МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Н. Э. БАУМАНА

Факультет информатики и систем управления

Кафедра теоретической информатики и компьютерных технологий

Курсовой проект

по курсу «Компьютерные системы и сети»

«Фреймворк и файловая система для распределённой обработки больших данных в рамках концепции map-reduce»

Выполнил:
студент ИУ9-91
Выборнов А. И.

Руководитель:
Дубанов А. В.

Москва 2014

Содержание

Введение

С каждом годом объём данных, которые необходимо обрабатывать, растёт, что привело к появлению термина *большие данные* и развитию инфраструктуры по их обработке. Обработка больших данных достаточно молодая и динамично развивающаяся отрасль ИТ.

В ходе данной работы были разработаны и реализованы фреймворк, который позволяет обрабатывать большие данные с использованием концепции map-reduce, и распределённая файловая система, обеспечивающая функционал, необходимый для работы фреймворка.

1. Теоретическая часть

1.1. Map-reduce

Map-reduce — концепция, используемая для распределённых вычислений над *большими данными* в компьютерных кластерах. Модель представлена компанией Google в 2004 году [1].

Большие данные (Big data) — термин, характеризующий любой набор данных, который достаточно велик и сложен для традиционных методов обработки, а также набор технологий для работы с такими данными.

Выполнение приложения в рамках концепции map-reduce состоит из двух последовательных этапов, между которыми происходит группировка результатов:

- *map* — предварительная обработка входных данных,
- *reduce* — свёртка предварительно обработанных данных.

Несмотря на то, что прототипами этапов *map* и *reduce* послужили одноимённые функции, используемые в функциональном программировании, — их семантика заметно отличается.

Базовым элементом в концепции map-reduce является структура (key, value) — (ключ, значение). Программирование представляет собой определение двух функций (квадратные скобки [ ] обозначают список):

- $map : (key, value) \to [(key, value)]$
- $reduce : (key, [value]) \to [(key, value)]$

Приведённое выше определение является достаточно абстрактным для прикладного применения. На практике достаточно следующего варианта выше приведённого определения:

- map : файл ввода $\to [(key, value)]$
- $reduce : (key, [value]) \to$ файл вывода

На рисунке 1 схематически изображена обработка данных с помощью map-reduce. Файл ввода разбивается на блоки, каждый из которых поступает на вход функции *map*. Функция map обрабатывает его и порождает список пар (ключ, значение). Множество полученных списков пар объединяется и группируется по ключу, получая пары вида (ключ, список значений). Все пары равномерно распределяются

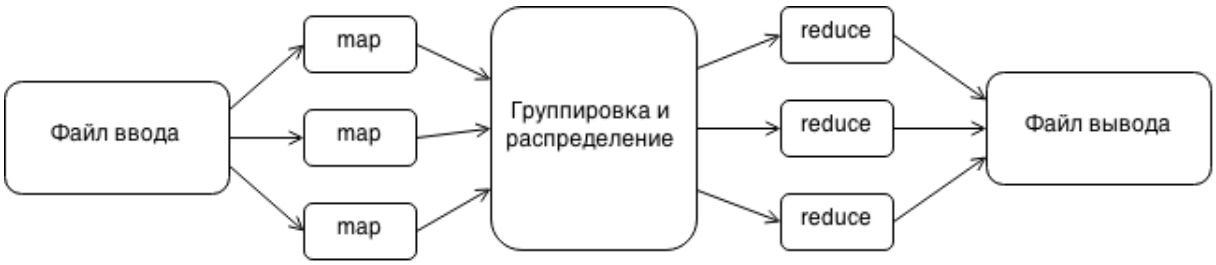

Рисунок 1 — Схематическое изображение обработки данных с помощью концепции map-reduce

на блоки и передаются на вход стадии *reduce*. Стадия *reduce* преобразовывает ключ и список значений в блок, являющийся частью файла вывода.

Как видно из описания, концепция налагает ограничения на формат файлов ввода и вывода: данные должны быть коммуникативны, с некоторой точностью (обычно до строки).

1.1.1. Проблемы, которые решаются с помощью map-reduce

Концепция map-reduce, довольно быстро обрела популярность, так как накопилось множество проблем, не имеющих универсального решения [2]. Вот основные из них (в скобках указан подход к решению используемый в концепции map-reduce):

- вычисления превосходят возможности одной машины (кластеры из сотен и тысяч машин),
- данные не помещаются в памяти, необходимо обращаться к диску (последовательное чтение и запись намного эффективнее случайного доступа),
- большое количество узлов в кластере вызывает множество отказов (все узлы унифицированы, что упрощает восстановление работы после отказа),
- данные хранятся на множестве машин (данные обрабатываются на той же машине, на которой они хранятся),
- разработка низкоуровневых приложений для подобных систем достаточно дорогая и сложная (высокоуровневая модель программирования, универсальная и масштабируемая среда выполнения).

1.1.2. Пример применения map-reduce

Задача: Есть граф пользователей некоторого ресурса, заданный в виде пар (пользователь, «друг$_1$ друг$_2$ ...»). Для каждой пары пользователей найти общих друзей.

Разбор решения задачи будет проводится для следующих входных данных:

(A, B C D)
(B, A C)
(C, A B D)
(D, A C)

Пусть функция map преобразовывает пару (пользователь, друзья) в множество пар. Для каждого друга формируется пара (ключ, значение) следующим образом: ключ — строка «пользователь друг», если строка «пользователь» > строки «друг», иначе строка «друг пользователь», значение — «друзья». Выполнения функции map в формате (входные даннные → результат):

(A, B C D) → [(A B, B C D), (A C, B C D), (A D, B C D)]
(B, A C) → [(A B, A C), (B C, A C)]
(C, A B D) → [(A C, A B D), (B C, A B D), (C D, A B D)]
(D, A C) → [(A D, A C), (C D, A C)]

По выполнении функции map происходит подготовка данных для функции reduce, множество полученных пар агрегируется по ключу:

(A B, [B C D, A C])
(A C, [B C D, A B D])
(A D, [B C D, A C])
(B C, [A B D, A C])
(C D, [A B D, A C])

Функция reduce пересекает все элементы списка значений. Выполнение функции reduce в формате (входные даннные → результат):

(A B, [B C D, A C]) → (A B, C)
(A C, [B C D, A B D]) → (A C, B D)
(A D, [B C D, A C]) → (A D, C)
(B C, [A B D, A C]) → (B C, A)
(C D, [A B D, A C]) → (C D, A)

По выполнении функции reduce получили множество пар (ключ значение), где ключ — пара пользователей, значение — множество общих друзей:

(A B, C)
(A C, B D)
(A D, C)
(B C, A)
(C D, A)

1.2. Распределённая файловая система

Распределённая файловая система (РФС) — файловая система, в которой данные хранятся на нескольких узлах.

При реализации фреймворка map-reduce РФС требуется для хранения одного файла в виде набора блоков, распределённых по нескольким узлам, а также для создания файла из блоков данных, расположенных на нескольких машинах.

Распределённая файловая система является вспомогательным компонентом, поверх которого работает map-reduce. В рамках данной работы рассматривалась упрощенная модель файловой системы, которая обеспечивает все потребности map-reduce, но при этом не обладает некоторыми важными возможностями файловых систем, такими как расширенные характеристики файлов и реплицируемость.

В рамках работы была разработана РФС, которая позволяет хранить файлы, упорядоченные с помощью древовидной структуры каталогов. Файл представляет собой именованную последовательность блоков данных, которые распределены по нескольким узлам. Файл бьётся на блоки равного размера (64 Мбайт) с точностью до переносов строк.

На рисунке 2 показана общая архитектура фреймворка map-reduce. Рассмотрим часть, связанную с РФС. Пользователь работает с элементом РФС расположенном на главном узле, на котором хранится структура файловой системы и информация о блоках, составляющих каждый файл. Вспомогательные узлы хранят блоки и имеют интерфейс, позволяющий создавать, получать, удалять и переименовывать блоки.

1.3. Распределённый map-reduce

Рассмотрим рисунок 2. Пользователь работает с интерфейсом map-reduce на главном узле. Map-reduce на главном узле занимается управлением процессом выполнения задачи, может взаимодействовать с РФС на главном узле, а также с map-reduce на дочерних узлах. Map-reduce на дочерних узлах выполняет код функций

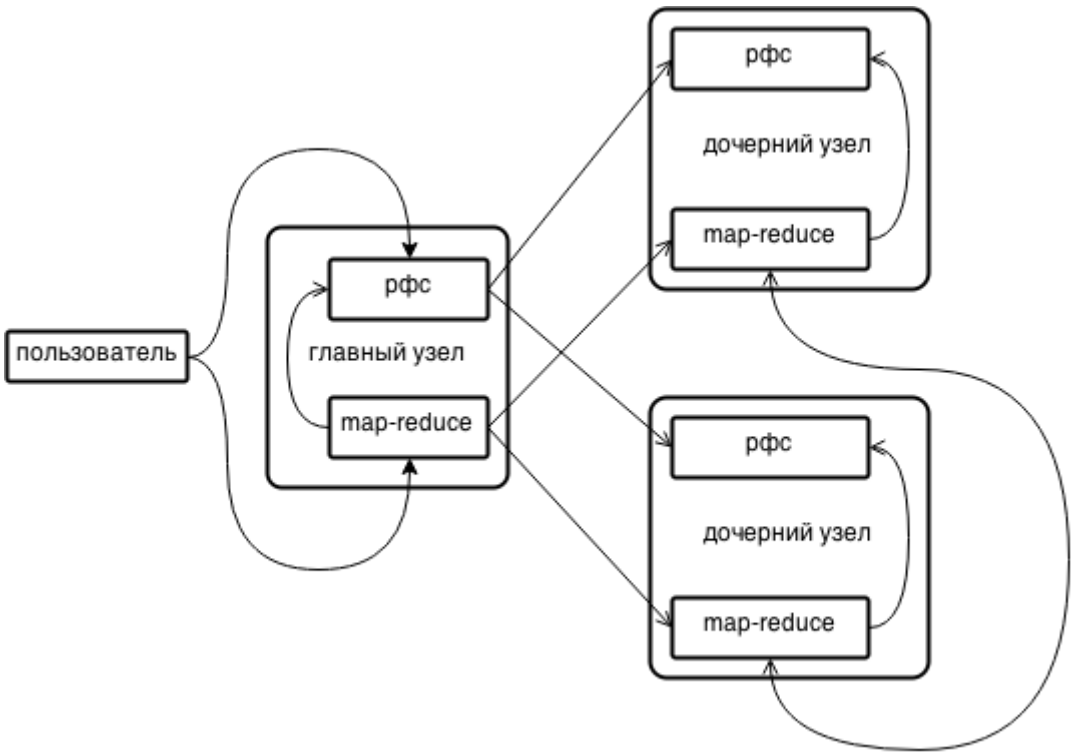

Рисунок 2 — Взаимосвязь узлов при работе фреймворка map-reduce

map и reduce, а также записывает и читает данные, взаимодействуя с элементом РФС на дочернем узле.

Выполнение программы с помощью фреймворка map-reduce происходит следующим образом:

1. На вход подаются адреса файлов из РФС, для ввода и вывода, и файл с функциями map и reduce.
2. Из РФС получается список индексов, соответствующий файлу ввода.
3. Полученный список индексов группируется по узлам.
4. Каждому дочернему узлу сообщается о начале стадии map: передаётся соответствующий ему список индексов и файл с функциями map и reduce.
5. Выполняется стадия map на дочерних узлах. Стадия map порождает данные, равномерно распределяющиеся по дочерним узлам.
6. Каждому дочернему узлу сообщается о начале стадии reduce.
7. Выполняется стадия reduce на дочерних узлах. Стадия reduce порождает множество блоков, задаёт им псевдоимена и сообщает об этом главному узлу.

8. Главный узел производит переименование блоков путём задания им новых имён и записывает информацию о файле вывода в РФС.

На рисунке 3 показано выполнение стадий map и reduce (отделены пунктиром) на двух узлах. Выполнение стадий происходит следующим образом:

- Стадия map.

    1. Для каждого блока РФС, соответствующего полученному индексу, выполняется функция map. Блоки обрабатываются последовательно.
    2. Во время обработки блока все результаты попадают в пул данных, где распределяются по дочерним узлам.
    3. По окончании обработки блока пул данных рассылает полученные результаты по соответствующим дочерним узлам.
    4. Результаты агрегируются на узлах в оперативной памяти.

- Стадия reduce.

    1. Для каждой пары (ключ, список значений) выполняется функция reduce.
    2. Результаты выполнения функции reduce агрегируются в оперативной памяти.
    3. По завершении обработки всех данных результат преобразуется в текст, который разбивается на блоки фиксированного размера (64 Мбайт).
    4. Полученные блоки записываются в РФС.

1.4. Решаемый класс задач

Необходимо обосновать существование и найти класс задач (здесь и далее под классом подразумевается некоторое множество задач), для которого актуальна приведённая выше схема распределённого map-reduce.

Пусть каждый узел располагает $memory$ доступной оперативной памяти (здесь и далее единицей измерения памяти будет байт) и $storageSize$ свободного места на диске. Всего узлов $numNodes$. Данный map-reduce проектировался для решения задач с *большими данными*, поэтому в дальнейшем будем считать, что входные данные $>> memory$.

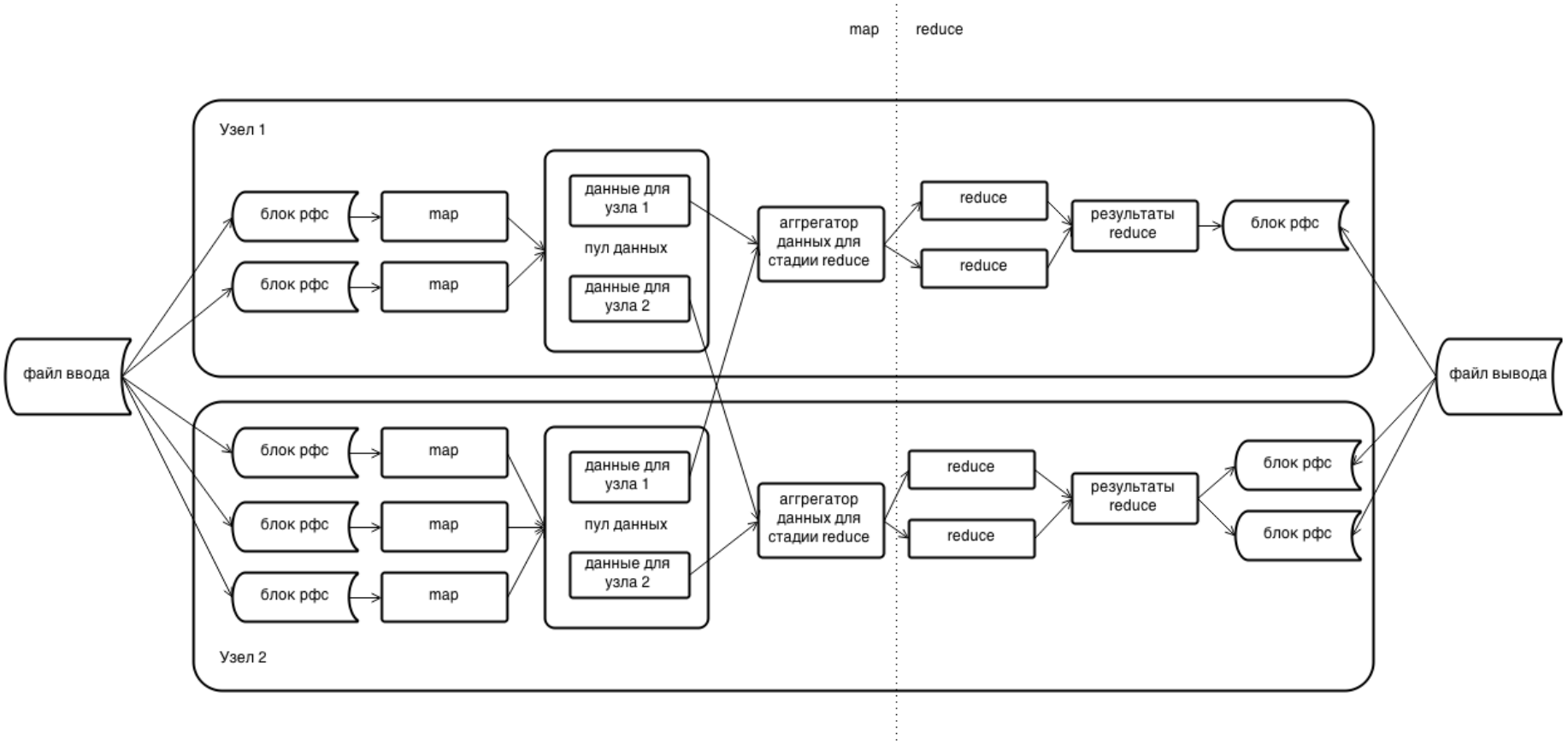

Рисунок 3 — Схема выполнения приложения с помощью фреймворка map-reduce

1.4.1. Ограничения на стадии map и reduce

На вход стадия map принимает данные с диска, а результат этой стадии равномерно (с точностью до пары (ключ, значение)) распределяется по оперативной памяти узлов. То есть входные данные $< storageSize * numNodes$, а результат $<< memory * numNodes$. Так как $storageSize >> memory$, получаем, что, в общем случае, стадия map должна сильно сокращать объём данных.

На вход стадия reduce принимает данные из оперативной памяти, которые $<< memory * numNodes$, результат этой стадии аккумулируется в оперативной памяти, то есть должен быть $<< memory$ для каждого узла, а затем записывается на диск. Получаем, что стадия reduce должна, как минимум, не увеличивать объём данных, поступивших на вход.

Рассмотрев полученные выше ограничения для стадий map и reduce, а также учитывая достаточно большие входные данные, можно увидеть, что распределённый map-reduce оптимально подходит для решения задач класса *информационный поиск* (*information retrieval*), что является одним из этапов решения задач *анализа данных* (*data mining*).

Информационный поиск — процесс поиска и получения информации как из структурированных, так и из неструктурированных данных. Обычно применяется в *анализе данных* для первичной обработки и сокращения объёма исходных данных [4].

1.4.2. Пример задачи класса информационный поиск

Исходные данные хранятся в виде текстового файла в n строк, в котором каждая строка соответствует строке в реляционной таблице $table$ с m столбцов ($f_1 ... f_m, m > 2$). Необходимо получить результат выполнения SQL запроса:

SELECT $f_0, ..., f_m$
FROM $table$
WHERE $f_j > const$

и сгруппировать его по ключу f_i. В результате все строки с одинаковым f_i должны идти одним непересекающимся блоком.

1.4.3. Решение задачи с помощью фреймворка map-reduce

Функция map определяется следующим образом: для каждой строки проверяется условие **WHERE** и все строки, удолетворяющие условию, составляют результат в виде пар: $(f_i, f_1, ..., f_m)$. Количество строк, удолетворяющих условию **WHERE**, обозначим как n'. Функция reduce возвращает список значений.

Стадия map получает $O(nm)$ данных и, по выполнении, выдаёт $O(n'm)$ данных, которые преобразуются и без изменений проходят стадию reduce. Сложность стадии map — $O(nm)$, преобразования между стадиями — $O(n'm + n'ms_{net})$ (s_{net} — стоимость передачи данных между узлами), стадии reduce — $O(n')$.

1.4.4. Решение задачи на одной машине

Решение задачи на одной машине без применения фреймворка состоит из двух стадий:

- получить из входного файла все строчки, удолетворяющие **WHERE** — сложность $O(nm)$ (по памяти $O(n'm)$),
- сгруппировать полученный результат по ключу f_i — сложность $O(n'ln(n'))$, если применить для агрегации быструю сортировку (по памяти $O(n'm)$), сложность $O(n')$, если применить для аггрегации сортировку подсчётом (по памяти $2O(n'm)$).

В случае когда n' и m определены таким образом, что получишиеся данные больше $memory$, возникают проблемы: необходимо сохранять на диск промежуточные результаты, полученные после первой стадии, и сортировать полученный результат на диске, что достаточно медленно. Рассмотрим вариант решения этих проблем: результат первой стадии разбивается на блоки, каждый из которых сортируется и сохраняется в файл, после обработки всех результатов полученные файлы сливаются). Сложность первой стадии — $O(nm + n'ln(n') + n's_{hdd})$, второй стадии $O(n'ms_{hdd})$ (s_{hdd} — стоимость обращения к диску).

1.4.5. Выводы

Суммарная сложность решения с помощью map-reduce — $O(nm + n'm + n'ms_{net} + n')$, с помощью описанного выше способа решения на одной машине — $O(nm + n'ln(n') + n's_{hdd} + n'ms_{hdd})$. Можно увидеть, что по сложности данные алгоритмы принципиально не отличаются (в обоих случаях сложность $O(nm)$) за исключением двух важных моментов (их влияние будет оценено в главе 4):

- обычно $s_{net} >> s_{hdd}$,
- время выполнения на одной машине существенно увеличивается за счёт последовательного выполнения всех действий, которые происходят параллельно в случае map-reduce.

Также необходимо отметить, что решение задачи на одной машине осложняется реализацией механизмов, альтернатива которых уже реализована в фреймворке.

Можно сказать, что предложенная схема реализации концепции map-reduce позволяет решать задачи класса информационный поиск. Эффективность решения таких задач оценена в главе 4.

Более подробную информация о задачах, которые решаются с помощью концепции map-reduce можно найти в [5].

2. Объекты и методы

Характеристики программного обеспечения:

- Операционная система — OpenSUSE 12.2 x86_64.
- Язык программирования — Python 2.7.3.

Характеристики оборудования:

- Процессор — Intel Core 2 Duo E6550 2.33 Гц 2 ядра.
- Оперативная память — 2 Гбайт DDR2.

3. Реализация

3.1. Используемые технологии

3.1.1. Коммуникация по сети

Распределённый фреймворк подразумевает согласованную работу нескольких узлов, которые образуют вычислительный кластер. Для коммуникации узлов друг с другом была использована библиотека ZeroMQ.

ZeroMQ — высоко-производительная библиотека для асинхронного обмена сообщениями по сети. Обладает интерфейсом программирования приложений сокет, который и был использован в рамках данной работы.

Библиотека ZMQ была выбрана ввиду поддержки асинхронных соединений, высокой производительности (приемлемой в рамках данной работы) и удобного интерфейса программирования приложений.

3.1.2. Сжатие данных

При работе РФС, по сети передаётся большое количество однотипных данных, что занимает значительное время. Сократить время передачи данных можно с помощью сжатия данных перед отправкой и распаковкой по получении.

Стандартная библиотека языка Python предоставляет сразу два способа сжатия данных: zlib и bz2. Как zlib, так и bz2 поддерживает несколько режимов сжатия, пронумерованных от 1 до 9 (с увеличением номера данные сжимаются сильнее, но это занимает больше времени).

В качестве входного файла берутся входные данные для задачи описанной в главе 1.4.2 размером 95 Мбайт.

Для каждого способа сжатия и для каждого режима используется метрика: $\frac{\Delta * 8}{t}$, где Δ — разность между размером несжатого и сжатого файла, t — суммарное время сжатия и распаковки. Приведённая метрика характеризует пропускную способность сети в Мбит/с, минимально необходимую для того, чтобы применение алгоритма сжатия было бессмысленно. Чем больше полученное значение для конкретного алгоримта, тем лучше он подходит для решения задачи. Результаты измерений записаны в таблице 1.

По результатам тестирования получили, что наилучшее время показал алгоритм zlib с установленным режимом 1. Причём полученное значение метрики показывает, что данный алгоритм будет актуален для всех сетей с пропускной способностью меньшей чем 248 Мбит/с.

Алгоритм	Режим	$\frac{\Delta*8}{t}$
zlib	1	248.75
zlib	2	219.16
zlib	3	146.66
zlib	4	181.69
zlib	5	90.68
zlib	6	37.30
zlib	7	37.00
zlib	8	37.03
zlib	9	37.26
bz2	1	53.59
bz2	2	52.69
bz2	3	51.90
bz2	4	50.93
bz2	5	50.36
bz2	6	49.01
bz2	7	49.48
bz2	8	48.90
bz2	9	49.27

Таблица 1: Результаты тестирования способов сжатия

3.1.3. Сериализация

Сериализация — процесс перевода структуры данных в последовательность битов. Обратной к операции сериализации является операция *десериализации* — восстановление начального состояния структуры данных из битовой последовательности.

При реализации фреймворка необходимо передавать сложные структурированные данные по сети. Данные сериализуются в строку, которая передаётся с помощью ZMQ. Я рассматривал следующие сериализаторы: pickle, cPickle, json, cjson, marshal [3]. Pickle и cPickle поддерживают режимы сериализации, которые влияют на работу алгоритма.

В качестве тестовых данных используется dict (ассоциативный массив в языке python), содержащий 100 ключей в виде строк из 6 случайных букв латинского алфавита. Каждому ключу поставлен в соответствие список из 10000 случайных целых чисел в диапазоне от 0 до 1000000. Результаты тестирования приведены в таблице 2 (под временем работы подразумевается суммарное время сериализации и десериализации).

По итогам тестирования был выбран marshal, так как он затрачивает значительно меньше времени, чем прочие сериализаторы, и при этом порождает сериали-

Сериализатор	Режим	Время работы (с)	Размер получаемых данных (байт)
pickle	0	2.40	8890380
pickle	1	1.79	4871161
pickle	2	1.80	4871163
cPickle	0	0.25	8890382
cPickle	1	0.05	4870961
cPickle	2	0.05	4870963
json		0.15	7889382
cjson		0.13	7889382
marshal		0.03	5001602

Таблица 2: Результаты тестирования сериализации

зованные данные относительно малого размера.

Также стоит отметить, что были использованы сериализаторы, которые порождают человекочитаемые данные: json — для файла конфигурации и lxml — для хранения дерева файловой системы.

3.2. Работа с большими данными

3.2.1. Распределение пар (ключ, список значений) по узлам

Для корректной работы стадии reduce необходимо, по возможности равномерно, распределить пары (ключ, список значений), являющиеся результатом работы стадии map по узлам. Так как распределение происходит не для полностью сформированных пар, а для пар, порождённых одним блоком исходных данных, то необходимо обеспечить попадание всех пар с общими ключами на один узел. Это решается вводом функции get_node, которая позволяет по ключу и количеству узлов однозначно получить целевой узел. Код функции get_node представлен ниже на языке Python:

```
def get_node(key, num_of_nodes):
    node = (key.__hash__()) % num_of_nodes
    return node
```

Использование хеш-функции от ключа позволяет равномерно распределить пары (ключ, список значений) по узлам с точностью до узла.

3.2.2. Разбиение большого файла на блоки

При работе с большими данными файлы удобно представлять в виде достаточно маленьких блоков для беспроблемной обработки каждого из них. Разбиение

файла на блоки происходит с помощью функции split, которая также позволяет порождать блоки из произвольных данных, представленных в виде списка. Функция split в виде кода на языке Python:

```
def split(elem_list, format_func, block_size_limit_mb=64):
    block_size_limit = block_size_limit_mb * 1024 * 1024
    block = []
    block_size = 0
    for elem in elem_list:
        to_append = format_func(elem)
        block.append(to_append)
        block_size += len(to_append)
        if block_size > block_size_limit:
            block_size = 0
            yield ''.join(block)
            block = []
    if block_size != 0:
        yield ''.join(block)
```

На вход функция split получает список elem_list, функцию для отображения элемента списка в строку format_func, и размер блока в мегабайтах block_size_limit_mb и возвращает генератор, позволяющий порождать блоки. Для обработки больших данных в качестве elem_list можно передать генератор.

3.3. Взаимодействие между узлами

Взаимодействие между узлами происходит с помощью интерфейса сокет, предоставляемого библиотекой ZMQ. Работу клиентского сокета обеспечивает класс nodes_manager. Этот класс имеет следующий интерфейс:

- send(node_index, func_word, data) — на узел с номером node_index отправляет данные data с ключевым словом func_word. Функция send создаёт структуру, которая состоит из данных, ключевого слова, обозначающего конкретное действие, и различной вспомогательной информации, а также сериализует её. Затем происходит отправка сериализованных данных по сети. Результаты всех функций send агрегируются в асинхронной очереди, в качестве которой выступает асинхронный класс Queue, предоставляемый стандартной библиотекой Python.

- wait() — блокирует выполнение программы до тех пор, пока на все вызванные ранее send не будет получен ответ.
- flush_q() — очищает очередь, в которой хранятся ответы на все отправленные с помощью функции send запросы, и возвращает список полученных ответов.

Сценарий работы с nodes_manager достаточно прост. В произвольном месте кода, можно вызвать функцию send, с ограничением: один вызов функция send для одного узла. После вызова всех необходимых функций send необходимо, с помощью функции wait, подождать их завершения. Когда все функции send завершатся, их результат можно получить с помощью функции flush_q.

На стороне сервера обработка производится путём взаимодействия с интерфейсом ZMQ. При поступлении запроса происходит поиск ключевых слов в полученных данных и, при успешном результате, выполняется код, соответствующий найденному ключевому слову.

3.4. Интерфейс

В качестве интерфейса пользователя выступает набор python-скриптов: dfs.py, dfs_slave.py, mr.py, mr_slave.py. Скрипты делятся на две категории: скрипты с префиксом dfs и с префиксом mr, которые являются скриптами для работы с РФС и для работы с map-reduce соответственно. Также элементом интерфейса является json-файл конфигурации

3.4.1. Скрипты для работы с РФС

На каждом дочернем узле запускается скрипт dfs_slave.py с двумя аргументами: -p и -s, задающими порт и путь до директории хранения блоков данных соответственно. Пример команды запуска скрипта:

```
python dfs_slave.py -p 5556 -s /home/username/storage
```

На главном узле можно запустить скрипт dfs.py, который предоставляет интерфейс для работы с РФС, а именно, выполнение команд: -ls — получить содержимое каталога, -mkdir — создать каталог, -put — положить локальный файл в РФС, -get — получить файл из РФС, -rm — удалить каталог и всё его содержимое или файл из РФС. Пример использования команд:

```
python dfs.py -ls /user/
python dfs.py -mkdir /user/username/data_folder
python dfs.py -put ./test /user/username/data_folder/test
```

```
python dfs.py -get /user/username/user_data_folder/test
python dfs.py -rm /user/username
```

3.4.2. Скрипты для работы с map-reduce

На каждом дочернем узле запускается скрипт mr_slave.py с двумя аргументами: -p1 и -p2, задающими два порта: p1 для взаимодействия с главным узлом, p2 для взаимодействия с дочерними узлами. Пример команды запуска скрипта:

```
python mr_slave.py -p1 5557 -p2 5558
```

Для запуска задачки с использованием фреймворка необходимо использовать скрипт mr.py с тремя аргументами -i, -o, -mr, задающими файл ввода в РФС, файл вывода в РФС и python файл с реализациями функций map и reduce соответственно. Пример запуска задачи с использованием фреймворка:

```
python mr.py -i /in -o /out -mr wordcount.py
```

3.4.3. Файл конфигурации

Для работы с фреймворком необходимо заполнить файл *config.json*, в котором в формате json задаются параметры дочерних узлов. Для каждого дочернего узла задаётся url и порты для работы с скриптами на дочерних узлах. Пример заполнения файла:

```
{
"nodes": [
    {
        "url":"tcp://localhost",
        "dfs_port":"5559",
        "mr_port":"5557",
        "mr_slave_port":"5558"},
    {
        "url":"tcp://192.168.1.1",
        "dfs_port":"5559",
        "mr_port":"5557",
        "mr_slave_port":"5558"}]
}
```

3.4.4. Файл с функциями map и reduce

На вход фреймворку передаётся python файл с функциями map_func и reduce_func. Функция map_func принимает на вход строку и возвращает генератор, содержащий пары (ключ, значение). Функция reduce_func принимает на вход пару (ключ, список значений) и возвращает список пар (ключ, значение).

Пример реализации функций map_func и reduce_func на примере решения задачи описанной в главе 1.4.2:

```
def map_func(string):
    domen = string.split('\t', 2)
    if int(domen[1]) > 500:
        yield (domen[1], string)

def reduce_func(key, values):
    return [('', value) for value in values]
```

4. Тестирование

Тестирование на производительность и работоспособность фреймворка производилось с помощью двух задач: wordcount и задачи класса информационный поиск, описанной в главе 1.4.2. В качестве оборудования выступает кластер из 6 узлов, описанных в главе 2. Все 6 узлов выступают в роли дочерних, и один из них выступает в роли главного узла.

Wordcount — задача, заключающаяся в подсчёте числа употреблений каждого слова в тексте. Для тестирования данной задачи было написано две реализации: одна для выполнения в виде отдельной программы на одной машине, другая для выполнения с помощью фреймворка map-reduce. В качестве входных данных использовался сгенерированный текстовый файл, содержащий разделённые пробелами слова, объёмом 1 Гбайт.

На рисунке 4 показаны графики зависимости времени выполнения задачи wordcount и загрузки данных в РФС (команда dfs.py -put) от количества узлов в вычислительном кластере. Как видно из этого рисунка загрузка данных в РФС не зависит от числа узлов. Время выполнения wordcount экспоненциально уменьшается при росте числа узлов.

На рисунке 5 показаны графики зависимости времени выполнения задачи wordcount и задачи wordcount вместе с загрузкой данных в РФС от количества узлов. Также для сравнения добавлен график, показывающий время выполнения wordcount в виде отдельной программы на одной машине. Из рисунка видно, что, при условии предзагруженного в РФС файла ввода, время выполнения wordcount с помощью map-reduce на кластере значительно меньше времени выполнения wordcount без фреймворка на одной машине, при использовании кластера из более чем 3 машин. Также из рисунка можно получить, что выполнение задачи wordcount на кластере без предзагруженных данных (добавляется загрузка данных в РФС) заметно медленее выполнения wordcount без фреймворка при 4 и менее дочерних узлах и практически не отличается при 5, 6 узлах.

По результатам тестирования выполнения задачи wordcount с помощью фреймворка можно сделать следующие выводы, которые можно распространить на произвольную задачу, выполняемую с помощью фреймворка:

- с увеличением количества узлов производительность выполнения задач с помощью фреймворка растёт и превосходит производительность выполнения на одной машине,
- время загрузки данных в РФС не зависит от количества узлов,

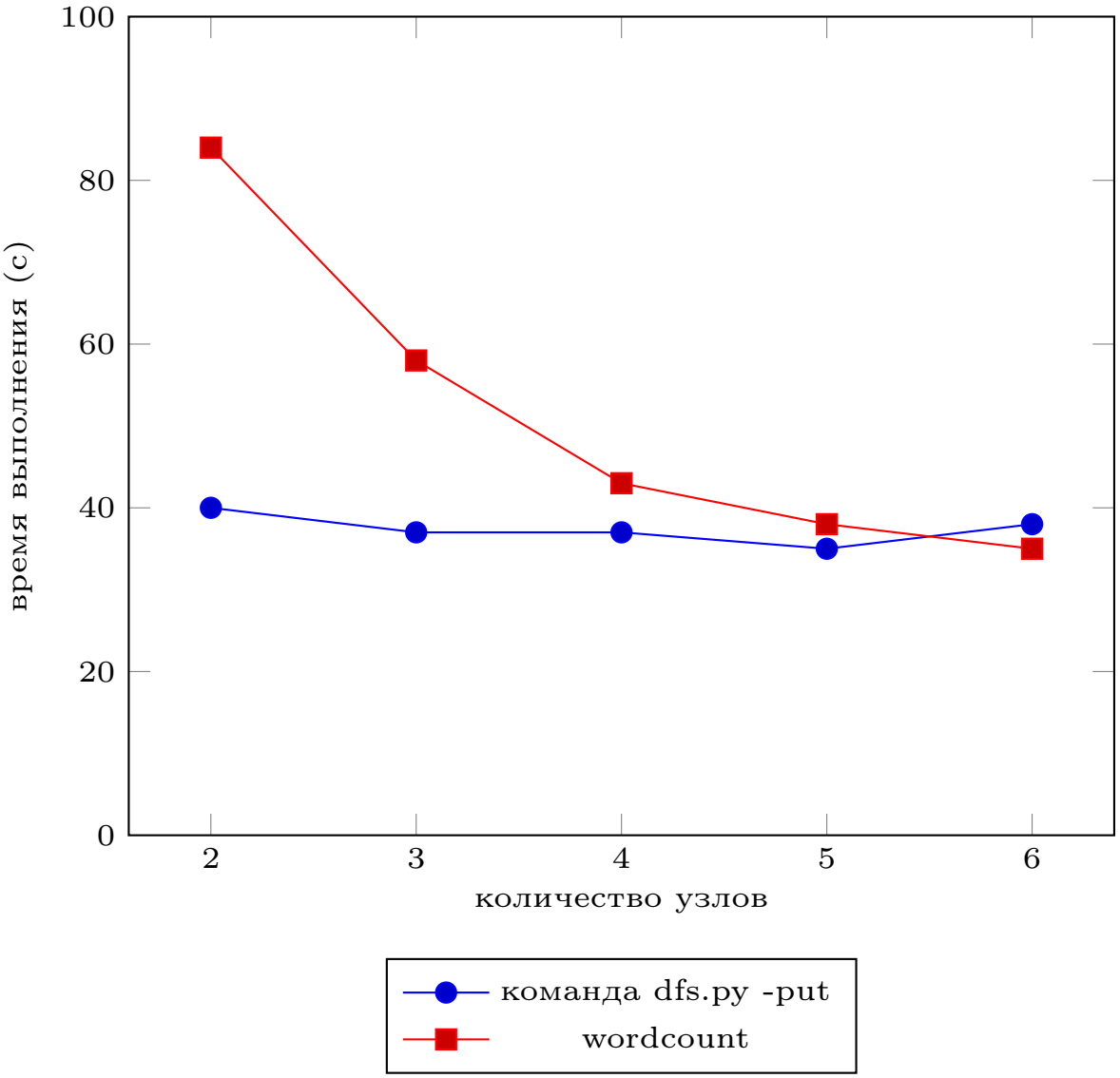

Рисунок 4 — Результаты тестирования задачи wordcount

- наиболее подходящей стратегий использования фреймворка является загрузка данных в РФС и их последующее использование в качестве входных данных для нескольких задач.

Тестирование эффективности решения задач с помощью фреймворка map-reduce производилось на задаче информационного поиска, описанной в главе 1.4.2. В качестве входных данных была использована сгенерированная таблица, содержащая по 400 целых чисел от 1 до 1000 в строке, объёмом 6 Гбайт. Запрос выполнялся с параметрами *const* равным 500 и i равным 1. Решение задачи с помощью map-reduce представлено в главе 3.4.4 и потребовало пару минут для его написания, решение задачи без использования фреймворка заняло заметно большее время (порядка часа)

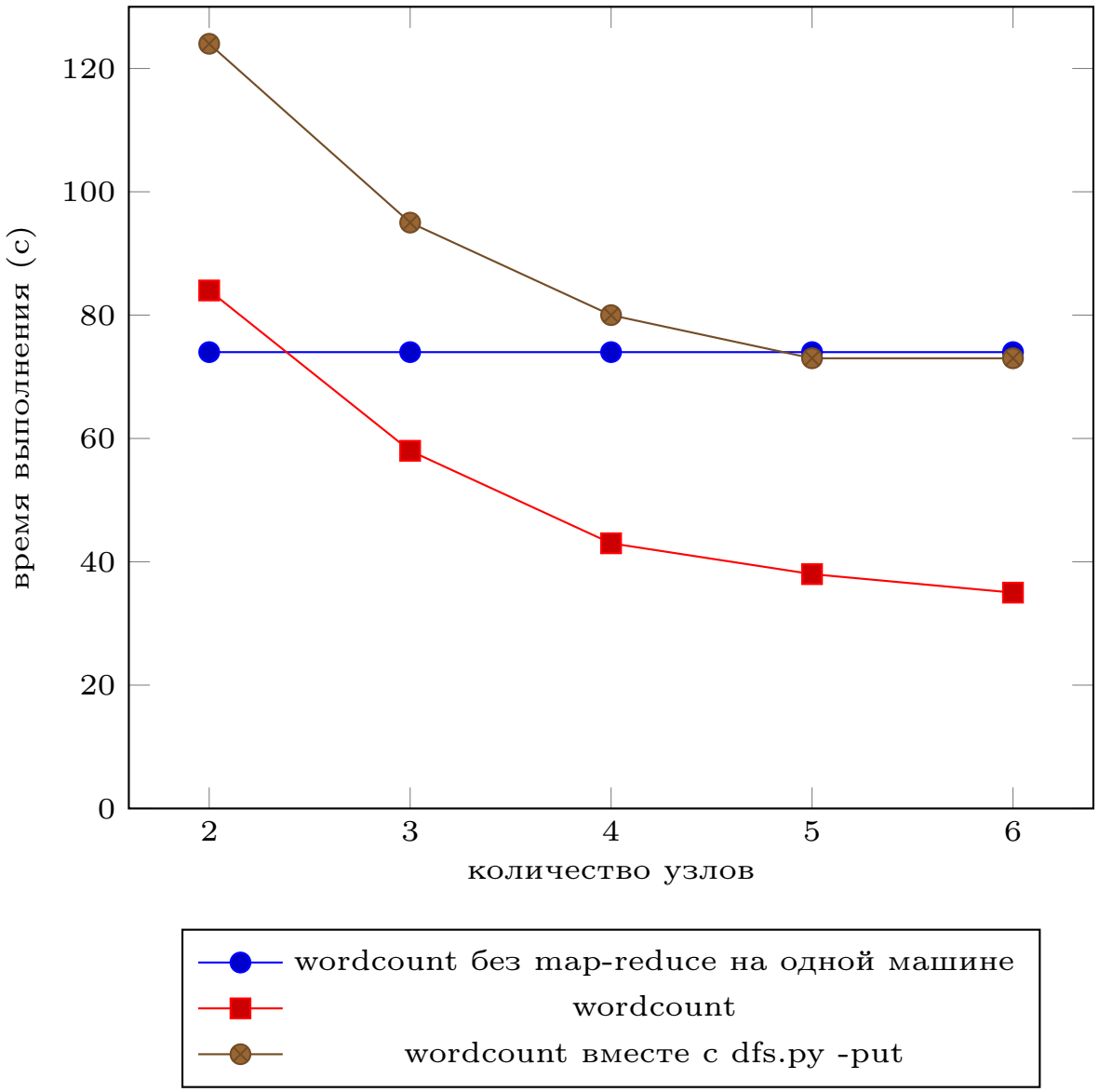

Рисунок 5 — Результаты тестирования задачи wordcount

и занимает 70 строчек кода на языке python.

По результатам тестирования были получены следующие результаты: время загрузки данных в РФС — 4мин. 30с., время выполнения задачи с помощью фреймворка map-reduce — 1мин. 54с., время выполнения задачи на одной машине без использования фреймворка — 6мин. 39с.. На основании результатов получены выводы:

- производительность решения задачи класса информационный поиск с помощью фреймворка map-reduce на кластере из 6 машин значительно превосходит производительность решения на одной машине без использования фреймворка.
- время, затраченное на написание решения данной задачи без фреймворка map-

reduce, намного превосходит время решения задачи с использованием фреймворка,

- map-reduce удобная абстракция для построения решения части задач класса информационный поиск.

5. Заключение

В рамках курсового проекта был разработан и реализован фреймворк для обработки больших данных в рамках концепции map-reduce. Также разработана и реализована распределённая файловая система с функционалом необходимым для полноценной работы фреймворка. Теоретически было найдено, что данный фреймворк оптимально подходит для решения задач класса информационный поиск, что было подтвержено в ходе тестирования.

Список литературы

[1] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters // Google, Inc. — 2004.

[2] MapReduce // slideshare: URL: http://www.slideshare.net/yandex/mapreduce-12321523

[3] What is the most efficient way to serialize in Python? // Quora: URL: http://www.quora.com/What-is-the-most-efficient-way-to-serialize-in-Python

[4] Hadoop save the World? // CODE1NSTINCT: URL: http://www.codeinstinct.pro/2012/08/hadoop-design.html

[5] MapReduce Patterns, Algorithms, and Use Cases // Highly Scalable Blog: URL: https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/