САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Статистическая оптимизация MapReduce

Дипломная работа студента 545 группы

Василинца Сергея Павловича

Научный руководитель		к. ф.-м. н., доцент
	/подпись/	Барашев Д.В.
Рецензент		старший преподаватель
	/подпись/	Луцив Д.В.
“Допустить к защите”		д.ф.-м.н., профессор
заведующий кафедрой,	/подпись/	Терехов А.Н.

Санкт-Петербург

2012

SAINT PETERSBURG STATE UNIVERSITY

Mathematics Mechanics Faculty

Software Engineering Chair

Statistical optimization of MapReduce

by

Sergei Vasilinetc

Supervisor		PhD, Associated Professor
	.....	D. V. Barashev
Reviewer		Assistant Professor
	.....	D. V. Luciv
“Approved by”		Professor
Head of Department,	.....	A. N. Terekhov

Saint Petersburg

2012

Содержание

1. Введение

В наше время компании собирают огромные объемы такой информации, как продажи их продукции, результаты рекламных кампаний в интернете, статистики использования различных функций продукта покупателем. Например, Facebook собирает 15 терабайт данных каждый день в свое петабайтное хранилище. Мощные телескопы в астрономии, ускорители частиц в физике, секвенсоры геномов в биологии генерируют большое количество данных, нуждающихся в дальнейшей обработке. Традиционные реляционные СУБД плохо справляются с такими объемами данных[1], которые к тому же часто не структурированы. В тоже время возможность совершать необходимые расчеты быстро и масштабируемо является необходимым условием для успешного исследования. Эффективная обработка огромных объемов является нетривиальной задачей, для решения которой в 2004 году компания Google разработала модель распределенных вычислений под названием MapReduce. MapReduce фреймворк предоставляет специальную программную модель и систему для обработки и генерации большого количества данных, которая применима для многих задач, возникающих в реальной жизни. Примеры таких задач рассматриваются в статье [2]. Программы, написанные для этой системы, автоматически распараллеливаются и исполняются на легкомасштабируемом кластере. Система также обрабатывает сбои машин в кластере, координирует сообщение между компьютерами внутри кластера, минимизируя нагрузку на сеть и хранилище данных. В связи с тем, что реализация Google является закрытой, другие большие компании, такие как Yahoo!, Facebook и Amazon активно разрабатывают свои реализации модели MapReduce или улучшают уже существующие.

1.1. Описание модели MapReduce

Прежде чем начать само описание модели MapReduce, хочется отметить, что оно несет исключительно ознакомительную функцию и содержит в себе большое количество упрощений. Как уже было сказано во введении, программы MapReduce используются для обработки большого количества данных, и потому обычно запускаются на кластерах из нескольких узлов. В основе модели лежат две функции Map и Reduce в целом аналогичные тем, которые часто используются в функциональном программировании [3]. Все MapReduce программы состоят из двух стадий:

1. На стадии Map входные данные разделяются на части и рассылаются на узлы. На каждом узле вызывается функция Map, результатом которой является список из пар: *(промежуточный ключ, промежуточное значение)*. Далее, полученные данные при помощи функции partition распределяются по машинам, которые будут выполнять последующую обработку. Чуть подробнее остановимся на функции partition, так как она будет во многом ключевой для данной работы. Partition является отображением множество промежуточных ключей во множество $[0, k-1]$, где k — количество машин. В классическом варианте MapReduce одинаковые промежуточные ключи должны попасть на одну и ту же машину для корректной обработки, поэтому эта функция не должна иметь побочных эффектов.

2. На стадии Reduce промежуточные данные на каждой машине сортируются и группируются по промежуточному ключу, после чего на них вызывается функция свертки (reduce). Функция свертки берет на вход промежуточный ключ и список всех промежуточных значений, сопоставленных ему, и возвращает некоторую пару.

Преимущество такой модели в том, что стадии Map и Reduce можно производить параллельно на множестве машин, что отсутствуют произвольные разделяемые данные и, как следствие, постоянное взаимодействие по сети, необходимое для синхронизации данных на узлах. Все это позволяет системе быть надежной и хорошо масштабируемоей. На рис 1 отображены основные стадии MapReduce программ:

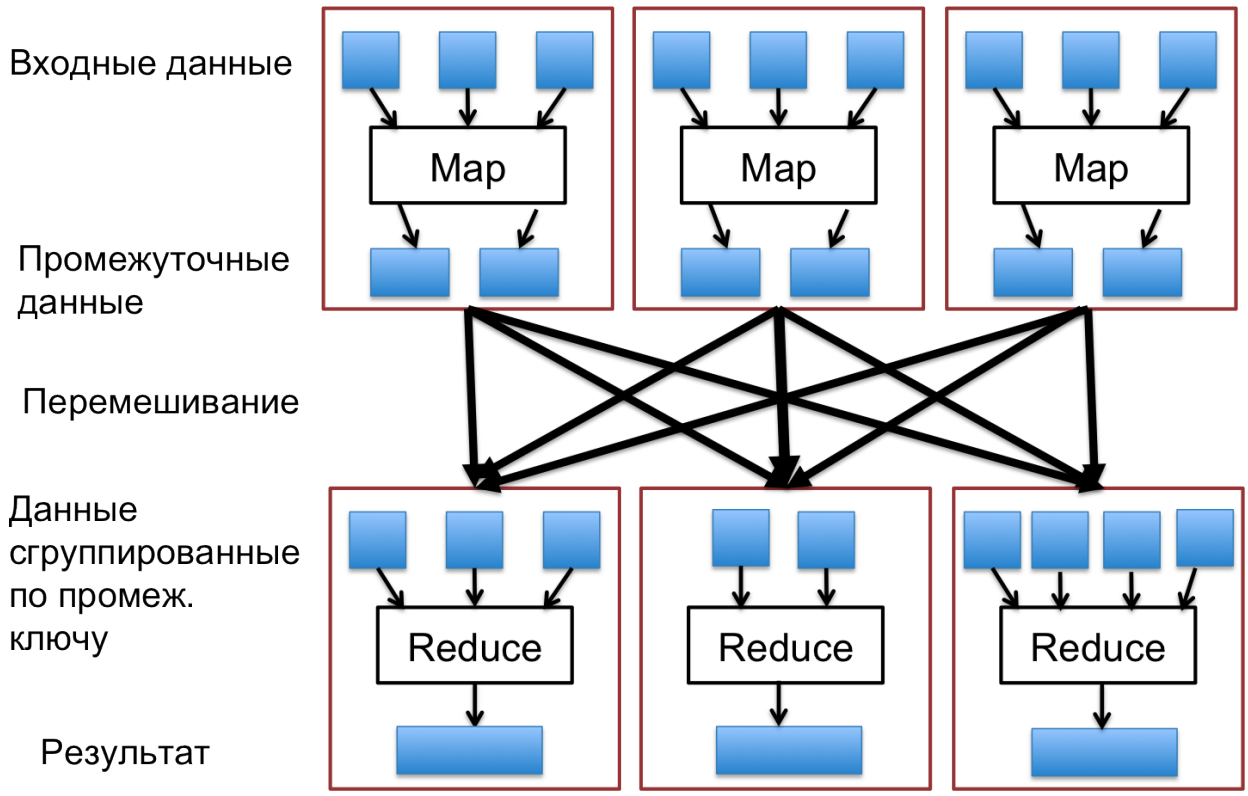

Рис. 1. Модель Map Reduce

1.2. Проблема

На некоторых MapReduce задачах можно наблюдать следующую ситуацию: одно или несколько Reduce заданий выполняются значительно дольше других, мешая стартовать последующей обработке данных. Логично предположить, что это может происходить из-за того, что объемы обрабатываемых данных на разных reducer-ах сильно различаются, в следствии неудачного распределения промежуточных ключей.

1.3. Предпосылки

Реальные MapReduce задачи выполняются с некоторой периодичностью изо дня в день на данных, которые слабо изменяются. В виду этого кажется возможным использование статистики распределения промежуточных ключей и значений, полученной во время предыдущих запусков, в целях оптимизации нагрузки на reducer-ы. Похожая идея используется в реляционных базах данных во время анализа стоимости плана запроса для предсказания размеров промежуточных данных[4].

2. Обзор

2.1. Существующие работы по оптимизации

В данной части очень сжато рассмотрим некоторые работы по оптимизации MapReduce программ. В большей части работ оптимизации SQL запросов адаптируются для MapReduce. Примером такой работы может быть работа [5], в которой представлен инструмент Manimal, решающий проблему нахождения в программах MapReduce операций проекции, выборки, агрегации, которые перемешены с другой программной логикой, и применяющий оптимизации для этих операций. В другой работе [6] рассмотрена оптимизация Join в задачах MapReduce. Стратегии, представленные в ней, хорошо работают в случае join-а одной очень большой таблицы с несколькими маленькими.

2.2. Открытая реализация MapReduce. Проект Hadoop

Наиболее распространенной реализацией модели MapReduce является Hadoop™ Project[7], основной вклад в который сделали разработчики из компаний Facebook и Yahoo! Hadoop был успешно внедрён в этих компаниях [8]. Также этот проект официально поддерживается на Amazon EC2, где доступны официальные AMI (Amazon Machine Images), которые сразу содержат в себе Hadoop. Кроме того, в дистрибуционном пакете Hadoop находятся инструменты, которые позволяют сравнительно легко развернуть кластер с Hadoop на Amazon EC2 и запускать MapReduce программы. К сожалению, эти инструменты и AMI на данный момент давно не обновлялись и сильно устарели. Об этом подробнее будет сказано в части 5.1.

В виду широкой поддержки этого проекта и возможности развернуть кластер на Amazon EC2 было решено проводить данное исследование на

его основе. Рассмотрим подпроекты, составляющие его:

1. Hadoop Common: общие утилитарные классы, необходимые для остальных подпроектов Hadoop.

2. Hadoop Distributed File System (HDFS^{TM}): распределенная файловая система [9] с высокой пропускной способностью.

3. Hadoop MapReduce: фреймворк для написания и исполнения программ MapReduce на вычислительных кластерах.

Наибольший интерес для нас представляет последний подпроект : Hadoop MapReduce. Опишем ключевые компоненты, входящие в него, и которые будут использоваться в дальнейшем в данной работе. Наиболее важный класс для данного исследования - это Partitioner, абстрактный класс, который отвечает за распределение ключей, полученных на стадии Map, по Reduce машинам. Он имеет всего лишь один метод с сигнатурой:

```
abstract class Partitioner<KEY, VAL>{
    abstract int getPartition(KEY key, VAL value, int k);
}
```

Этот метод определяет, на какую из k машин будет отправлена пара (key, value) для свертки. Реализацией по умолчанию этого класса является класс HashPartitioner, в котором метод getPartition берет хэш код ключа key по модулю k. Стоит отметить, что эта реализация не является специфичной для проекта Hadoop, она упоминалась еще в самой первой публикации о парадигме MapReduce[2].

Двумя другими основополагающими классами являются абстрактные классы Mapper и Reducer. Классы, наследующиеся от Mapper, реализуют пользовательскую функцию map, аналогично классы наследники Reducer реализуют пользовательскую функцию reduce.

3. Постановка задачи

Целью данной работы является исследование применимости и актуальности статистической оптимизации в задачах MapReduce. Для этого необходимо:

1. Реализовать сбор статистики распределения промежуточных значений по промежуточным ключам.

2. Реализовать алгоритм, обеспечивающий равномерную нагрузку на Reduce машины на основе собранной статистики.

3. Реализовать инструментарий для сравнения распределений промежуточных ключей по reduce-машинам.

4. Провести тестирование и сравнительный анализ оптимизированной версии на реальных задачах.

4. Решение

4.1. Алгоритм

Допустим, что мы умеем собирать статистику по промежуточным ключам и на основе ее предсказывать количество промежуточных значений, соответствующих каждому промежуточному ключу или группе ключей. При помощи этих данных надо загрузить reduce машины так, чтобы reduce стадия на всех них занимала одинаковое время. Будем исходить из естественного предположения, что время исполнения reduce задания на одной машине пропорционально суммарному количеству промежуточных значений по всем промежуточным ключам, обрабатываемых на данной машине. Таким образом, хочется добиться того, что это количество на всех машинах было одинаковым. Если пронумеровать ключи, то мы можем получить массив, в котором на i-ой позиции стоит количество промежуточных значений соответствующих i-ому промежуточному ключу, и этот массив требуется разбить на k как можно более близких по сумме множеств.

Математическая постановка задачи

Дан массив положительных чисел $a_1, \ldots, a_n$ и его надо разбить на k множеств $M_1, \ldots, M_k$ так, чтобы разница $\max(S_1, \ldots, S_k) - \min(S_1, \ldots, S_k)$, была минимальной, где $S_k = \sum_{a \in M_k} a$.

Это один из частных случаев известной K-balance partition problem, которая в жизни встречается в разных формулировках таких как: распределение сетевых запросов к серверам или задач по процессорам; она является NP-полной [10]. По решению этой и аналогичных ей задач с различными ограничениями на входные данные существует множество исследовательских работ [11, 12], целью же данной работы не являлась разработка или улучшение существующих подходов. В текущей реализации использовался простейший алгоритм, аналогичный алгоритму First Fit Descending[13],

решающий "задачу об упаковке в контейнеры".

Шаги этого алгоритма:

1. Отсортируем массив a по убыванию.

2. На i-ой итерации берется элемент a_i и кладется в множество с минимальной суммой на данный момент.

4.2. Архитектура

В предыдущем разделе было сделано допущение, что мы умеем собирать статистику распределения промежуточных значений по промежуточным ключам. Этот раздел будет посвящен следующим вопросам: как же ее можно собирать, какой из вариантов сбора был реализован и какие точки расширения есть в данной реализации.

Итак, первым и основополагающим вопросом, на которые надо ответить: какие возможности предоставляет API Hadoop для сбора статистики. Данные, необходимые для статистики, доступны с конца Map стадии и до начала стадии Reduce. Рассмотрим, где можно в этом промежутке добавить сбор статистики:

1. Конец стадии Map, на основе класса *Mapper*. Первым путем является попытка перехвата пары *(промежуточный ключ, промежуточное значение)* во время их записи в экземпляр класса *Mapper.Context*. Для этого потребовалось бы создать класс, проксирующий класс *Mapper.Context*, за тем исключением, что метод *write (KOUT key, VOUT value)* еще бы и обновлял статистику. Следующим шагом было бы создание нового класса, наследующего *Mapper* и передающего в метод *map(KEYIN key, VALUEIN value, Mapper.Context context)*, не исходный context, а его вышеописанный прокси. Этот подход имеет тот

минус, что для того, чтобы опробовать нашу оптимизацию на уже существующих программах, необходимо вносить изменения в код, а именно менять родительский класс для *Mapper*.

2. Конец стадии Map, на основе класса *Partitioner*. Это второй путь, позволяющий собрать статистику в конце Map стадии. На этом пути создается абстрактный класс, наследующийся от *Partitioner*, который перед вызовом реальным вызовом *getPartition(KEY key, VALUE value, int k)* будет обновлять статистику. К сожалению, этот метод также содержит свои подводные камни: *Partitioner* не имеет методов жизненного цикла. Это может быть плохо, если статистика будет сохраняться просто в файл, так как нам необходимо узнавать, когда *Partitioner* закончил свою работу, и можно выполнить запись собранной статистики в файловую систему.

3. Начало стадии Reduce, на основе класса *Reducer*. Последний вариант - это обновлять статистику в наследнике класса *Reducer* перед тем, как вызывать функцию reduce. Рассмотрим его минусы: во-первых, этот вариант имеет тот же самый минус, что и первый вариант, во-вторых, дополнительное итерирование по всем промежуточным значениям крайне опасно, так как в случае их большого количества будет выполнено много дисковых операций, что может свести на нет попытки оптимизации.

Для облегчения дальнейшего использования данной разработки было решено реализовать сбор статистики на основе *Partitioner*. (Для его использования необходимо прописать новый *Partitioner* в конфигурационных файлах, что пришлось бы делать в любом случае, так как мы заменяем и сам алгоритм распределения ключей). Следующим этапом после сбора

статистики является ее хранение, и на этом этапе тоже есть два альтернативных пути:

1. хранить статистику в файловой системе;

2. хранить статистику в базе данных;

В виду того, что второй путь гораздо более тяжеловесен, так как тянет в проект новые зависимости и требует дополнительно разворачивания сервера базы данных в кластере, и не имеет столь же очевидных и весомых плюсов, было решено пойти первым и простейшим путем.

После того, как сделан выбор на основе каких компонентов будет собираться статистика, и где она будет храниться, обратим внимание на саму статистику. Очевидным и неприятным фактом является то, что хранить ее в виде пары промежуточного ключа и количества соответствующих ему промежуточных значений невозможно, в ввиду того, что MapReduce обрабатывает огромные объемы данных, и как следствие:

1. промежуточных ключей может быть очень много;

2. сами ключи могут быть тяжелыми, то есть занимать много места на диске. Например, промежуточными ключами могут быть длинные строки;

Из-за этой проблемы приходится объединять ключи в группы, и распределять уже эти группы ключей по reduce машинам. И переходя к программной реализации данной работы, эта необходимость была учтена в интерфейсе Statistic, который должны реализовать классы, собирающие и обрабатывающие статистику.

```
public interface Statistic<K, V, M> extends Writable {
    void add(K key, V value);
    Converter<K, M> getConverter();
    Map<M, Integer> export();
}

public interface Converter<K, M> {
    M convert(K key);
}
```

Statistic<K, V, M> правильно читать следующим образом: статистика принимающая ключи класса K, значения класса V, и объединяющая ключи в группы класса M. Ниже будет описана стандартная реализация этого интерфейса, но сначала кратко опишем, зачем нужны те или иные его методы. Заметим, что этот интерфейс наследует интерфейс $Writable$ пакета *org.apache.hadoop.io*, который содержит методы, необходимые для сериализации объектов этого класса при сохранении его на диск или передачи по сети. Вернемся к методам интерфейса *Statistic*:

1. *add* — добавляет в статистику *key* и *value*.

2. *getConverter* — возвращает функцию, которая сопоставляет ключу группу, в которую будет входить ключ. Эта функция должна для одного и того же ключа возвращать одну и ту же группу, так в противном случае пары с одинаковым промежуточным ключом могут оказаться на разных reduce машинах. Так же надо иметь в виду, что эта функция будет вызвана для каждой пары *(промежуточный ключ, промежуточное значение)*.

3. *export* — возвращает Map, в которой каждой группе ключей сопоставлено ожидаемое количество промежуточных значений, сопоставленных всем промежуточных ключам в этой группе. На основе этих

данных и исполняется алгоритм из части 4.1.

Класс HashStatistic является реализацией по умолчанию интерфейса Statistic. Он группирует ключи по хеш-коду взятому по модулю некоторого числа N, которое сильно превосходит количество машин.

На основе выше описанного алгоритма и интерфейса статистики, создан абстрактный класс StatistiscalPartitioner, который имеет один абстрактный метод:

```
abstract Statistic<KEY, VALUE, M> newStatistic();
```

Класс StatisticalPartitioner при помощи этого метода считывает статистику с предыдущих запусков и на ее основе распределяет промежуточные ключи. HStatisticalPartitioner является наследников StatisticalPartitioner, использующим класс HashStatistic, как реализацию интерфейса Statistic.

HStatisticalPartitioner является простейшим примером Partitioner, который работает на основе собранной статистики распределения промежуточных значений. Даже такой наивной реализации достаточно для проверки применимости статистической оптимизации в парадигме MapReduce.

5. Эксперименты

5.1. Окружение эксперимента

Как было уже сказано в части 2.2 Amazon EC2 официально поддерживает Hadoop Project и предоставляет официальные публичные AMI для развертывания кластера, в виду чего было решено проводить эксперименты на этой площадке. Кроме того, на Amazon доступны публичные наборы данных (Public Data Sets) из различных научных областей таких как бионформатика, алгебра, астрономия, обработка естественных языков[14]. Чтобы начать пользоваться этими данными надо лишь выполнить пару команд. Эти наборы снимают с исследователей задачу поиска и составления тестовых наборов для своих экспериментов. Один из них - Freebase Simple Topic Dump (снапшот с номером snap-46ca9b28) будет активно использоваться в экспериментах, описанных в 5.2 и 5.3.

Кластер во время проведения всех экспериментов состоял из машин типа Small Instance, которые имеют 1.7 гигабайт оперативной памяти и один EC2 compute unit, который эквивалентен процессору 1.0 - 1.20 GHz 2007 opteron или процессору 2007 Xeon.

В части 2.2 было упомянуто, что наша реализация основана на API Hadoop версии 0.20.203. К сожалению, официальные образы Amazon содержали только устаревшую версию 0.18.0, потому была использована публичная, но не официальная AMI с номером ami-975390fe, которая содержала Hadoop с API 0.20.203. Также скрипты в дистрибуционном пакете, запускающие и конфигурирующие кластер на Amazon EC2, ориентированы на старое API. В коде, прилагающемся к данной работе, содержатся исправленные скрипты, записывающие конфигурацию кластера соответствующую API 0.20.203.

5.2. Word count

Классическим и одновременно простейшим примером MapReduce программы является Word count, задачей которой является подсчет количества вхождения слова в данный текст. Эта задача возникла как часть программ, считающих TF-IDF[15] — меры важности слова в документе, являющегося частью набора документов.

Кратко опишем работу этой программы: на стадии Map на вход приходит пара из номера строчки в тексте и самой строчки, функция map разбивает строчку на слова и выдает промежуточные пары вида *(слово, 1)*. На стадии reduce промежуточные пары группируются по слову и все единицы суммируются, и результатом является пара *(слово, количество его вхождений в текст)*.

Дальнейшие результаты приведены на тестах, где входными данными были 5000000 последних строчек из Freebase Simple Topic Dump, которые суммарно занимали 319 мегабайт, программы исполнялись на машинах в вышеописанной конфигурации. Для сбора статистики оптимизированная версия сначала запускалась на первых 1500000 строчках Freebase Simple Topic Dump.

Прежде, чем переходить к результатам эксперимента, опишем два типа гистограмм, которые будут встречаться в этом и дальнейших экспериментах:

1. «Количество промежуточных значений», где на оси Y отображается количество промежуточных значений, а по оси X reducer-ы. Этот тип сопоставляет каждой reduce машине количество промежуточных значений, которых она обработала. Reducer-ы на гистограмме отсортированы по убыванию количества обработанных значений.

2. «Время», где по оси Y — секунды, по оси X — reducer-ы. Этот тип

отображает сколько секунд исполнялось reduce задание на машине. Reducer-ы на гистограмме отсортированы по убыванию времени исполнения reduce задания.

Как можно заметить из гистограммы на рис. 2, оптимизированная версия действительно лучше распределила промежуточные ключи по reduce машинам: количество обрабатываемых промежуточных значений на каждой из машин примерно равно, в то время как количество обрабатываемых промежуточных значений в стандартной реализации колеблется гораздо сильнее, и максимальное количество превосходит минимальное более чем в 2 раза. Но из гистограммы на рис. 3 видно, что на итоговом времени исполнении стадии Reduce, и, как следствие, MapReduce программ в целом, это не сказывается. Такие результаты можно объяснить тем, что на стадии Reduce в Word count выполняется лишь сложение, то есть очень простая операция. Наша же оптимизация должна работать, где на стадии Reduce выполняется множество операций. Классический Word Count является примером, где наша оптимизация не нужна.

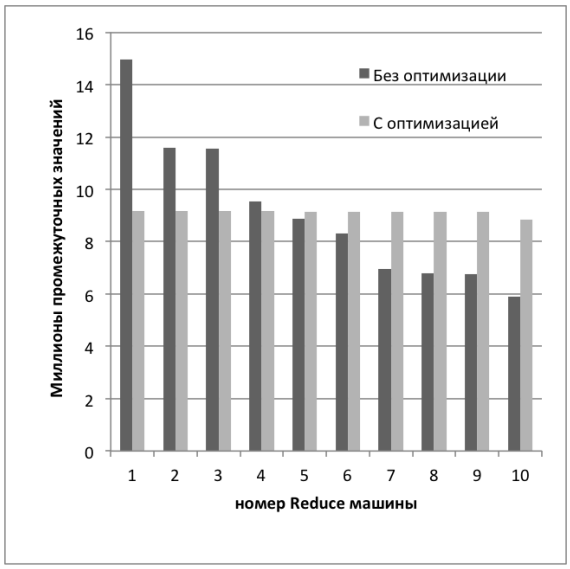

Рис. 2. Количество промежуточных значений в эксперименте с 10-ю задачами reduce

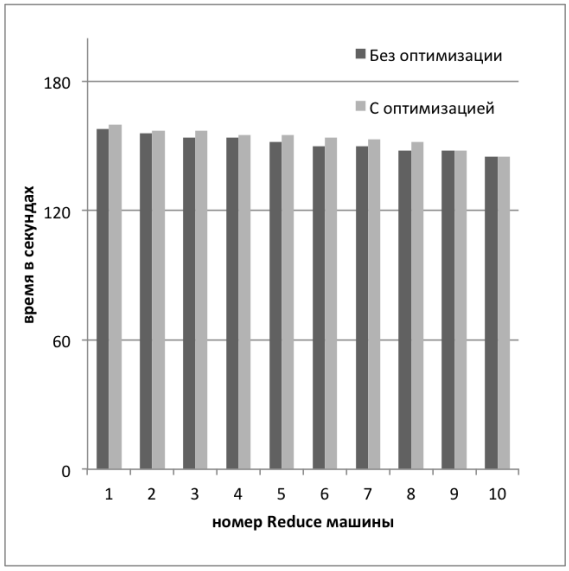

Рис. 3. Время в эксперименте с 10-ю задачами reduce

5.3. First Character

В следующем эксперименте производится подсчет количества слов, начинающихся с каждой из букв. Программа была искусственно замедлена так, что каждые 500000 промежуточных ключей обрабатываются заданные пользователем t миллисекунд. Интерес этого эксперимента для нас состоит в том, что, во-первых, слова, начинающиеся с разных букв алфавита, употребляются с разной частотой: очевидно, что слова, начинающиеся с буквы 'x', гораздо реже употребляются, чем слова, начинающиеся с 's'. Таким образом стандартный Partitioner должен сработать довольно плохо, и количество обрабатываемых промежуточных значений на разных машинах должно сильно отличаться. И при увеличении t эта разница должна значительно отразиться на времени исполнения соответствующих Reduce заданий.

Эксперименты проводились со следующими параметрами: программы обрабатывали первые 1500000 строчек из Freebase Simple Topic Dump, вер-

сия со статистической оптимизацией, перед этим запускалась и собирала статистику с последних 500000 строчек того же дампа.

При малых t программа ожидаемо введет себя идентично Word Count, поэтому диаграммы с малым t не представляют никакого интереса для нас и не будут приведены в данный работе. Рассмотрим диаграммы на рис. 4, 5, 6, 7, соответствующие запускам эксперимента при $t = 30000$ на восьми и десяти reduce машинах, соответственно. На этих диаграммах видно, что промежуточные ключи, как и в случае Word Count, стандартный Partitioner распределил неудачно, но в отличие от предыдущего случая это отразилось на времени исполнения стадии Reduce. Как результат версия с оптимизацией, распределяющая промежуточные ключи на основе статистики, выигрывает почти в два раза. А общее время исполнения программ:

1. в случае восьми reducer-ов, без оптимизации равно 12 минут 46 секунд, с оптимизацией — 8 минут 5 секунд;

2. в случае десяти reducer-ов, без оптимизации — 11 минут 40 секунд, с оптимизацией — 6 минут 40 секунд;

Также отметим, что при увеличении количества Reducer-ов общее время исполнения программ ожидаемо снизилось.

На данном искусственном тесте наша оптимизация дала очень хорошие результаты и выигрыш, относительно не оптимизированной версии, чуть менее 50%. Из этого теста можно сделать вывод, что в случае неравномерно распределенных промежуточных ключей и долго исполняющейся стадии Reduce и в более жизненных MapReduce программах можно ожидать улучшения от данной оптимизации.

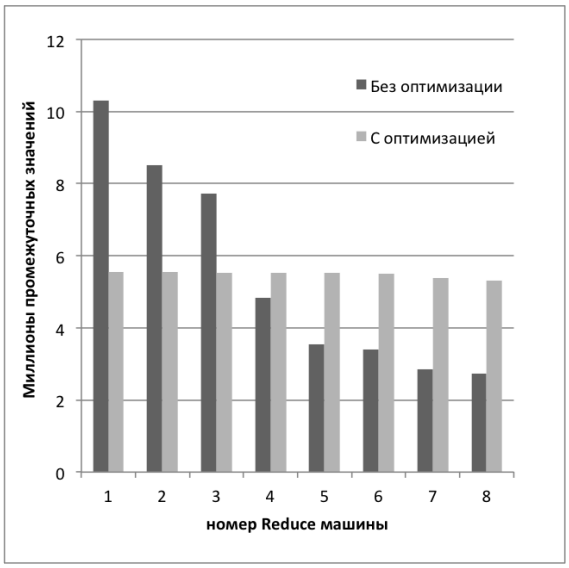

Рис. 4. Количество промежуточных значений в эксперименте с 8-ю задачами reduce

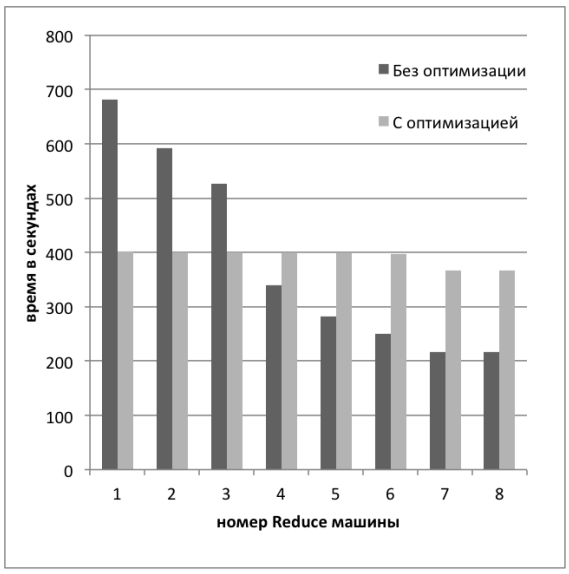

Рис. 5. Время в эксперименте с 8-ю задачами reduce

5.4. Средняя длина сессии

В последнем эксперименте будет рассмотрено одно из классических применений MapReduce технологии — задача анализа логов. В нашем случае будет проведен анализ географии и поведения пользователей сайта, а точнее нас будет интересовать следующая статистика: сколько в среднем страниц за одну сессию просматривают пользователи каждой из стран, то есть пользователи из России, пользователи из Италии и так далее.

Кратко опишем подход к ее решению: на вход в программу приходят логи запросов к сайту за некоторый период времени, на стадии Map каждому запросу сопоставляется страна пользователя при помощи ip адреса, находящегося в записи лога. Таким образом результатом стадии Map является пара *(страна, запрос)*. На стадии Reduce запросы, сгруппированные по странам, разбиваются на сессии, и подсчитывается отношение запросов страниц к количеству сессий, которое и является искомым ответом. Более подробное описание реализации, представленной в данной работе, будет

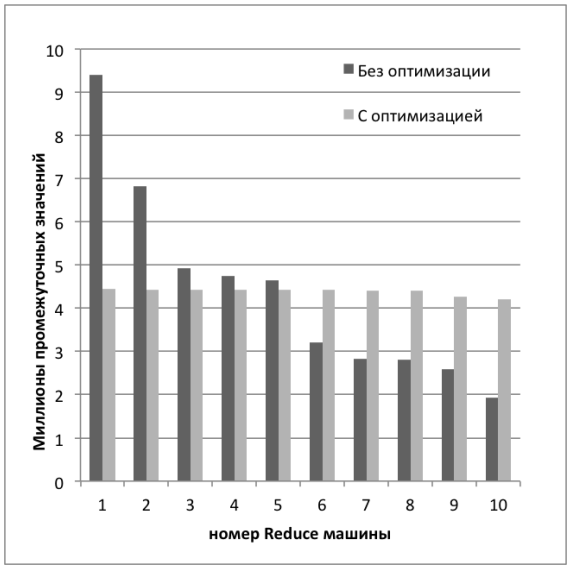

Рис. 6. Количество промежуточных значений в эксперименте с 10-ю задачами reduce

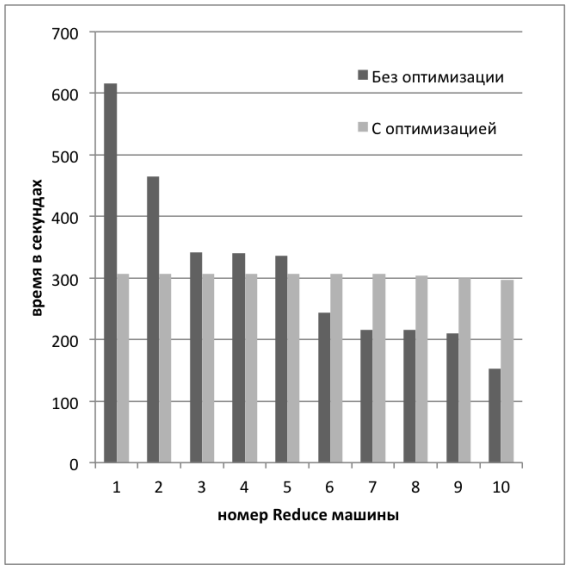

Рис. 7. Время в эксперименте с 10-ю задачами reduce

приведено чуть позже в данном разделе.

Интерес данной задачи для нашего исследования состоит в том, что:

1. она взята из жизни;
2. количество промежуточных значений у разных промежуточных ключей должно сильно разниться;
3. как видно, даже из краткого описания, на стадии Reduce происходит большое количество действий;

Как следствие, в этой задаче наша оптимизация имеет шансы на успех.

Итак, вернемся к реализации данного эксперимента. Она в большой степени зависит от того, в каком формате и какая информация содержится во входных данных. Реализации в данной работе обрабатывает логи с сайта проекта GanttProject[16], приходящие в формате Combined Logs для Apache Server 1.3. Эти логи не содержат cookies для легкого определения сессии запросов к серверу, поэтому сессия определяется при помощи IP

адреса, User Agent и времени запроса. Сессии считаются разными, если между двумя запросами с одним и тем же IP адресом и User Agent прошло полчаса. Для того, чтобы эффективно определять принадлежат ли запросы одной сессии, они должны быть отсортированы по времени к началу стадии Reduce. Этого можно достичь, сделав время вторичным ключом и и отсортировав сгруппированные промежуточные результаты по значению вторичного ключа. Такой прием называется *Secondary Sort*. Его наличие в данной реализации добавляет ценности этому эксперименту, так как для использования Secondary Sort в Hadoop требуется изменить поведение Partitioner, и в теории наложение двух изменений в Partitioner могло бы привести к проблемам. Но как будет далее показано в этой части, использовать StatisticalPartitioner возможно и для Secondary Sort, и это использование не требует большого количества усилий. Но сначала опишем как реализовывается Secondary Sort в Hadoop.

Secondary Sort

Итак, повторим желаемый результат: на стадию Reduce данные должны приходить с группированные по первичному ключу и отсортированные по вторичному. К сожалению, в Hadoop нет легкого механизма реализации Secondary Sort. Кратко опишем сложный:

1. Создать класс, который будет промежуточным ключом, и который содержит в себе первичный и вторичный ключи. Также он должен реализовывать интерфейс WritableComparable и сравнивать промежуточные ключи сначала по первичным, потом вторичным ключам.

2. Создать Partitioner, который распределяет промежуточные ключи, основываясь только на первичных.

3. Создать RawComparator, сравнивающий промежуточные ключи, основываясь только на первичных. Далее этот класс надо выставить

как GroupingComparator для этой MapReduce задачи, что можно сделать при помощи соответствующего set метода у объекта класса Job.

Рассмотрим второй пункт чуть более подробнее. Пусть класс промежуточного ключа называется SecSortKey, а класс partitioner'a — SecSortPart. Он должен выглядеть примерно следующим образом:

```
class SecSortPart<V> extends Partitioner<SecSortKey, V>{
...
}
```

К сожалению, для применения нашей оптимизации было бы неправильно просто заменить Partitioner на HStatisticalPartitioner, так как таким образом собиралась бы статистика целиком по промежуточному ключу, в то время как интересна статистика только по первичному. Прием, который будет применен, похож на рефакторинг "Замена наследования делегированием"[17]: создадим поле, которое будет ссылаться экземпляр HStatisticalPartitioner. Этот экземпляр будет собирать статистику только по первичному ключу и распределять по reduce машинам, тоже, только по первичному ключу. Таким образом получается следующий код:

```
class SecSortPart<V> extends Partitioner<SecSortKey, V>{
   private HStatisticalPartitioner<PK, V> hPart = new ...

   public int getPartition(SecSortKey sKey, V vak){
       hpart.getPartition(sKey.getPk(), val);
   }
}
```

Подытоживая выше сказанное, эта техника дает возможность использовать статистическую оптимизация вместе с Secondary Sort, и она была успешна опробована в данном эксперименте.

Эксперименты

Наконец, перейдем к экспериментам. Эксперименты проводились на

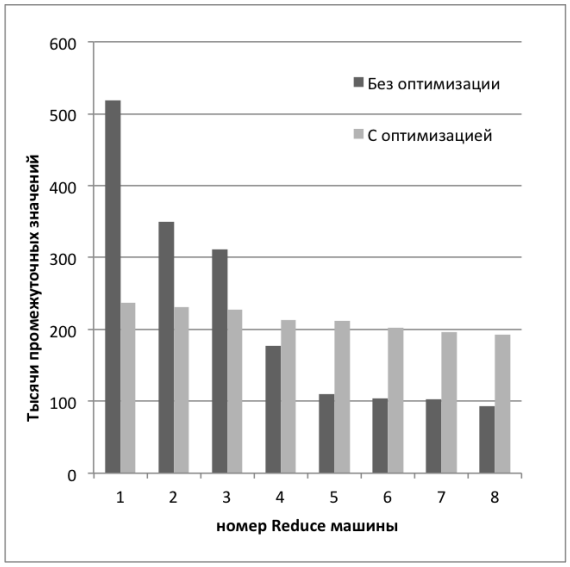

Рис. 8. Количество промежуточных значений в эксперименте с 8-ю задачами reduce

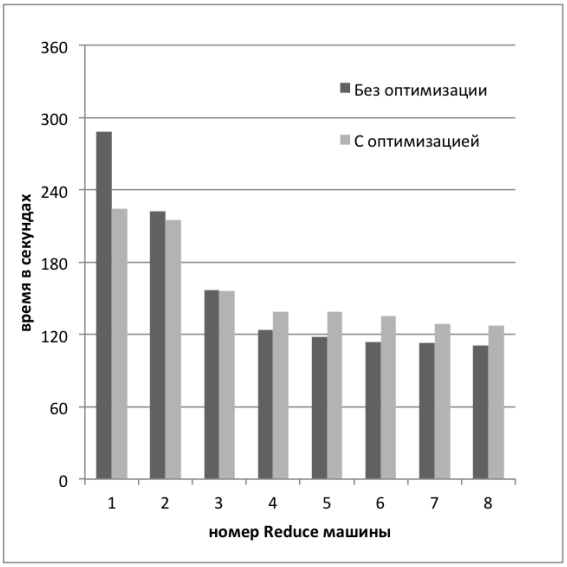

Рис. 9. Время в эксперименте с 8-ю задачами reduce

логах запросов к сайту проекта GanttProject за март 2012 года. Версия с оптимизацией перед запусками имела собранную статистику за первые 9 дней января 2012 года. Рассмотрим гистограммы, изображенные на рис. 9 и рис. 11. На них очевиден выигрыш оптимизированной версии, разница между самыми длинными reduce заданиями в оптимизированной и не оптимизированной версиями более минуты, при общей длине в 5 минут, то есть выигрыш около 20%. Но в этих результатах можно заметить еще интересные факты. Например, в неоптимизированной версии стадия Reduce на 10 машинах исполнялась дольше, чем на 8. Если взглянуть на гистограммы на рис. 8 и рис. 10, картина проясняется: для 10 reducer стандартный Partitioner распределил ключи очень неудачно, на одном из reducer оказалось сильно больше промежуточных значений для обработки. Еще один интересный факт: не произошло улучшения скорости исполнения стадии Reduce между запусками на 8 и 10 reducer-ах и в оптимизированной версии. Изучение результатов запусков показало, что самое продолжительное

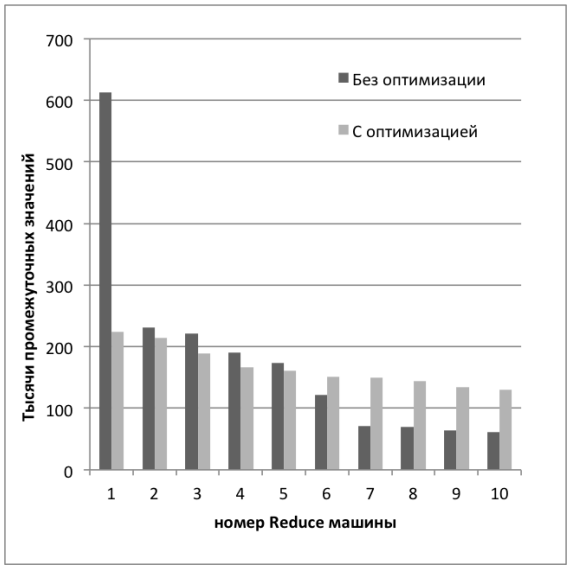

Рис. 10. Количество промежуточных значений в эксперименте с 10-ю задачами reduce

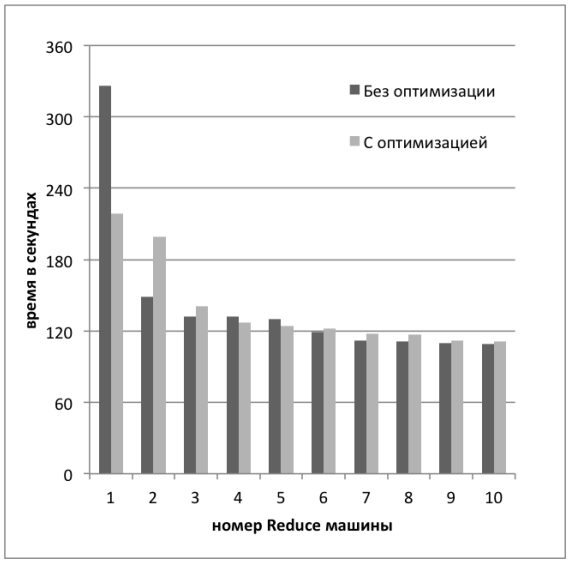

Рис. 11. Время в эксперименте с 10-ю задачами reduce

reduce задание обрабатывало одно промежуточные значения одного промежуточного ключа — США. И такое поведение нашей оптимизации неподвластно.

6. Заключение

Для параллельной обработки большого количества неструктурированных данных часто используют модель вычислений MapReduce. Одним из ее недостатков является неравномерная нагрузка на reducer-ы из-за неудачного распределения промежуточных ключей. В данном исследовании рассматривалась оптимизация, распределяющая промежуточные ключи на основании данных, полученных на предыдущих запусках этой MapReduce программы. Эта оптимизация была опробована на некоторых жизненных задачах, и часть из этих задач была существенно ускорена. Статистическая оптимизация показала хорошие результаты в случае неравномерного распределения промежуточных значений по промежуточным ключам и Reduce стадии, выполняющей большое количество операций. В ходе данной работы:

1. Реализован сбор статистики распределения промежуточных значений по промежуточным ключам.

2. Реализован алгоритм, обеспечивающий равномерную нагрузку на машины, исполняющие Reduce задания, на основе собранной статистики.

3. Реализованы инструменты для сравнения распределений промежуточных ключей по reduce машинам.

4. Проведено тестирование и сравнительный анализ оптимизированной версии на задачах «Word Count», «First Character», «Средняя длина сессии».

6.1. Выводы

Целью данной работы было проверить применимость идеи статистической оптимизации для MapReduce программ. Как видно из раздела 5 статистическая оптимизация работает на части примеров, из которых можно сделать следующий вывод: для MapReduce программ с легкой стадией Reduce, на которой производится мало операций, например, когда функция Reduce просто суммирует переданные значения или является ID отображением, результаты нашей оптимизации не заметны. Такое поведение легко объяснимо и ожидаемо, так как наша оптимизация направлена на стадию Reduce. В примерах же, где на стадия Reduce выполняется достаточно долго, как в примере описанном в 5.4, и сами промежуточные значения распределены не равномерно по ключам, можно ожидать заметного улучшения скорости исполнения MapReduce программ.

6.2. Развитие работы

Данная работа рассматривала возможность создания статистической оптимизации в программах MapReduce. Как было показано, на некоторых примерах она может приносить заметные и хорошие результаты, даже при самой наивной реализации, представленной в данной работе. Дальнейшие развитие можно проводить по следующим направлениям:

1. Улучшение методов сбора статистики. В этой области, во-первых, можно улучшить функцию группировки промежуточных ключей, во-вторых, можно добавить выделение тренда изменений во входных данных, для более точного реагирования на изменения данных со временем.

2. Реализация более совершенного алгоритма равномерного распределения ключей по reduce машинам.

3. Проведение экспериментов на промышленных кластерах из 1000 узлов и данных размером терабайт и более. К сожалению, в ходе данной работы не было финансовой возможности провести такое масштабное исследование.

4. Проведение дополнительных экспериментов на примерах использования MapReduce программ в Data mining.

5. Приведение код к промышленным стандартам и стандартам проекта Hadoop Apache, и предложение внесения кода в проект Hadoop.

Литература

1. Stonebraker M., Abadi D., DeWitt D. J. et al. MapReduce and parallel DBMSs: friends or foes? // Commun. ACM. 2010. — . Vol. 53, no. 1. Pp. 64–71. URL: http://doi.acm.org/10.1145/1629175.1629197.

2. Dean J., Ghemawat S. Mapreduce: Simplified data processing on large clusters // OSDI. 2004. Pp. 137–150.

3. MapReduce. URL: http://en.wikipedia.org/wiki/Partition_problem (дата обращения: 20.05.2012).

4. Managing Optimizer Statistics. URL: http://docs.oracle.com/cd/B13789_01/server.101/b10752/stats.htm (дата обращения: 21.05.2012).

5. Jahani E., Cafarella M. J., Ré C. Automatic Optimization for MapReduce Programs // CoRR. 2011. Vol. abs/1104.3217.

6. Afrati F. N., Ullman J. D. Optimizing Joins in a Map-Reduce Environment: Technical report: National Technical University of Athens, Stanford University, 2009. — December. URL: http://ilpubs.stanford.edu:8090/952/.

7. Hadoop Project. URL: http://hadoop.apache.org/ (дата обращения: 20.05.2012).

8. Vance A. Hadoop, a Free Software Program, Finds Uses Beyond Search. URL: http://www.nytimes.com/2009/03/17/technology/business-computing/17cloud.html?_r=1 (дата обращения: 17.03.2009).

9. Distributed File System. URL: http://ru.wikipedia.org/wiki/Distributed_File_System (дата обращения: 21.05.2012).

10. Partition problem. URL: http://en.wikipedia.org/wiki/Partition_problem (дата обращения: 01.05.2012).

11. Korf R. E. A complete anytime algorithm for number partitioning // Artificial Intelligence. 1998. Vol. 106. Pp. 181–203.

12. Mertens S. A complete anytime algorithm for balanced number partitioning // CoRR. 1999. Vol. cs.DS/9903011.

13. Zulawinski B. W., Iii W. F. P., Goodman E. D. The grouping genetic algorithm (gga) applied to the bin balancing problem.

14. Public Data Sets. URL: http://aws.amazon.com/publicdatasets/ (дата обращения: 20.05.2012).

15. Tf-idf weighting. URL: http://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html (дата обращения: 20.05.2012).

16. GanttProject. URL: http://www.ganttproject.biz/ (дата обращения: 20.05.2012).

17. Фаулер Рефакторинг. Улучшение существующего кода. 2010. Pp. 352 – 353.