делается при обработке коллекции. Алгоритм проходит через коллекцию, создавая пары

<termID, dID> в памяти, отсортированные сначала с помощью termID, а затем с помощью docID. Если алгоритм выходит за пределы памяти, то он перемещает весь блок отсортированных пар на диск и продолжает создавать следующий блок. Когда все пары извлечены, отсортированы и сохранены в отдельных блоках на диске, алгоритм использует внешний алгоритм сортировки. Из каждого блока считывается часть пар, и у пар с одинаковым идентификатором termID, поле docID объединяется в список документов. Когда для каждого термина собран список документов, записывается итоговый инвертированный индекс.

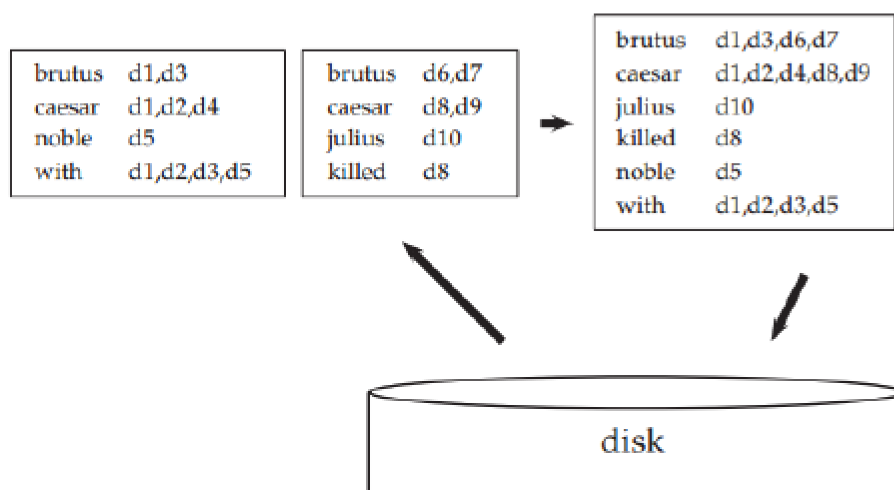

Рисунок 1.2 – Построение инвертированного индекса [26].

Временная сложность алгоритма - $O(T \log T)$, где T - число пар, подлежащих сортировке, потому что сортировка - это шаг наивысшей сложности. Но обычно, большую часть времени выполнения занимают операции, которые работают с диском - чтение пар из блоков и итоговое слияние[26].

Когда словарь терминов не помещается в память, используется другой подход, называемый однопроходной индексацией в памяти (SPIMI). Вместо termID используются термины, и для каждого блока создается собственный словарь, записанный на диске.

Алгоритм поочередно обрабатывает пары <term,docID>, термин добавляется в словарь блока, если он еще не встречался, и создается новый список документов. Если этот термин уже встречался, то соответствующий docID добавляется в конец списка документов. Это возможно, потому что docID обрабатываются в порядке возрастания. Это также намного быстрее из-за отсутствия сортировки.

Трудно предсказать окончательную длину списка документов. Поэтому алгоритм сначала выделяет пространство для короткого списка документов, а затем удваивает его каждый раз, когда список заполняется. Когда память заполнена, алгоритм сортирует термины и записывает весь блок со словарем на диск. В конечном итоге, все блоки объединяются путем однократного последовательного перебора каждого блока.

Еще одно преимущество SPIMI - сжатие каждого документа. Это делает эффективность еще более высокой, потому что позволяет поместить в память более крупные блоки данных, а также экономить дисковое пространство. Поскольку необходимость в сортировки отсутствует, то временная сложность алгоритма SPIMI равна O (T) [26].

1.2.2 Инвертированный индекс в MapReduce

Предыдущие алгоритмы предполагают использование относительно небольших коллекций документов. Для крупномасштабных коллекций, т. е. более чем миллиардной веб-коллекции, построение инверсированного индекса на одиночной машине невозможно [8]. Для выполнения этой задачи необходимы большие компьютерные кластеры. В существующих механизмах веб-поиска используются распределенные алгоритмы индексирования, которые строят распределенные индексы. Индекс разделяется на несколько компьютеров в соответствии с термином, либо с документом. Такой алгоритм строится на модели распределенных вычислений MapReduce. MapReduce работает на больших компьютерных кластерах, где каждый узел является

обычным компьютером. Эти узлы управляются главным узлом, который назначает задачи рабочим узлам. MapReduce имеет две фазы:

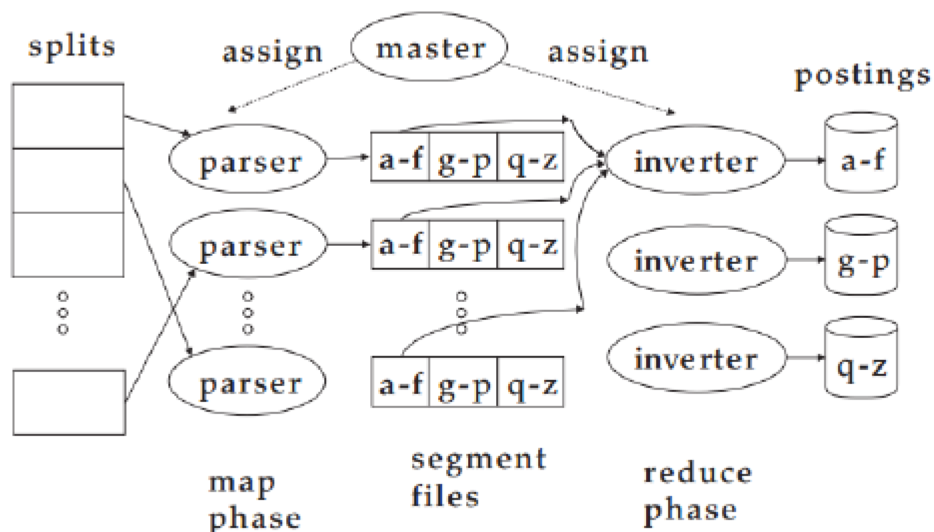

Рисунок 1.3 – Инвертированный индекс в MapReduce [26].

Map.На этом этапе главный узел сначала разбивает коллекцию документов на одинаковые по размеру части, которые в дальнейшем будут равномерно распределены между узлами. Затем главный узел назначает элементы рабочим узлам, которые называются парсерами или синтаксическими анализаторами. Процесс синтаксического анализа очень похож на процесс, выполняемый в нераспределенных алгоритмах. После обработки каждый парсер записывает результат (пары ключ-значение) на локальный диск рабочего узла.

Reduce. На данном этапе происходит сортировка и агрегация пар ключ-значение по ключу и передача их на инверторный рабочий узел. Инверторы собирают и сортируют все docID для данного termID, и в результате получают данные в виде <ключ, список документов>.

Был рассмотрен ряд алгоритмов построения инвертированного индекса. Для реализации был выбран распределенный алгоритм, построенный на модели MapReduce, так как он единственный из рассмотренных алгоритмов способен работать с большими коллекциями данных.

Для реализации алгоритма построения инвертированного индекса, основанного на модели MapReduce, необходимо детально разобрать принцип работы этой парадигмы.

1.3 Парадигма MapReduce.

MapReduce - это модель распределенных вычислений, представленная компанией Google в 2004 году для сканирования и обработки множества страниц из сети Интернет. Первая реализация этой модели была выполнена на основе распределенной файловой системы GFS (Google File System). Эта реализация запатентована и активно используется компанией, но использование ее вне Google недоступно.

Альтернативная, свободно доступная реализация Hadoop MapReduce была выполнена в проекте Hadoop сообщества Apache. Эта реализация основывается на использовании распределенной файловой системы HDFS (Hadoop Distributed File System), также разработанной в проекте Hadoop. Реальную популярность MapReduce принесла именно реализация Hadoop в силу своей доступности и открытости, а широкое использование Hadoop MapReduce в различных исследовательских проектах приносит несомненную пользу этой системе, стимулируя разработчиков к ее постоянному совершенствованию [1].

Реализация MapReduce способна выполнять множество распределенных вычислений на сотнях или даже тысячах связанных между собой компьютеров таким образом, что сложность распараллеливания, обработка ошибок и аппаратные неисправности не являются проблемой [2]. Все, что нужно разработчику - это написать две функции, которые называются Map и Reduce, в то время как система самостоятельно управляет распределенными вычислениями, координацией выполняющихся задач, а также контролирует ситуацию, если одна из этих задач не выполняется. Схема работы MapReduce изображена на рис. 1.4.

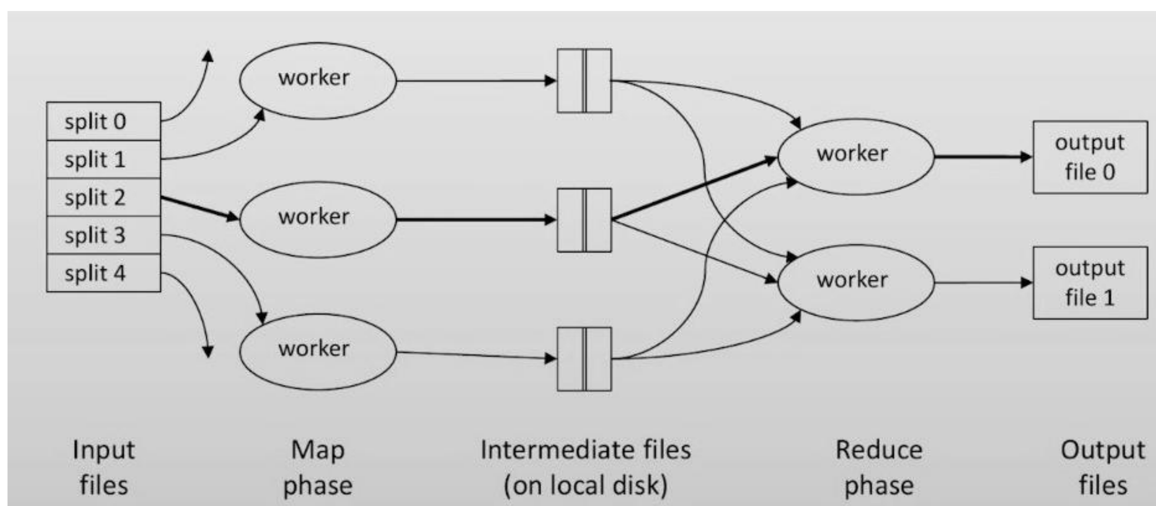

Рисунок 1.4 — Схема работы MapReduce.

Алгоритм выполнения MapReduce:

1. Часть данных поступает на один из вычислительных узлов и обрабатывается функцией Map. Выводом этой программы с одного кластера являются пары ключ-значение.

2. Все пары ключ-значение сортируются по ключу и передаются функции Reduce.

3. Функция Reduce обрабатывает данные, упорядоченные по ключу. Способ обработки определяет код, написанный пользователем для функции Reduce.

Это факт, что рано или поздно аппаратные средства выходят из строя, и чем больше вычислительных узлов и соединений имеет система, тем больше вероятность того, что произойдет сбой и остановка работы системы.

Некоторые вычисления могут занять несколько минут или даже часов в тысячах вычислительных узлов. Если бы приходилось прерывать и перезапускать вычисления каждый раз, когда один компонент выходит из строя, то система бы работала не эффективно.

В MapReduce эта проблема решается следующими способами:

1. Файлы имеют резервные копии. Если не дублировать файл на нескольких вычислительных узлах, то при выходе из строя узла, все его файлы станут недоступны, до тех пор, пока он не будет отремонтирован. А если произойдет сбой диска, то файлы будут потеряны навсегда.

2. Вычисления разделены на задачи таким образом, что, если одна задача не выполняется до конца, то она перезапускается, не затрагивая другие задачи [7].

Для того, чтобы использовать кластерные вычисления, файлы должны выглядеть несколько иначе, чем в обычных файловых системах. Для этого используется так называемая распределенная файловая система (Distributed File System). Она функционирует следующим образом:

Файлы разделены на части, как правило, размером по 128 МБ. Эти части реплицируются три раза в три различных вычислительных узла. Кроме того, каждая часть должна быть расположена на разных стойках, чтобы не произошло потери всех копий из-за выхода из строя всей стойки. Как правило, такие параметры, как размер блока данных и степень репликации задаются пользователем. Для того, чтобы найти части файла, существует специальный небольшой метафайл, которым управляет главный узел.

Входные данные для задачи Map состоят из независимых между собой элементов, которые могут быть любого вида: массив, документ, и тому подобное. Часть данных может представлять собой совокупность элементов. Все выходые данные функции Map и выходные данные функции Reduce представлены в виде пар ключ-значение. Функция Map принимает входной элемент в качестве аргумента и производит ноль или более пар ключ-значение. Типы ключей и значений каждый узел генерирует произвольно. Кроме того, ключи не должны быть уникальными. Чаще всего одна задача функции Map может произвести несколько пар ключ-значение с одинаковым ключом.

Как только задача Map успешно завершится, пары ключ-значение группируются по ключу, а значения каждого ключа формируются в список значений. Группировка осуществляется внутри системы, и не задается разработчиком. Разработчик обычно указывает системе количество Reduce задач (обозначим r). Во время выполнения главный контроллер выбирает хэш-функцию, которая применяется к ключам и производит ряд значений от 0 до r - 1. Каждый ключ из задачи Map хешируется и его пары ключ-значение

записываются в один из r локальных файлов. Каждый файл предназначен для одной из Reduce задач. Чтобы выполнить группировку по ключу и распределить файлы по Reduce задачам, главный контроллер объединяет файлы, предназначенные для конкретной Reduce задачи и передает объединенный файл в этот процесс в виде последовательности пар ключ-списком значений.

Аргументом функции Reduce является ключ и список, соответствующих ему значений. Выходом функции Reduce является последовательность из нуля или более пар ключ-значение. Эти пары могут быть иного типа, нежели при выходе из Map задачи, но обычно их типы совпадают. Задача Reduce получает один или несколько ключей и связанных с ними списков значений. То есть, Reduce выполняет одно или несколько заданий. Выходы из всех Reduce задач объединяются в один файл. Иногда функция Reduce обладает свойствами ассоциативности и коммутативности. То есть, значения могут быть объединены в любом порядке, а результат останется неизменным.

На рисунке 1.5 изображена подробная схема, отображающая взаимодействие между файлами, процессами и задачами во время выполнения MapReduce программы. При помощи библиотек, представленных системой Hadoop MapReduce, главный узел производит разветвление процессов на различных вычислительных узлах. Как правило, на узлах обрабатывается задача Map, или Reduce, но не обе одновременно [14].

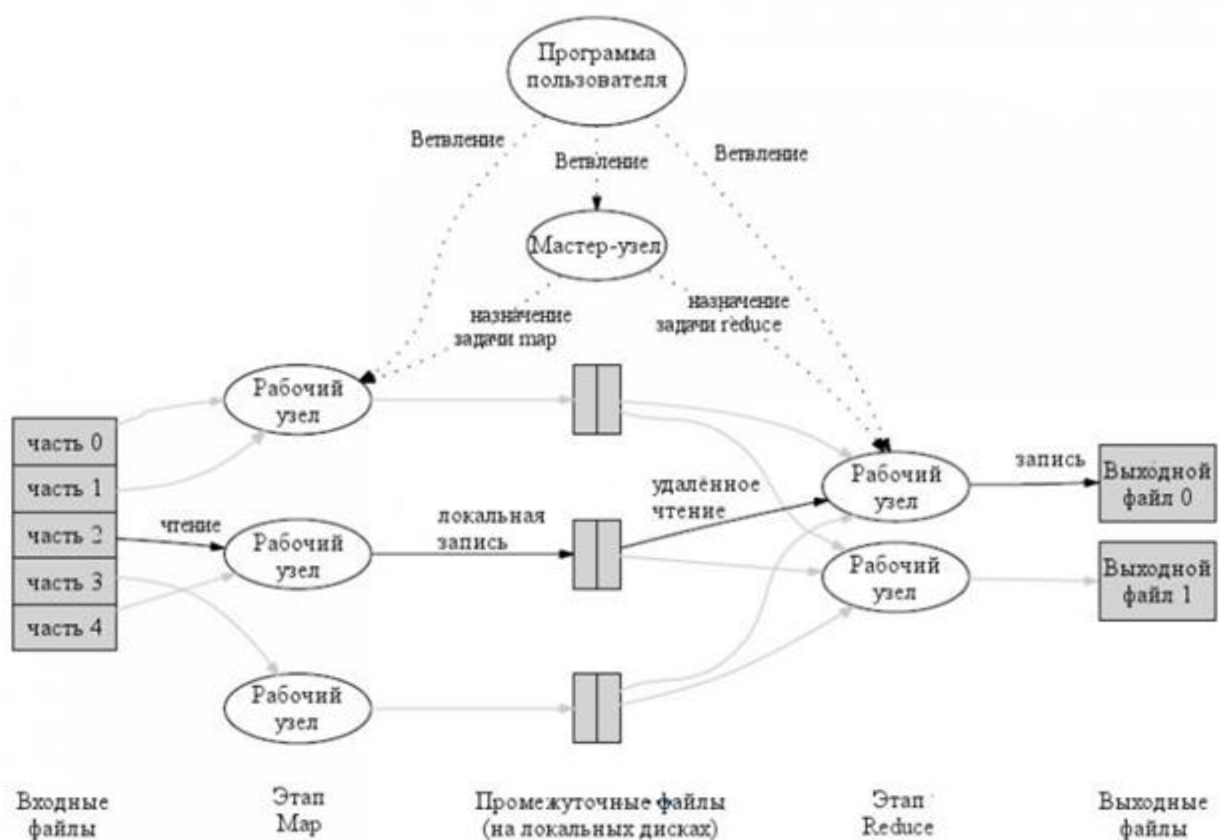

Рисунок 1.5 – Детальный обзор работы MapReduce программы.

Главный узел имеет много обязанностей. Одной из них является создание определенного количества Map и Reduce задач. Их количество задается разработчиком в программе. Эти задачи будут назначены на рабочие процессы с помощью главного узла. Разумно создавать по одной Map задачи для каждого фрагмента входного файла, тогда потребуется меньше Reduce задач [13]. Причиной ограничения является необходимость создания промежуточных файлов для каждой Map задачи. Если Map задач будет слишком много, то число промежуточных файлов будет значительно больше числа Reduce задач. Поэтому перед тем, как задавать количество Map задач необходимо проанализировать возможности Reduce задач.

Главный процесс отслеживает состояние каждой Map и Reduce задачи. Когда рабочий процесс завершает выполнение задачи, главный процесс назначает ему новую.

Каждой Map задаче задается одна или несколько частей входного файла и после этого она выполняет код, написанный пользователем. Map задача создает

файл для каждой Reduce задачи на локальном диске узла, который выполняет задание. Главному процессу сообщается о месте и размере каждого из этих файлов. Когда Reduce задача назначается мастером в рабочий процесс, то ей передаются все необходимые файлы, формулирующие ее вход. Reduce задача выполняет код, написанный разработчиком, и записывает свой результат в файл, который является частью распределенной файловой системы.

Худшее, что может произойти - вычислительный узел, на котором работает главный процесс, перестанет работать. В этом случае вся MapReduce программа должна быть перезапущена. Это единственный узел, который может нанести ущерб всему процессу; другие отказы будут управляться главным узлом, и работа MapReduce будет завершена без больших задержек.

Предположим, что вычислительный узел, на котором выполняется задача Map перестает работать. Эта остановка будет обнаружена главным узлом, потому что он периодически проверяет состояние рабочих процессов. Все задания Map, которые были назначены на этот процесс должны будут быть перезапущены, даже если они были завершены. Причиной переработки завершенных Map задач является то, что их результат сохранен именно на этом узле. Главный процесс устанавливает статус каждой из этих задач Map, как готовой к исполнению на другом узле.

Случай с отказом в Reduce процессе обрабатывать проще. Главный процесс просто прерывает выполнение на этом узле и назначает выполнение на другом, как только тот становится доступным [16].

Рассмотрим технологию MapReduce на простом примере. Предположим, что имеется пять файлов, и каждый файл содержит два столбца (ключ и значение с точки зрения Hadoop), которые представляют город и соответствующую температуру, записанную в этом городе в разные дни измерений.

Таблица 1.1 — Исходные данные.

Город	Температура
Торонто	20
Уитби	25
Нью-Йорк	22
Рим	32
Торонто	4
Рим	33
Нью-Йорк	18

Из всех собранных данных мы хотим найти максимальную температуру для каждого города (стоит обратить внимание, что каждый файл может иметь один и тот же город, представленный несколько раз). Используя механизмы MapReduce, мы можем разбить вычисления на пять Map задач, каждая из которых будет работать с одним из пяти файлов. Map задачи обрабатывают файл и возвращают максимальную температуру для каждого города. Например, результаты, полученные с одной Map задачи для приведенных выше данных, будут выглядеть, как показано в таблице 1.2.

Таблица 1.2 — Результат работы одной Map задачи.

Город	Температура
Торонто	20
Уитби	25
Нью-Йорк	22
Рим	33

Предположим, что остальные четыре Map задачи (которые работают с четырьмя оставшимися файлами) дали следующие промежуточные результаты.

Таблица 1.3 - Промежуточные результаты работы над другими файлами.

Номер файла	**Город**	**Температура**
1	Торонто	18
1	Уитби	27
1	Нью-Йорк	32
1	Рим	37
2	Торонто	32
2	Уитби	20
2	Нью-Йорк	33
2	Рим	38
3	Торонто	22
3	Уитби	19
3	Нью-Йорк	20
3	Рим	31
4	Торонто	31
4	Уитби	22
4	Нью-Йорк	19
4	Рим	30

Все пять из этих выходных потоков будут передаваться в Reduce функции, которые объединяют входные результаты и выводят одно значение для каждого города, выдавая окончательный результат, как показано в таблице 1.4.

Таблица 1.4 — Окончательный результат.

Город	**Температура**
Торонто	32
Уитби	27
Нью-Йорк	33
Рим	38

В данном пункте был представлен общий обзор технологии MapReduce и пример ее использования. Теперь необходимо рассмотреть принцип работы распределенной файловой системы HDFS, так как все MapReduce программы выполняются поверх нее.

1.4 Распределенна файловая система HDFS.

HDFS (Hadoop File System) — распределенная файловая система с открытым исходным кодом, разработанная сообществом Apache. HDFS основана на распределенной файловой системе компании Google (GFS) [25].

Основная идея распределенной файловой системы состоит в том, чтобы разделить пользовательские данные на блоки и реплицировать эти блоки по локальным дискам узлов в кластере. В распределенной файловой системе используется архитектура ведущий-ведомый, в которой главный узел управляет всеми остальными процессами и файлами (метаданные, структуры каталогов, расположение блоков и разрешения доступа), а подчиненные узлы - управляют блоками данных. В Hadoop главный узел называют namenode, а ведомые datanode. Архитектура HDFS представлена на рисунке 1.6.

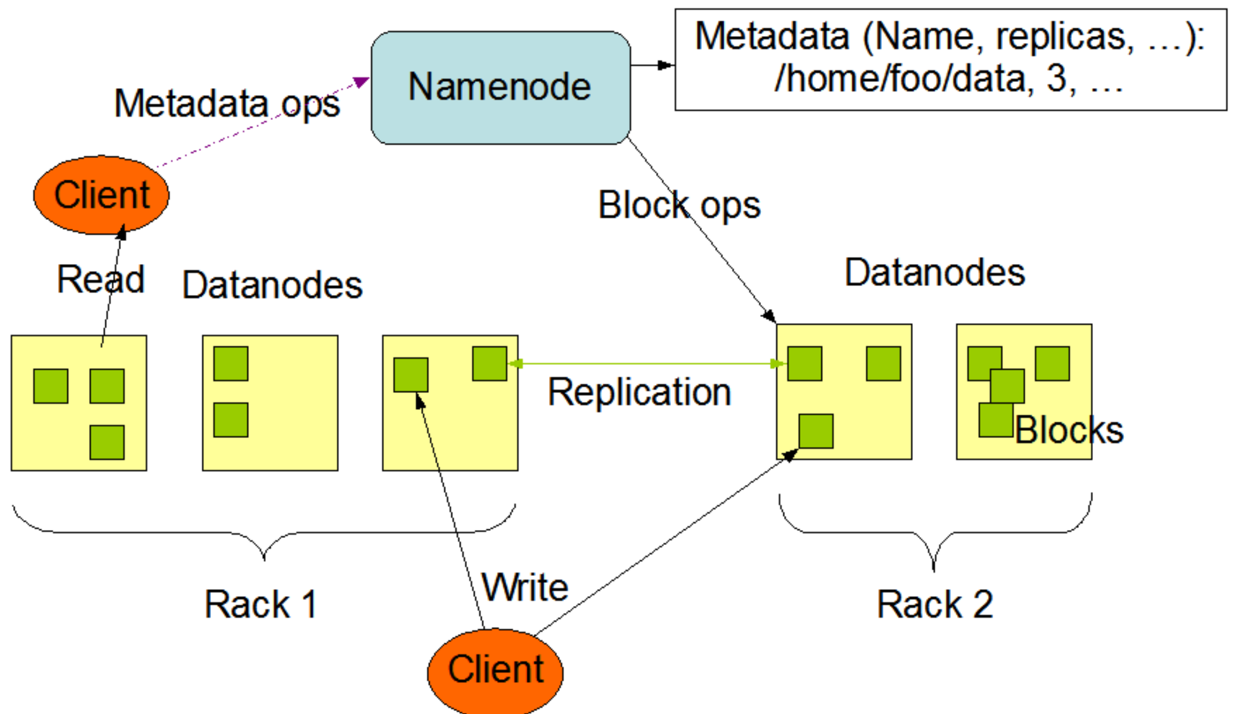

Рисунок 1.6 — Структура Hadoop Distributed File System [1].

В HDFS клиент приложения, желающий прочитать файл (или его часть), должен сначала связаться с namenode, чтобы определить, где этот файл хранится. В ответ на запрос клиента namenode возвращает идентификатор соответствующего блока данных и адрес узла (datanode), в котором находится блок. Затем клиент связывается с данным datanode для извлечения данных. Сами блоки данных хранятся в стандартных однокомпонентных файловых системах, поэтому HDFS лежит поверх стандартного стека ОС (например, Linux). Важной особенностью конструкции является то, что данные никогда не перемещаются через namenode. Вместо этого вся передача данных происходит непосредственно между клиентами и datanode. Связь с главным узлом происходит только через передачу метаданных [25].

По умолчанию HDFS хранит три отдельные копии каждого блока данных для обеспечения надежности и доступности. В больших кластерах эти три реплики распределены по разным физическим стойкам, поэтому HDFS устойчив к двум главным сценариям сбоя: выход из строя отдельного узла и

сбои в сетевом оборудовании, которые нарушают работу всей стойки. Главный узел периодически связывается с данными, чтобы обеспечивать правильную репликацию всех блоков. Если реплик недостаточно, например, из-за сбоев рабочих узлов или потерь связи из-за сбоев сетевого оборудования, то namenode направляет инструкции на создание дополнительных копий.

Чтобы создать новый файл и записать данные в HDFS, клиент приложения сначала связывается с namenode, который обновляет пространство имен файлов. После проверки разрешений и удостоверения, что файл еще не существует, Namenode выделяет новый блок на подходящем datanode, и записывает на него данные. После записи данных на datenode, данные реплицируются и распространяются на остальные datenode.

Таким образом, namenode HDFS выполняет следующие функции:

• Управление пространством имен файлов. Namenode отвечает за сохранение пространства имен файлов, которые включают в себя метаданные, структуру каталогов, файл для блочного сопоставления, расположение блоков и разрешения доступа. Эти данные хранятся в памяти для быстрого доступа, и все изменения постоянно регистрируются.

• Координация операций с файлами. Namenode направляет клиентов приложений в datanode для чтения данных и выделяет блоки для подходящих datanode для записи данных. Все передачи данных происходят непосредственно между клиентами и datanode.

• Поддержание общего состояния файловой системы. Namenode периодически контактирует с datanode для обеспечения целостности системы. Если namenode отмечает, что блок данных не реплицирован, он создаст новые реплики. Наконец, namenode также отвечает за перераспределение файловой системы. В ходе обычных операций некоторые datanode могут в конечном итоге содержать больше блоков, чем другие. Перераспределение включает в себя перемещение блоков из datanode с большим количеством блоков в datanode с меньшим количеством блоков. Это приводит к лучшей балансировки нагрузок и более равномерному использованию диска.

HDFS была специально разработана для реализации MapReduce, поэтому понимание того, как работает распределенная файловая система имеет решающее значение для разработки эффективных алгоритмов MapReduce.

В HDFS желательно хранить файлы большого размера. Существует несколько причин, почему следует избегать большого количества небольших файлов. Поскольку namenode должен хранить все метаданные файлов в памяти, большие многоблочные файлы представляют собой более эффективное использование памяти namenode, чем много одноблочных файлов. Кроме того, функция map использует отдельные файлы в качестве основного блока для обработки входных данных. В настоящее время в Hadoop нет механизма, который позволяет функции обрабатывать несколько файлов одновременно. В результате, обработка огромного количества очень маленьких файлов приведет к тому, что будет задано слишком много map задач. Это может привести к двум проблемам: во-первых, затраты на запуск map задач могут стать значительными по сравнению со временем, затраченным на обработку входных данных; во-вторых, это может привести к чрезмерному количеству операций копирования по сети во время фазы сортировки [24].

Система построена из ненадежных, но недорогостоящих компонентов. Поэтому ошибки являются нормой, а не исключением. В HDFS заложены функции самоконтроля и самовосстановления, чтобы эффективно справляться с общими режимами отказа.

Главной проблемой HDFS является то, что если главный узел (namenode) выходит из строя, то вся файловая система становится недоступна. Это единственный вариант сбоя всей системы. А так как никакие данные никогда не передаются через namenode и не нагружают его, кроме метаданных, то такие сбои происходят не часто.

2. АНАЛИЗ ТЕХНОЛОГИЙ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ

2.1 Альтернативы технологии MapReduce

Традиционным решением для использования в аналитических системах и крупных хранилищах данных с объемами от сотен гигабайт до сотен терабайт были реляционные СУБД. По сравнению с анализом данных в СУБД с большим количеством дисков-накопителей, MapReduce выигрывает на аппаратном уровне. Причина в особенности работы накопителей на жестких дисках, а именно то, что время поиска данных происходит медленнее, чем скорость их передачи. Процесс поиска характеризуется задержкой дисковых операций, в то время как скорость передачи данных зависит от пропускной способности диска. Если модель доступа к данным преобладает над поиском, то процесс займет больше времени, чтобы считать или записать большие наборы данных, чем время, затраченное на проход через них, которое зависит от скорости передачи данных [21].

Во многих отношениях MapReduce можно рассматривать как дополнение к СУБД. MapReduce хорошо подходит для задач, в которых необходимо проанализировать весь набор данных в пакетном представлении. Предпочтение СУБД следует предоставить при работе с одиночными запросами или обновлениями, в которых набор данных был проиндексирован [24]. С другой стороны, для обновления небольшой части записей в базе данных эффективно используются традиционные B-Tree (структуры данных в реляционных базах данных, которые ограничены по скорости выполнения поиска). Однако при работе с большинством баз данных, B-Tree оказались менее эффективными, чем MapReduce, использующий сортировку - слияние (Sort-Merge) для перестройки базы данных. То есть, MapReduce подходит для приложений, где данные записываются один раз, а считываются многократно, в то время как реляционные базы данных эффективнее для обработки наборов данных, которые постоянно обновляются.

Реляционные данные часто нормализуются для обеспечения своей целостности и удаления избыточности. В MapReduce считывания записей

является нелокальной операцией, что делает использование нормализации проблематичным.

Log веб-сервера является удачным примером большого количества ненормированных записей - это обосновывает удобство анализа log-файлов любого типа с помощью MapReduce. MapReduce является моделью с линейной масштабируемостью. Важно, что для написанных программистом функций map и reduce не важны особенности кластера, на котором осуществляется обработка или размер данных, которые обрабатываются. Они могут оставаться неизменными как для небольшого набора данных, так и для гигантского массива. Особенность в том, что если удвоить размер входных данных, то это приведет к замедлению выполнения задания вдвое (рис.2.1) [11]. В то же время удвоения размера кластера вернет прежнюю скорость выполнения, что не свойственно для SQL запросов (рис.2.2) [11]. Данные графики получены в ходе исследования эффективности технологии MapReduce. Конфигурации кластеров также важны для анализа результатов и приведены ниже.

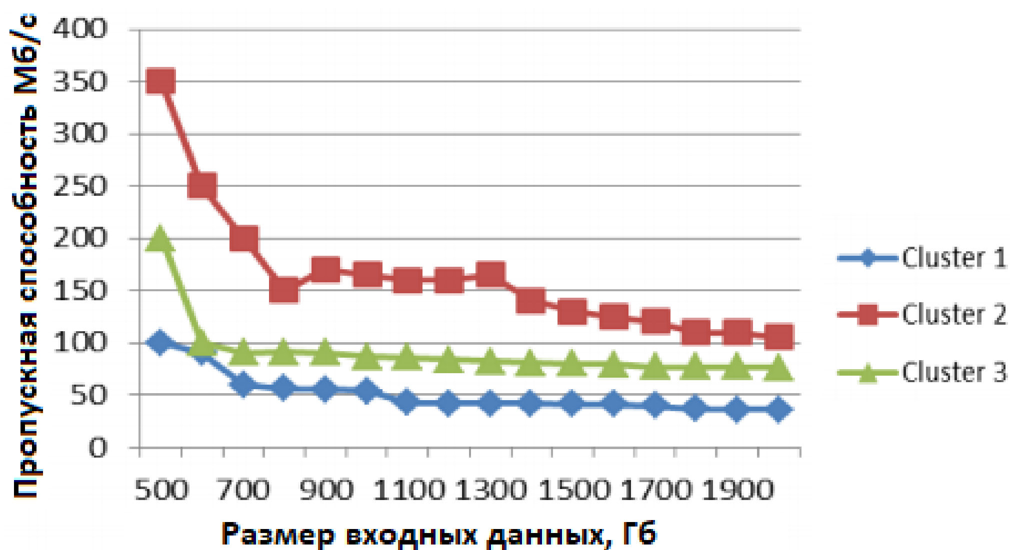

Рисунок 2.1 - График зависимости пропускной способности для кластеров различной конфигурации от размера входных данных.

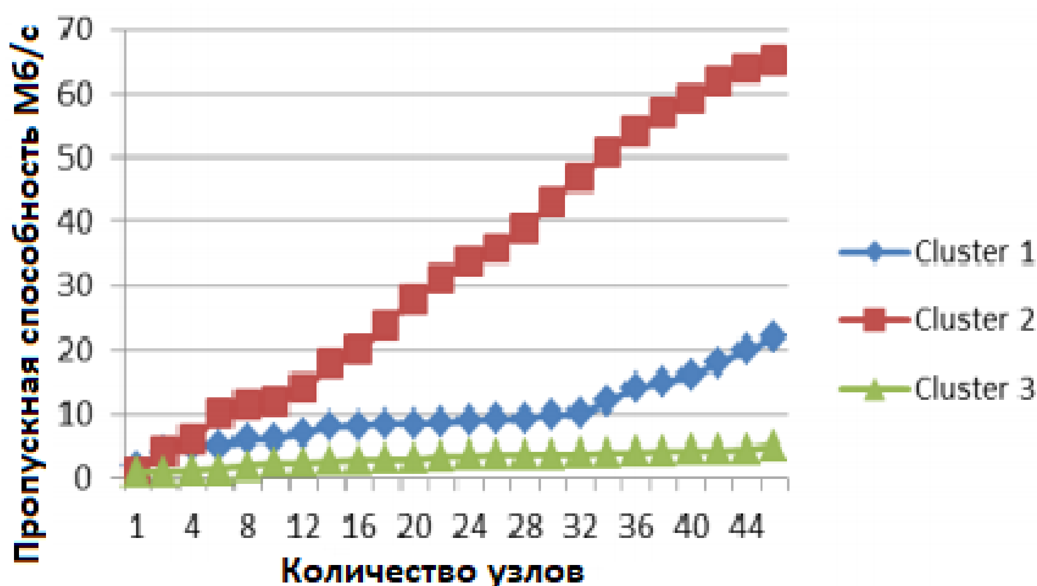

Рисунок 2.2 - График зависимости пропускной способности для кластеров различной конфигурации от количества узлов в кластере.

Однако, различия между реляционными базами данных и MapReduce системами стираются. В реляционных базах данных начали учитывать идеи MapReduce (например, базы данных Greenplum, основанные на доработанной PostgreSQL для базы данных с массивно-параллельной архитектурой). Так и с другой стороны, на основе MapReduce разработали языки запросов высокого уровня (такие, как Pig и Hive) [1].

Высокопроизводительные вычисления (HPC) и Грид-вычисления (Grid Computing) использовались для крупномасштабных обработок данных в течение многих лет, используя такие API, как Message Passing Interface (MPI). Такой подход эффективно работает преимущественно с интенсивными вычислениями, но возникают проблемы, когда узлы должны получить доступ к большим объемам данных (для размеров от сотни гигабайт), так как пропускная способность сети является слабым местом и вычисления на узлах переходит в состояние простоя [20]. MapReduce распределяет данные так, чтобы они находились рядом с узлом, который осуществляет вычисления, что позволяет обеспечить быстрый доступ, поскольку данные является локальными. Все MapReduce реализации строго придерживаются этого свойства, неявно моделируя топологию сети, так как пропускная способность сети является ценным ресурсом в среде дата-центра.

Использование MPI позволяет программисту контролировать процесс, но взамен требует от него явной обработки механики потока данных доступных через сокеты и низкоуровневые подпрограммы C. MapReduce работает только на высоком уровне [28]: программист задает функции обработки данных, но поток данных не является явным. При использовании MPI проблемой является координация процессов для крупномасштабных распределенных вычислений, особенно, когда речь идет об обработке частичного отказа. При таком отказе должно продолжаться выполнение общего вычисления. Парадигма MapReduce неявно обрабатывает невыполненный процесс и перенаправляет выполнение на рабочие машины. При использовании же MPI, основной контроль реализует программист, что усложняет написание MPI программ.

В то же время MapReduce проигрывает MPI программам, когда речь идет о итеративных вычислениях [11]. Для анализа производительности MapReduce и MPI программ в контексте этого эксперимента рассматриваются следующие задачи: кластеризация с помощью K-means алгоритма и умножения матриц. Перемножение матриц иллюстрирует неэффективность осуществления итеративных вычислений MapReduce программами (рис.2.3) [11]. Объясняется данный факт отсутствием возможности сохранять статические данные в памяти между вызовами. Для каждой итерации, при умножении матриц A и B, все map-задачи получают на входе колонки блока матрицы B, и блок строк матрицы A. На выходе имеем строку результирующей матрицы C. Блок столбцов, связанный с конкретной Map-задачей, не изменяется в течении вычислений, в то время как блок строк изменяется в каждой итерации. В модели Hadoop MapReduce нет возможности хранить статические данные между вызовами. Таким образом, загружаются оба блока и столбы, и строки на каждой итерации вычислений, что снижает производительность по сравнению с MPI-программами.

K-means кластеризация является наглядным примером задач с несколькими итеративными вычислениями, которые выполняются одновременно для общего вычисления. С этой целью выбран алгоритм

кластеризации коллекции 2D точек данных. Для выполнения 16 итераций для 5.12E+ 06 точек, выполнение MPI программы проходит на два порядка быстрее (рис.2.4)[11].

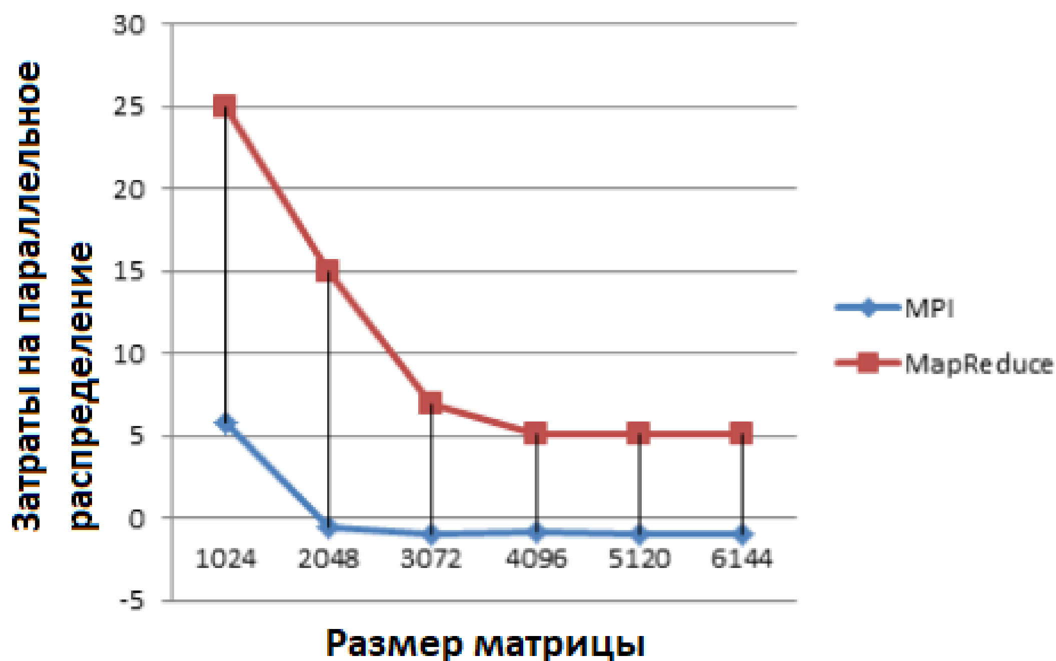

Рисунок 2.3 - Затраты на параллельные распределения программы MapReduce и MPI для перемножения матриц.

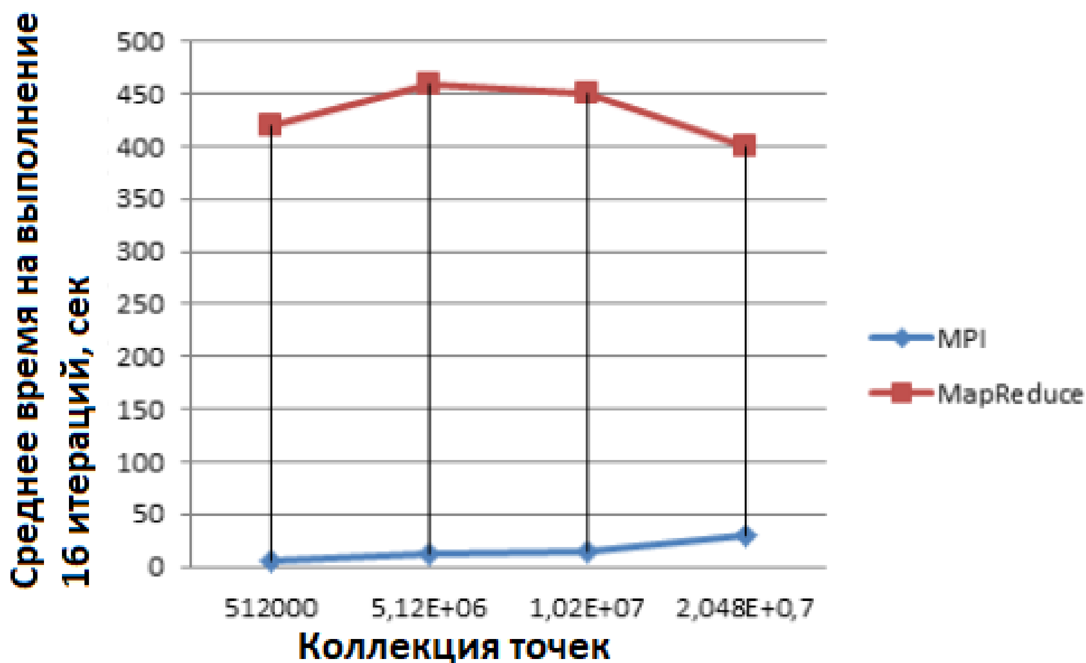

Рисунок 2.4 - Производительность в MapReduce и MPI для K-means кластерных вычислений

Другим аналогом MapReduce является Spark - фреймворк, предназначенный для распределенной обработки данных с использованием специализированных примитивов, позволяющих проводить рекуррентную обработку данных в оперативной памяти. Такой подход позволяет увеличить скорость пакетной обработки данных в 100 раз по сравнению с MapReduce, в котором происходит запись данных на диск [28]. Такая скорость, а также возможность многократного доступа к занесенным в память данным делает Spark привлекательным для разработчиков алгоритмов машинного обучения. Однако, Spark больше подходит для обработки потоковых данных, чем тех, которые уже существуют. Также следует отметить, что пуст Spark обрабатывает данные быстрее, чем MapReduce, но требует много дорогостоящей оперативной памяти.

2.2 Сравнительный анализ технологий для обработки больших данных.

Согласно определенным критериям был осуществлен сравнительный анализ технологий для обработки больших данных, результаты которого представлены в таблице 2.1. Модель MapReduce позволяет относительно легко распараллеливать вычисления, направлять данные в процессы и балансировать нагрузку между ними. MapReduce создана в расчете на использование кластерной аппаратной архитектуры для решения задач параллельной обработки больших данных. Применение данного подхода обработки данных в противовес традиционным решением обоснованно такими преимуществами, как высокая производительность и возможность задания функций на высоком уровне. Как показал проведенный анализ, возможности кода MapReduce гораздо шире SQL, даже без использования специализированных решений.

Представленный обзор данных технологий распределенных вычислений обосновывает выбор именно технологии MapReduce для осуществления обработки больших массивов данных в форме не итеративных алгоритмов, где узлы не требуют обмена информацией (не итеративные и независимые). Масивно-параллельные СУБД выигрывают в работе с структурированными данными, которые часто записываются, а также за счет интегрированности

системы. Отдать предпочтение MPI системам стоит, при необходимости тонкого контроля процесса и интенсивных вычислениях, в то время, как MapReduce хорошо зарекомендовал себя в обработке задач с большим объемом данных. Spark следует применять в случаях, когда используется итеративный алгоритм или требуется доступ к данным или предварительным результатам [23].

Apache Spark - это быстрый и универсальный кластер вычислительной системы. Он предоставляет API-интерфейсы высокого уровня на Java, Scala, Python и R. И также поддерживает богатый набор инструментов высшего уровня, в том числе Spark SQL для SQL и структурированной обработки данных, MLlib для машинного обучения, Graphx для обработки графов, и Spark Streaming. Каждое приложение Spark состоит из управляющей программы, которая запускает основную функцию пользователя и выполняет различные параллельные операции в кластере. Основным понятием в Spark является гибкий распределенный набор данных (Resilient Distributed Dataset), который в свою очередь представляет собой совокупность элементов, распределенных между узлами кластера и способных работать параллельно [23]. RDD создается, начиная с файла в файловой системе Hadoop. Пользователи также могут запрограммировать Spark хранить RDD в памяти, что обеспечивает эффективное повторное использование в параллельных операциях [23]. Также RDD имеет свойство автоматического восстановления после сбоев.

В таблице 2.1 приведен сравнительный анализ технологий распределенных вычислений.

Таблица 2.1 - Сравнительная характеристика использования MapReduce, Spark, MPI и СУБД.

Свойства	**MapReduce**	**Spark**	**MPI**	**СУБД**
Размер данных	Петабайты	Петабайты	Петабайты	Гигабайты
Пакетный доступ	Да	Да	Да	Да
Масштабируемость	Линейная	Линейная	Линейная	Нелинейная
Распределение нагрузки	Автоматически	Автоматически	Программно	Да

Свойства	**MapReduce**	**Spark**	**MPI**	**СУБД**
Локальная оптимизация	Да	Да	Нет	Частично
Обновление данных	Запись один раз, чтение много раз	Запись один раз, чтение один раз	Запись и чтение много раз	Преимущество в работе с одиночными записями, которые постоянно обновляются
Параллельное распределение ресурсов	Автоматически	Автоматически	Программно	Автоматическ и или программно (зависит от вида архитектуры)
Тип вычислений	Неитеративные алгоритмы	Итеративные алгоритмы	Итеративные вычисления	-
Работа с неструктурированным и данными	Да	Да	Да	Нет, необходима нормализация
Требования к специалисту	Программист без опыта работы с параллельными и распределенным и системами	Программист без опыта работы с параллельными и распределенным и системами	Требует от специалиста явной обработки механики потока данных доступных через сокеты и низкоуровневы е подпрограммы С	Аналогичные Map операции, несмотря на то что некоторые СУБД поддерживают определенные пользователем функции, сложно представить в виде SQL

2.3 Выбор технологии для реализации алгоритма.

Существует несколько альтернатив MapReduce, которые имеют свои преимущества и недостатки. Технологию Spark можно назвать конкурирующей с MapReduce из-за того, что она была создана на базе MapReduce и в ней были учтены определенные недостатки этой модели. Также стоит отметить, что Spark загружает данные в оперативную память в то время, как MapReduce выполняет постоянный процесс загрузки данных в оперативную память и запись в файловую систему. Это является самой медленной частью MapReduce.

Среди рассмотренных моделей и методов обработки больших данных была выбрана модель MapReduce. Она обеспечивает отказоустойчивость, берет на себя задачи синхронизации, позволяет обрабатывать большие объемы данных на обычном, широко используемом оборудовании.

3. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА АЛГОРИТМА ПОСТРОЕНИЯ ИНВЕРТИРОВАННОГО ИНДЕКСА НА МОДЕЛИ MAPREDUCE.

3.1 Требования к разрабатываемой программе.

В качестве платформы для распределённых вычислений в рамках модели MapReduce была выбрана платформа Apache Hadoop, как самая стабильная открытая реализация на текущий момент.

Языком программирования, для написания программы был выбран язык Java, так как платформа Hadoop сама реализована на этом языке программирования, следовательно, программы, написанные на Java, будут более эффективны.

Требуется построить инвертированный индекс на основе модели MapReduce для коллекции документов википедии. Инвертированный индекс должен обрабатывать стоп-слова, а также содержать число появления термина в каждом документе.

Результирующий инвертированный индекс должен иметь следующую структуру:

(word, [<docID1, TF>, <docID2, TF> …])

TF(t, d) — это число вхождений слова t в документ d.

Аппаратное обеспечение

- AMD Phenom™ II X6 1100T Processor 3.30 GHz
- RAM: 4 GB
- HDD: SATAII Seagate Barracuda 500 GB

Программное обеспечение

- Ubuntu 16.04
- Java: Oracle JDK 1.8.x
- Apache Hadoop 2.8.4

3.2 Архитектура разрабатываемой программы.

Для начала необходимо определить, какие именно возможности будет иметь пользователь, который будет использовать разработанную программу. Для этого на рисунке 3.1 изображена диаграмма прецедентов (или use-case -

диаграмма). Как видно на диаграмме, у пользователя будут следующие возможности: загрузка исходных данных, загрузка списка стоп-слов, запуск процесса построения инвертированного индекса и загрузка полученного результата.

Рисунок 3.1 – Диаграмма прецедентов программы построения инвертированного индекса.

Диаграмма прецедентов определенным образом изображает требования, предъявляемые к разработчику программного обеспечения. Соответственно, после анализа этой диаграммы следует выделить независимые компоненты программы и описать их взаимодействие на диаграмме классов (рис. 3.2).

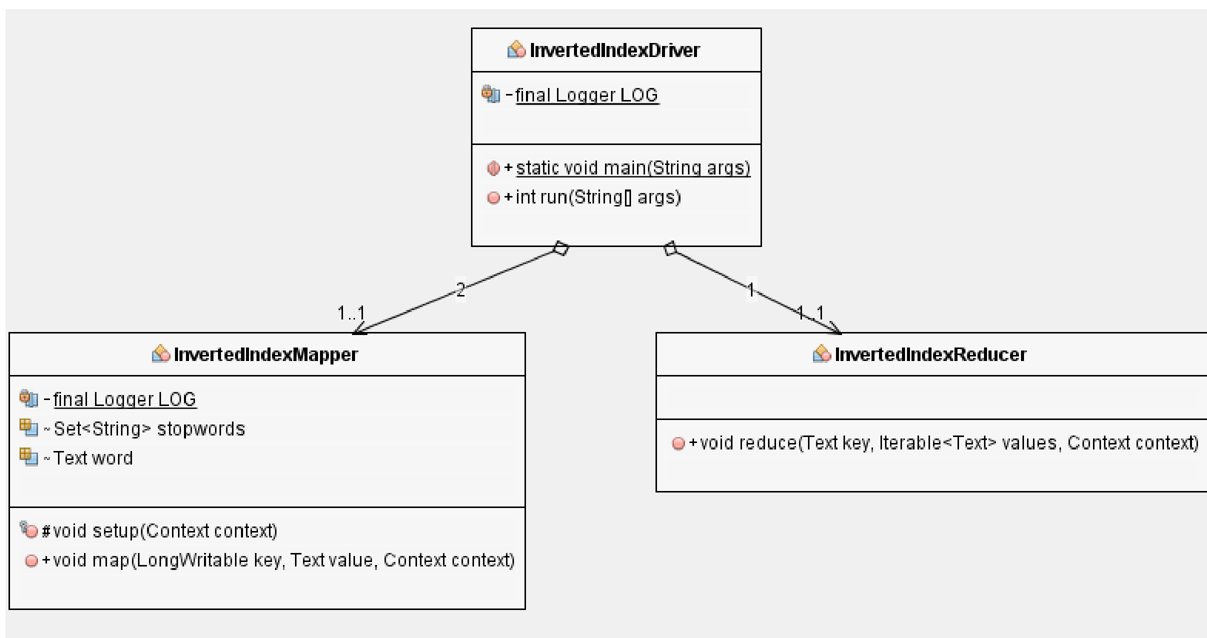

Рисунок 3.2 – Диаграмма классов программы построения инвертированного индекса.

В диаграмме классов представлены классы «InvertedIndexDriver», «InvertedIndexMapper» и «InvertedIndexReducer».

Класс «InvertedIndexDriver» является главным классом программы - он реализует запуск программы и общую настройку работы всей программы.

Класс «InvertedIndexMapper» осуществляет первый этап построения инвертированного индекса – последовательную обработку документа, выявление терминов, обработку стоп-слов и запись промежуточных результатов.

Класс «InvertedIndexReducer» реализует завершающий этап построения инвертированного индекса – производит агрегацию промежуточных результатов по ключу, а также считает количество вхождений термина в каждый документ и записывает итоговый результат.

Стоит также описать работу программы, основываясь именно на технологии MapReduce. На рисунке 3.3 изображена диаграмма деятельности, на которой отображено взаимодействие этапов Map и Reduce с точки зрения строения программы.

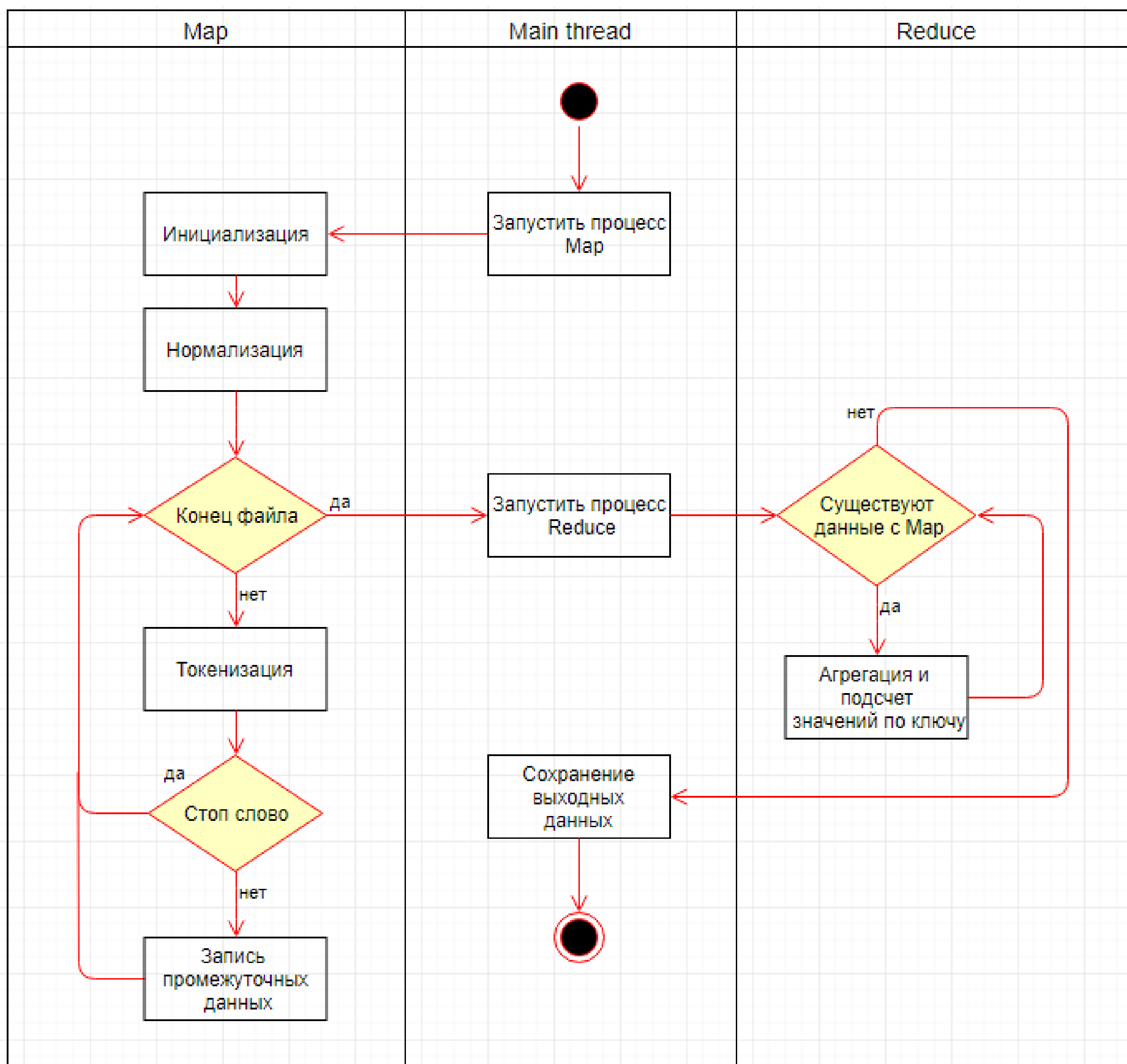

Рисунок 3.3 – Диаграмма деятельности программы построения инвертированного индекса.

Программа начинает работу в главном потоке, который запускает процессы Map. Далее после инициализации этого процесса, на каждом отдельном узле процесс Map открывает соответствующий файл входных данных и начинается его локальная обработка. Обработка включает в себя нормализацию данных, такую, как замена всех знаков препинаний и символов разделителей на пробел, токенизацию или разбиение текста на отдельные слова, исключение стоп-слов и запись промежуточных данных в виде пар <слово, имя документа>. Как только процесс Map на всех узлах завершается, главный поток создает процессы Reduce. В этом процессе происходит

объединение промежуточных данных, полученных из различных вычислительных узлов. Когда все данные обработаны - главный поток завершает работу программы.

3.3 Разработка программы построения инвертированного индекса

Первым делом был реализован класс InvertedIndexMapper, который наследуется от класса Mapper из библиотеки org.apache.hadoop.mapreduce.

Метод map считывает входной набор данных и производит обработку текста путем приведения всех символов к нижнему регистру и заменой всех знаков пунктуации на единый разделитель слов – пробел. Замена происходит при помощи метода replaceAll() класса String. Так же метод производит сравнение каждого слова со списком стоп-слов, и если текущее слово содержится в списке, то оно не записывается в итоговый результат. Результат записывается в виде пар <слово, имя документа>.

```
public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String fileName = ((FileSplit) context.getInputSplit()).getPath()
                .getName();

        String line = value
                .toString()
                .replaceAll("[^\\w\\s]|(.|,|\\?|'|:|;) ", " ")
                .toLowerCase();

        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            String wordText = tokenizer.nextToken();
            if (stopwords.contains(wordText))
                continue;
            word.set(wordText);
            context.write(word, new Text(fileName));
        }
}
```

Рисунок 3.4 – Метод map.

Стоп-слова хранятся в файле в HDFS, который можно кэшировать в память, используя распределенный кэш. Этот файл можно прочитать из метода setup() и сохранить все стоп-слова в hashset.

```java
private static final Logger LOG = Logger
            .getLogger(InvertedIndexMapper.class);
Set<String> stopwords = new HashSet<String>();
Text word = new Text();

@Override
protected void setup(Context context) throws IOException {
        Configuration conf = context.getConfiguration();

        if (conf.getBoolean("wordcount.skip.patterns", false)) {
                URI[] localPaths = context.getCacheFiles();
                Path path = new Path(localPaths[0]);

                try {
                        FileSystem fs = FileSystem.get(context.getConfiguration());
                        FSDataInputStream in = fs.open(path);
                        BufferedReader br = new BufferedReader(
                                        new InputStreamReader(in));
                        String pattern;
                        LOG.info("Adding stopwords to hashset");
                        while ((pattern = br.readLine()) != null) {
                                stopwords.add(pattern);
                        }
                } catch (IOException ioe) {
                        System.err.println("Caught exception while parsing the cached file '"
                                + path);
                }
```

Рисунок 3.5 – Добавление файла стоп-слов

Затем был реализован класс InvertedReducer, который наследуется от класса Reducer библиотеки org.apache.hadoop.mapreduce.

Метод reduce получает данные в виде пар <слово, список документов>. Для хранения имени документа и частоты его появления создан объект fileFreq класса HashMap. Метод обрабатывает данные, считает количество появления слова для каждого документа и записывает данные в виде <word, docName count>

```java
public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
        StringBuilder stb = new StringBuilder();
        HashMap<String, Integer> fileFreq = new HashMap<String, Integer>();

        for (Text val : values) {
                Integer count = fileFreq.get(val.toString());
                if (count == null) {
                        count = 0;
                }
                fileFreq.put(val.toString(), count + 1);
        }
        context.write(key, new Text(fileFreq.toString()));
}
```

Рисунок 3.6 – Метод reduce.

После реализации классов InvertedIndexMapper и InvertedIndexReducer был написан класс InvertedIndexDriver, который наследует класс Configured и реализует интерфейс Tool. Эти библиотеки упрощают запуск команд из командной строки. В этом классе происходит настройка задачи, установка mapper'а, reducer'а, а также типа входных и выходных данных.

Точкой входа является метод main(), в котором вызывается метод ToolRunner.run().

```
public class InvertedIndexDriver extends Configured implements Tool {

        private static final Logger LOG = Logger
                        .getLogger(InvertedIndexDriver.class);

        public static void main(String args[]) throws Exception {
                ToolRunner.run(new InvertedIndexDriver(), args);
        }
```

Рисунок 3.7 – Точка входа в программу.

В методе run() задается конфигурация для заданий, а также добавляется файл стоп-слов в распределенный кэш, при помощи метода job.addCacheFile().

```
@Override
public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args)
                        .getRemainingArgs();
        if (otherArgs.length < 2) {
                System.err.println("Usage: wordcount -skip [wordcount stop word file]"
                        + " <input_file> <output_file>");
                System.exit(2);
        }

        Job job = Job.getInstance(getConf());

        for (int i = 0; i < args.length; i += 1) {
                if ("-skip".equals(args[i])) {
                        job.getConfiguration().setBoolean("wordcount.skip.patterns",
                                        true);
                        i += 1;
                        job.addCacheFile(new URI(
                                        "hdfs://localhost:50070/InvertedIndex/"
                                                        + args[i]));
                        LOG.info("Added file to the distributed cache: " + args[i]);
                }
```

Рисунок 3.8 – Метод run(), добавление файла стоп-слов.

```
job.setJarByClass(InvertedIndexDriver.class);

job.setMapperClass(InvertedIndexMapper.class);
job.setReducerClass(InvertedIndexReducer.class);

job.setInputFormatClass(TextInputFormat.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

Path inputFilePath = new Path(args[0]);
Path outputFilePath = new Path(args[1]);

FileInputFormat.addInputPath(job, inputFilePath);
FileOutputFormat.setOutputPath(job, outputFilePath);

FileSystem fs = FileSystem.newInstance(getConf());
if (fs.exists(outputFilePath)) {
        fs.delete(outputFilePath, true);
}
job.waitForCompletion(true);
return 0;
```

Рисунок 3.9 – Метод run(), настройка задачи.

3.4 Компиляция и запуск программы построения инвертированного индекса в системе Hadoop.

Перед запуском программы ее сперва необходимом скомпилировать. Так как программа содержит специальные библиотеки системы Hadoop, необходимо указать путь к этим библиотекам. Для запуска процесса компиляции нужно запустить терминал, перейти в директорию, где находятся файлы программы и выполнить следующую команду:

*javac –classpath ${HADOOP_CLASSPATH} –d classes *.java*

После этого требуется добавить скомпилированные файлы в архив jar:

jar –cvf InvertedIndex.jar –C classes/ .

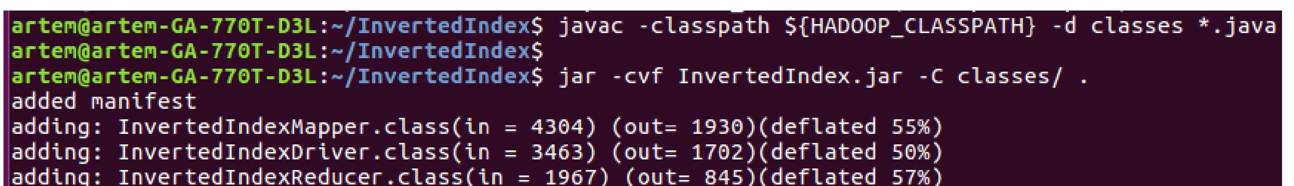

Рисунок 3.10 – Компиляция и помещение в архив файлов программы.

Перед запуском программы, необходимо поместить входные данные в распределенную файловую систему HDFS.

Для этого надо сначала создать директорию:

hadoop fs –mkdir InvertedIndex

hadoop fs –mkdir /InvertedIndex/InputFiles

Для проверки того, что директория создана можно воспользоваться веб-интерфейсом.

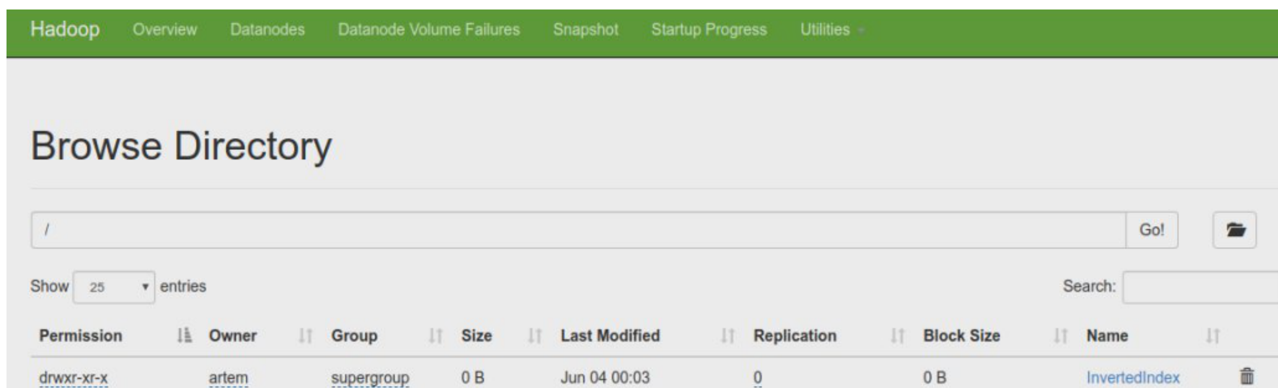

Рисунок 3.11 – Web-интерфейс Hadoop.

Для загрузки входных данных в распределенную файловую систему HDFS используется команда:

Hadoop fs –put /InvertedIndex/InputFiles

Теперь, когда входные данные загружены, можно перейти непосредственно к запуску программы. Для этого выполняется следующая команда:

hadoop jar InvertedIndex.jar InvertedIndexDriver /InvertedIndex/InputFiles /InvertedIndex/Output

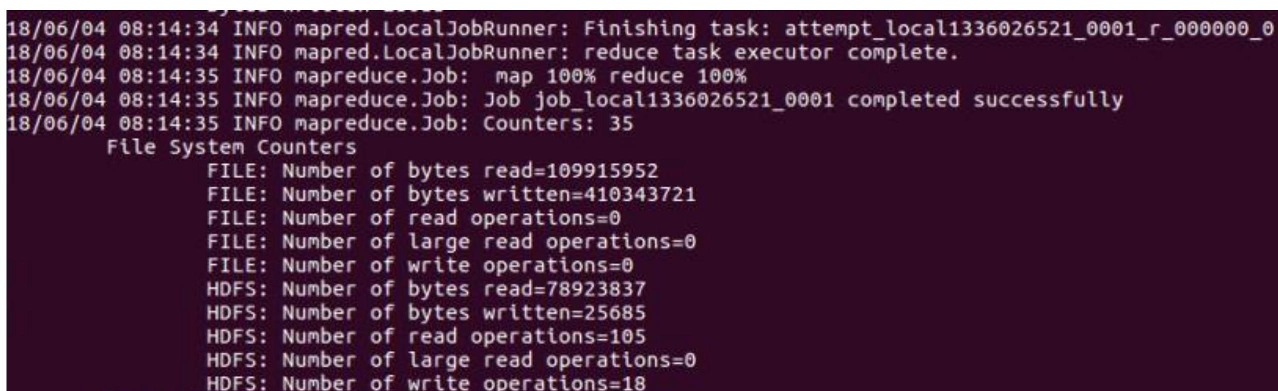

Рисунок 3.12 – Информация об успешном выполнении программы.

Для того, чтобы вывести результат работы программы в терминал, необходимо выполнить команду:

*hadoop dfs –cat /InvertedIndex/Output/**

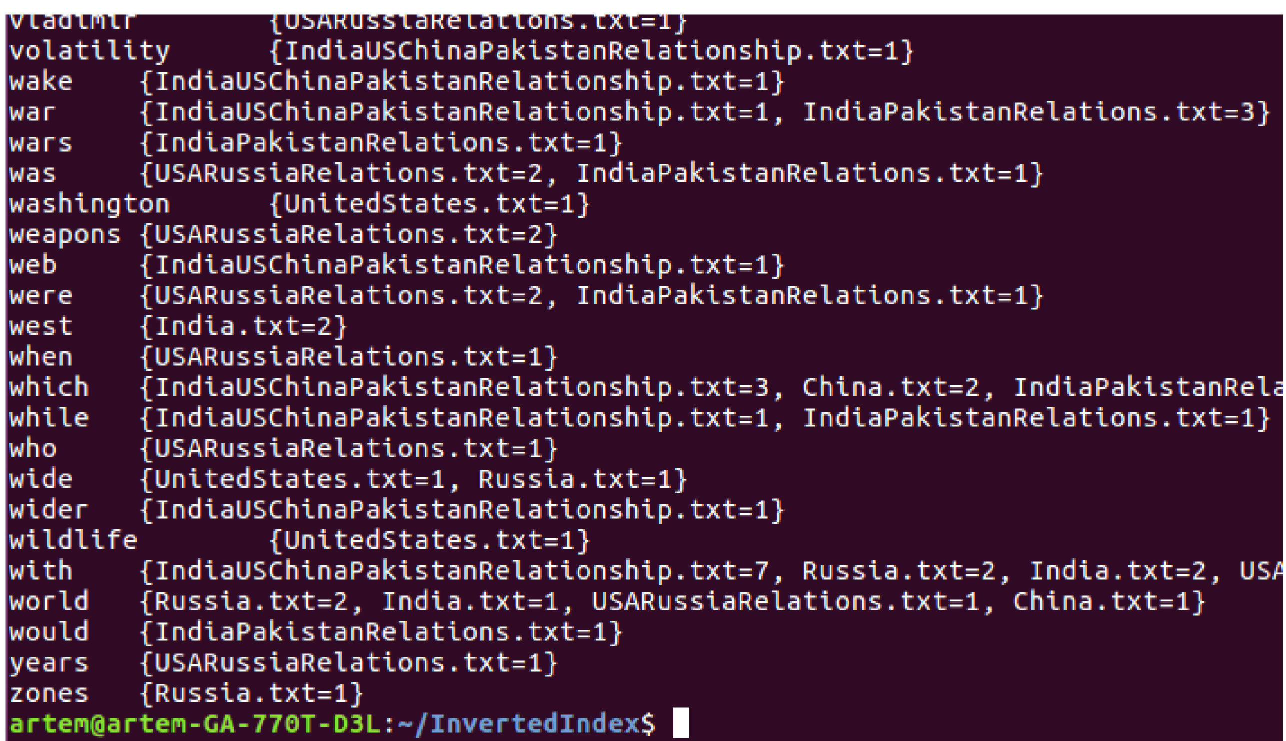

Рисунок 3.13 – Вывод результата выполнения программы.

Как видно на рисунке 3.14, программа успешно построила инвертированный индекс.

3.5 Сравнение с последовательной программой

Для определения эффективности данного метода, была также реализована последовательная программа построения инвертированного индекса (Приложегние Г).

Был проведен эксперимент, подсчитывающий время работы последовательной программы и программы на основе MapReduce. В ходе экспериментов были получены следующие данные представленные в таблице 3.1.

Таблица 3.1 – расчет времени выполнения последовательной и MapReduce программ

	Размер коллекции документов (мегабайт)	**Время выполнения MapReduce программы (секунд)**	**Время выполнения последовательной программы (секунд)**
1	100	67	95
2	200	132	203
3	500	354	472

Для более наглядного сравнения, все результаты, полученные во время проведения эксперимента, были помещены на график, отображающий зависимость времени выполнения программы от размера обрабатываемой коллекции документов.

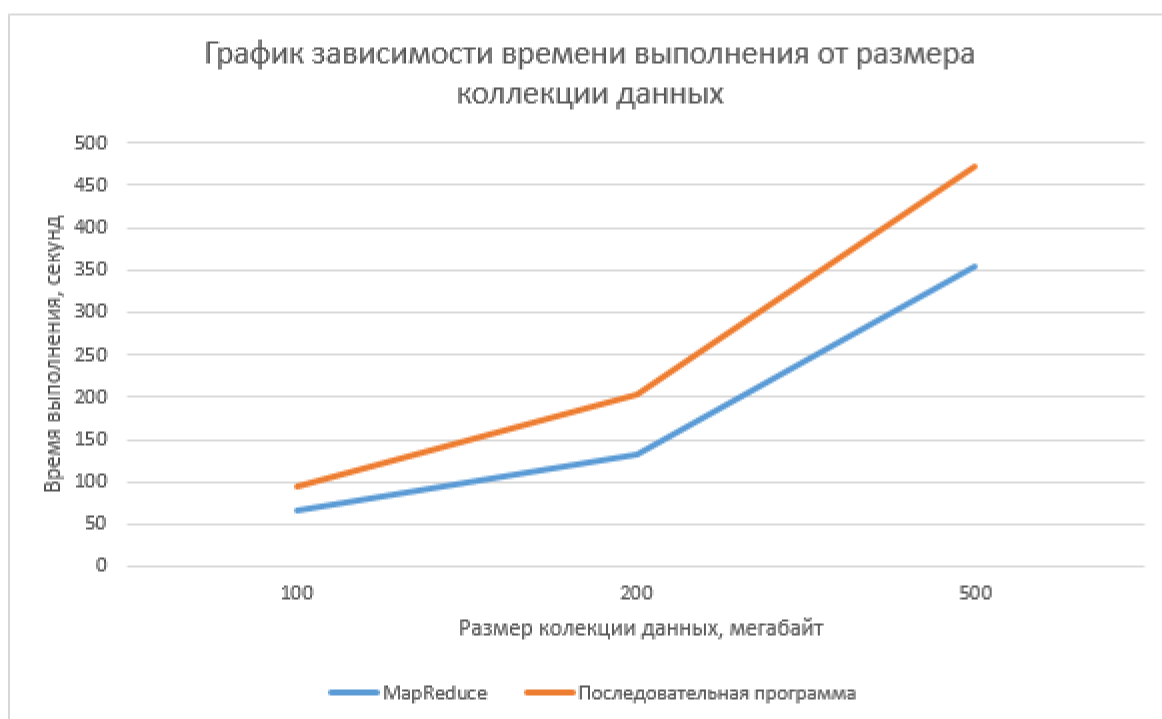

Рисунок 3.14 – График зависимости времени выполнения программы от размера коллекции документов

На основе проведенного эксперимента можно сделать вывод, что программа, основанная на модели MapReduce, является более эффективной, чем последовательная программа.

ЗАКЛЮЧЕНИЕ

Целью бакалаврской работы являлась разработка алгоритма построения инвертированного индекса на основе технологии MapReduce. Создание данного алгоритма позволяет распределено строить инвертированный индекс для большого набора документов, который в дальнейшем может быть использован для полнотекстового поиска.

Были рассмотрены теоретические аспекты и методы построения инвертированного индекса. Проведено сравнение технологий распределенных вычислений и установлено, что MapReduce является наиболее подходящей для данной задачи.

Был разработан алгоритм построения инвертированного индекса на основе технологии MapReduce, учитывающий частоту появления слова в каждом документе и обработку стоп-слов.

Было произведено сравнения времени последовательной программы и программы на основе Mapreduce. Результат эксперимента показал, что реализация, использующая парадигму MapReduce является эффективнее.

В дальнейшем инвертированный индекс можно использовать для поиска по текстам.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Apache Hadoop [Электронный ресурс]. – Режим доступа: http://hadoop.apache.org/, свободный.

2. Arun Murthy, Vinod Vavilapalli, Douglas Eadline, Joseph Niemiec, Jeff Markham. Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2. – "Addison–Wesley Professional", 2014. – 400 с.

3. Sammer E. Hadoop Operations: A Guide for Developers and Administrators // E. Sammer. — М.: Эксмо, 2015. – 328 с.

4. White T. Hadoop: The Definitive Guide, 4th Edition // T. White. — СПб.: Питер, - 2015. - 235 с.

5. Miner, D. MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems // D. Miner, A. Shook. — ACM, 2016. -p. 252.

6. Mayer V. Big Data: A Revolution That Will Transform How We Live, Work and Think// V. Mayer. — The IT University of Copenhagen, Copenhagen, - 2013.

7.Srinath Perera. Hadoop Mapreduce Cookbook. –: "Packt Publishing Ltd", 2013. – 300 с.

8. V. Jain. Big Data and Hadoop. –: "Khanna Publishing", 2017. – 600 с.

9. Jason Venner, Sameer Wadkar, Madhu Siddalingaiah. Pro Apache Hadoop. –: "Apress", 2014. – 444 с.

10. Jimmy L. Data-Intensive Text Processing with MapReduce // L. Jimmy, C. Dyer. — СПб.: Питер, -2010. – 175 с.

11 Т.В.Борис, М.О.Алексеев, Сравнительный анализ технологии параллельного вычисления больших массивов данных MapReduce. - Second International Conference "Cluster Computing", 2013.

12. Vignesh Prajapati. Big Data Analytics with R and Hadoop. –: "Packt Publishing Ltd", 2013. – 238 с.

13. Boris Lublinsky, Kevin T. Smith, Alexey Yakubovich. Professional Hadoop Solutions. –: "John Wiley & Sons", 2013. – 504 с.

14. Kevin Sitto, Marshall Presser. Field Guide to Hadoop: An Introduction to Hadoop, Its Ecosystem, and Aligned Technologies. –: "O'Reilly Media, Inc.", 2015. – 132 с.
15. Garry Turkington. Hadoop Beginner's Guide. –: "Packt Publishing Ltd", 2013. – 398 с.
16. Danil Zburivsky. Hadoop Cluster Deployment. –: "Packt Publishing Ltd",
17. Kevin Roebuck. MapReduce: High–impact Strategies – What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors. –: "Lightning Source", 2011. – 170 с.
18. Donald Miner, Adam Shook. MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems. "O'Reilly Media, Inc.", 2012. – 250 с.
19. Thilina Gunarathne. Hadoop MapReduce v2 Cookbook – Second Edition. –: "Packt Publishing Ltd", 2015. – 322 с.
20. Билл Фрэнкс. Укрощение больших данных: Как извлекать знания из массивов информации с помощью глубокой аналитики. –: "Манн, Иванов и Фербер", 2014.
21. Виктор Майер–Шенбергер, Кеннет Кукьер. Большие данные: Революция, которая изменит то, как мы живем, работаем и мыслим. –: "Манн, Иванов и Фербер", 2013. – 240 с.
22. Peter Bühlmann, Petros Drineas, Michael Kane, Mark van der Laan. Handbook of Big Data. –: "CRC Press", 2016. – 464 с.
23. What Apache Spark Does [Электронный ресурс]. – 2017. – Режим доступа: https://hortonworks.com/apache/spark/.
24. MapReduce: A programming paradigm that allows for massive scalability across hundreds or thousands of servers in a Hadoop cluster [Электронный ресурс]. 2017.– Режим доступа к ресурсу: https://www.ibm.com/analytics/hadoop/mapreduce
25. Exploring Hadoop Framework: Hadoop Distributed File System (HDFS) [Электронный ресурс] // GENTLAB. – 2016. – Режим доступа к ресурсу: https://www.gentlab.com/articles/exploring-hadoop-framework-hadoopdistributed-

file-system-hdfs.
26. A. Harbara, Inverted index implementation, - Masaryk University, 2015
27. Leskovec J. Mining of Massive Datasets / J. Leskovec, A. Rajaraman, J. D. Ullman. – Cambridge: Cambridge University Press, 2014. – 495 p.
28. Jon Z. A profile of Apache Hadoop MapReduce computing efficiency [Электронный ресурс] / Zuanich Jon. – 2010. – Режим доступа к ресурсу: http://blog.cloudera.com/blog/2010/12/a-profile-of-hadoop-mapreducecomputing-efficiency-continued/.

ПРИЛОЖЕНИЕ А

Листинг InvertedIndexDriver

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import java.net.URI;
import org.apache.log4j.Logger;

public class InvertedIndexDriver extends Configured implements Tool {

	private static final Logger LOG = Logger
			.getLogger(InvertedIndexDriver.class);

	public static void main(String args[]) throws Exception {
		ToolRunner.run(new InvertedIndexDriver(), args);
	}

	@Override
	public int run(String[] args) throws Exception {
		Configuration conf = new Configuration();
```

```
String[] otherArgs = new GenericOptionsParser(conf, args)
        .getRemainingArgs();
if (otherArgs.length < 2) {
    System.err.println("Usage: wordcount -skip [wordcount stop
word file]"
        + " <input_file> <output_file>");
    System.exit(2);
}

Job job = Job.getInstance(getConf());

for (int i = 0; i < args.length; i += 1) {
    if ("-skip".equals(args[i])) {

job.getConfiguration().setBoolean("wordcount.skip.patterns",
                true);
        i += 1;
        job.addCacheFile(new URI(
                "hdfs://localhost:50070/InvertedIndex/"
                        + args[i]));
        LOG.info("Added file to the distributed cache: " +
args[i]);
    }
}
job.setJarByClass(InvertedIndexDriver.class);

job.setMapperClass(InvertedIndexMapper.class);
job.setReducerClass(InvertedIndexReducer.class);

job.setInputFormatClass(TextInputFormat.class);
```

```
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        Path inputFilePath = new Path(args[0]);
        Path outputFilePath = new Path(args[1]);

        FileInputFormat.addInputPath(job, inputFilePath);
        FileOutputFormat.setOutputPath(job, outputFilePath);

        FileSystem fs = FileSystem.newInstance(getConf());
        if (fs.exists(outputFilePath)) {
            fs.delete(outputFilePath, true);
        }
        job.waitForCompletion(true);
        return 0;
    }
}
```

ПРИЛОЖЕНИЕ Б

Листинг InvertedIndexMapper:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;
import java.util.HashSet;
import java.util.Set;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;


public class InvertedIndexMapper extends Mapper<LongWritable, Text, Text, Text> {

    Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String fileName = ((FileSplit) context.getInputSplit()).getPath()
                .getName();
```

```
                String line = value.toString().toLowerCase();

                StringTokenizer tokenizer = new StringTokenizer(line);
                while (tokenizer.hasMoreTokens()) {
                        String wordText = tokenizer.nextToken();
                        word.set(wordText);
                        context.write(word, new Text(fileName));
                }
        }
}
```

Приложение В

ЛистингInvertedIndexReducer:

```
import java.io.IOException;
import java.util.HashMap;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class InvertedIndexReducer extends Reducer<Text, Text, Text, Text> {
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        StringBuilder  stb = new StringBuilder();
        HashMap<String, Integer> fileFreq = new HashMap<String,
Integer>();

        for (Text val : values) {
            Integer count = fileFreq.get(val.toString());
            if (count == null) {
                count = 0;
            }
            fileFreq.put(val.toString(),  count + 1);
        }
        context.write(key,  new Text(fileFreq.toString()));
    }
}
```

Приложение Г

Листинг последовательной программы:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
public class InvertedIndex {
List<String> stopwords = Arrays.asList("a", "able", "about",
"across", "after", "all", "almost", "also", "am", "among", "an",
"and", "any", "are", "as", "at", "be", "because", "been", "but",
"by", "can", "cannot", "could", "dear", "did", "do", "does",
"either", "else", "ever", "every", "for", "from", "get", "got",
```

```
"had", "has", "have", "he", "her", "hers", "him", "his", "how",
"however", "i", "if", "in", "into", "is", "it", "its", "just",
"least", "let", "like", "likely", "may", "me", "might", "most",
"must", "my", "neither", "no", "nor", "not", "of", "off", "often",
"on", "only", "or", "other", "our", "own", "rather", "said", "say",
"says", "she", "should", "since", "so", "some", "than", "that",
"the", "their", "them", "then", "there", "these", "they", "this",
"tis", "to", "too", "twas", "us", "wants", "was", "we", "were",
"what", "when", "where", "which", "while", "who", "whom", "why",
"will", "with", "would", "yet", "you", "your");
Map<String, List<Tuple>> index = new HashMap<String, List<Tuple>>();
List<String> files = new ArrayList<String>();


public static void main(String[] args) {
try {
InvertedIndex idx = new InvertedIndex();
for (int i = 1; i < args.length; i++) {
idx.indexFile(new File(args[i]));
}
```

```
} catch (Exception e) {
e.printStackTrace();
}
}
public void indexFile(File file) throws IOException {
int fileno = files.indexOf(file.getPath());
if (fileno == -1) {
files.add(file.getPath());
fileno = files.size() - 1;
}
int pos = 0;
BufferedReader reader = new BufferedReader(new FileReader(file));
for (String line = reader.readLine(); line != null; line = reader
.readLine()) {
for (String _word : line.split("\\W+")) {
String word = _word.toLowerCase();
pos++;
if (stopwords.contains(word))
continue;
```

```
}
List<Tuple> idx = index.get(word);
if (idx == null) {
idx = new LinkedList<Tuple>();
index.put(word, idx);
}
idx.add(new Tuple(fileno, pos));
}
}
System.out.println("indexed " + file.getPath() + " " + pos + " words");
}
private class Tuple {
private int fileno;
private int position;
public Tuple(int fileno, int position) {
this.fileno = fileno;
this.position = position;
}
}
}
```