УДК 681.324

А. В. Митяков, Ю. С. Татаринов
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Подходы к эффективному исполнению итеративных алгоритмов на модели MapReduce

Итеративные алгоритмы представляют собой важный класс задач и встречаются во многих областях, таких, как data-mining, машинное обучение, ссылочный анализ и др. Наиболее известным и часто применимым инструментом обработки «больших данных» является модель MapReduce, и в частности ее открытая реализация Hadoop. В данной статье приводится обзор существующих подходов к реализации модели MapReduce, поддерживающей эффективное выполнение итеративных алгоритмов. Показано, что основным моментом, отрицательно влияющим на производительность итеративных алгоритмов на модели MapReduce, является дисковый и сетевой ввод-вывод, которого в большинстве случаев можно избежать посредством различных модификаций модели и программной среды, ее реализующей.

Big-data, обработка данных, MapReduce, итеративные алгоритмы

Необходимость обработки больших объемов данных существует в различных областях. В таких науках, как астрономия, физика частиц или биоинформатика, объем данных уже исчисляется в петабайтах. С появлением web 2.0 и ростом популярности социальных сетей огромные объемы данных сохраняются ежедневно. По отчетам компании Facebook 500 Гбайт новых данных сохраняется каждый день*.

В связи с этим эффективная обработка больших объемов данных является весьма актуальной задачей. MapReduce [1] – в настоящий момент наиболее популярный фреймворк для решения задач, связанных с обработкой больших объемов данных. MapReduce предоставляет разработчику простую программную модель и возможность распределенных вычислений с гарантированной отказоустойчивостью. Фреймворк позволяет разработчику, не знакомому с особенностями распределенных систем, реализовывать алгоритмы, работающие над большими объемами данных в рамках вычислительного кластера.

MapReduce спроектирован для задач, требующих пакетной обработки данных, однако позволяет решать произвольные задачи. Как любая обобщенная модель, MapReduce проигрывает на ряде задач более узкоспециализированным системам [2]. Одним из классов таких задач являются итеративные алгоритмы. Класс итеративных алгоритмов весьма важен в различных областях знаний, а модель MapReduce является уже хорошо проверенным и простым инструментом для реализации распределенных алгоритмов обработки данных. В связи с этим существует ряд работ, основная цель которых – модификация модели MapReduce или реализующего ее программного комплекса для эффективной поддержки итеративных алгоритмов. В данной статье рассмотрим основные предлагаемые подходы в этом направлении.

MapReduce. MapReduce [3] – это модель распределенных вычислений на кластере из гомогенных вычислительных машин. Данная модель, в отличие от наиболее распространенных и используемых в данной области моделей, таких, как MPI и PVM, находится на более высоком уровне абстракции с точки зрения разработчика. Фреймворк, реализующий данную модель, берет на себя задачи синхронизации параллельных процессов, обеспечения отказоустойчивости и масштабируемости, позволяя разработчику сосредоточиться на решении конкретной задачи. Основой данной мо-

* Facebook is collecting data – 500 terabytes a day. URL: http://gigaom.com/2012/08/22/facebook-is-collecting-your-data-500-terabytes-a-day.

дели являются две описываемые разработчиком функции – Map и Reduce. Решение конкретной задачи сводится к последовательному выполнению MapReduce-задач, в свою очередь состоящих из двух этапов – выполнения функций Map и Reduce. Схема работы модели MapReduce представлена на рис. 1.

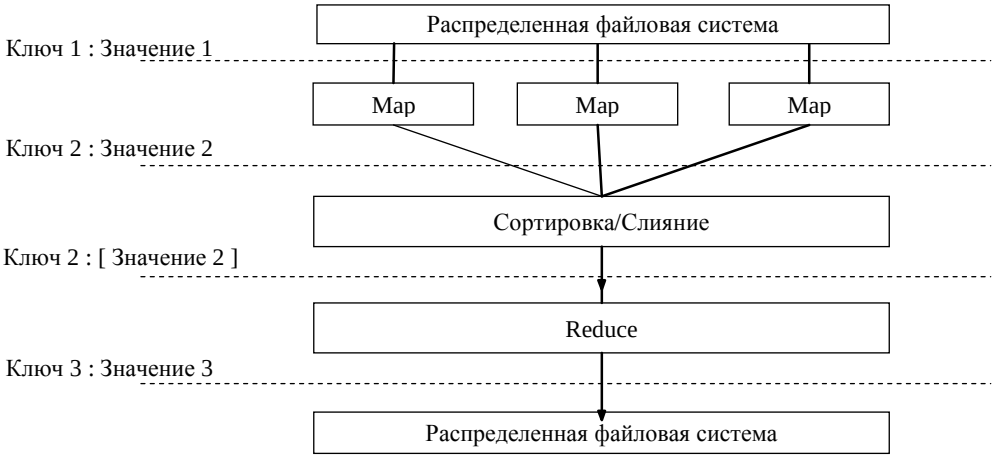

Рис. 1

Функция Map получает на входе пару ключ-значение и на выходе дает массив промежуточных значений, также представляющих собой пары ключ-значение. Среда исполнения MapReduce запускает экземпляры функции Map сразу на нескольких вычислительных узлах, автоматически распределяя между ними данные (partitioning). Далее в процессе выполнения MapReduce-задачи следует барьер – ожидание выполнения всех Map-задач, после которого промежуточные данные из всех Map-узлов группируются по ключу и передаются в функцию Reduce. На входе функция Reduce получает пару ключ-«массив значений» и производит некоторую агрегацию данных, вновь порождая на выходе пары ключ-значение, которые записываются в распределенную файловую систему и могут быть как окончательным результатом, так и исходными данными для следующей MapReduce-задачи.

Благодаря двум функциям Map/Reduce и синхронизации между ними модель гарантирует отсутствие связей между данными, что дает возможность их параллельной обработки и обеспечивает простое масштабирование на огромное количество вычислительных узлов (компания Yahoo запустила MapReduce-кластер на 10 000 машин[*]). Основой отказоустойчивости данной модели является распределенная файловая система, которая обеспечивает доступность данных для всех узлов кластера. Когда один из узлов кластера выходит из строя, управляющий узел назначает его часть работы другому свободному узлу, который загружает исходные данные из распределенной файловой системы.

Многочисленные работы по изучению возможности реализации различных алгоритмов на модели MapReduce показали, что данная модель подходит для решения широкого класса задач: индексация [4], data mining [5], машинное обучение [6] и др. Одни алгоритмы реализуются на MapReduce естественным образом, другие приходится сильно адаптировать под данную модель, что зачастую существенно сказывается на их производительности (в сравнении с реализацией на MPI). Примером последних является целый класс – итеративные алгоритмы.

Итеративные алгоритмы. Итеративные алгоритмы представляют собой важный класс задач, лежащий в основе многих алгоритмов, таких, как алгоритмы анализа, data mining, машинного обучения, алгоритмы обработки графов и др. В данном классе алгоритмов вычисления производятся в цикле, итерационно – при этом выходные данные одной итерации являются входными для следующей. Итеративный процесс продолжается до тех пор, пока не выполнится некоторое условие останова.

Данный класс алгоритмов можно реализовать на модели MapReduce, однако не самым лучшим по быстродействию способом. Рассмотрим основные недостатки модели с точки зрения реализации итеративных алгоритмов на примере алгоритма кластеризации *k*-средних.

[*] The Next Generation of Apache Hadoop MapReduce URL: http://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-3061.html.

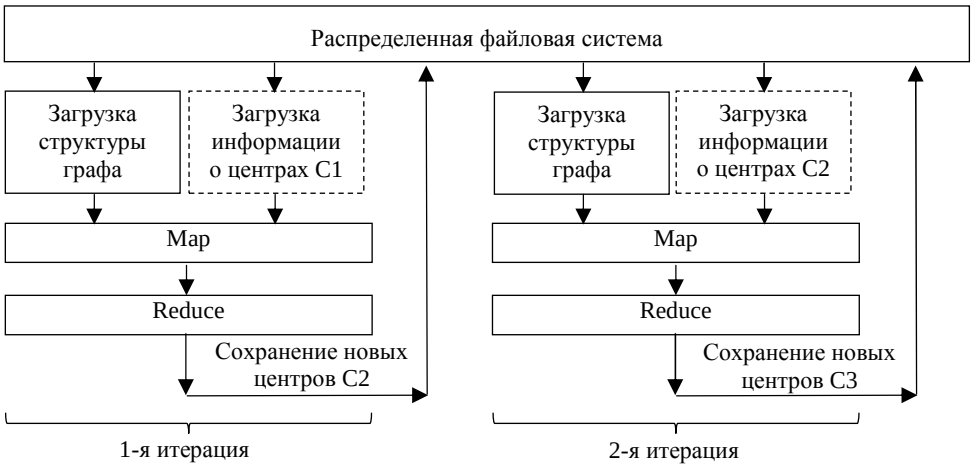

Рис. 2

Алгоритм *k*-средних. Алгоритм *k*-средних является общепризнанной техникой кластерного анализа. Реализация данного алгоритма на модели MapReduce представлена в [7], [8] и коротко состоит из следующих шагов:

1. Выбирается *k* произвольных центров.

2. На этапе map для каждой точки определяется ближайший центр:

вход: номер точки – вектор точки;

выход: номер центра – вектор точки.

3. После завершения всех map-задач на этапе reduce для всех точек, принадлежащих некоторому центру, определяется новый центр как среднее всех точек в рамках текущего центра. В качестве расстояния между двумя точками обычно используется евклидово расстояние.

4. Шаги 2–4 повторяются либо до тех пор, пока разница между значениями центров находится в рамках определенной погрешности, либо до выполнения определенного числа итераций.

На рис. 2 представлена схема работы алгоритма *k*-средних, реализованного на модели MapReduce. На схеме представлены две итерации алгоритма. В начале каждой итерации перед выполнением функции Map происходит загрузка данных в узлы кластера. При этом каждый раз загружается информация об анализируемом графе, неизменная на протяжении работы всего алгоритма, и информация о текущих рассчитанных центрах, которая изменяется между итерациями.

Проверка условия. Проверка условия останова в данном конкретном примере легко осуществляется в рамках управляющей программы, так как количество центров кластеров обычно небольшое и легко помещается в память одной машины. Однако в общем случае проверка условия может быть самостоятельной MapReduce-задачей (например, в ряде графовых алгоритмов, таких, как PageRank [9], поиск в ширину и др.), что приведет к дополнительной передаче данных в узлы из распределенной файловой системы.

Ограничения модели MapReduce. Таким образом, итеративные алгоритмы можно реализовать на модели MapReduce, однако для этого разработчику необходимо явно описать множество MapReduce-задач и вручную заниматься их оркестровкой с помощью управляющей программы. При ручной оркестровке итеративных алгоритмов можно выделить две основные проблемы.

Первая проблема связана с тем, что для итеративных алгоритмов характерна неизменность большей части данных между итерациями (например, множество вершин и дуг исходного графа). При этом в модели MapReduce данные в любом случае должны быть вновь загружены в узлы и обработаны заново в каждой итерации, что требует значительного времени на дисковый и сетевой ввод-вывод. Связано это с тем, что результаты функции Reduce записываются в распределенную файловую систему для того, чтобы было возможно использовать их в следующей MapReduce-задаче, которая может сильно отличаться от предыдущей. В случае итеративных алгоритмов следующие MapReduce-задачи заранее известны и совпадают с предыдущими, вследствие чего выгрузка результатов функции Reduce в распределенную файловую систему не является обязательной. В различных исследованиях приводятся данные решения реальных задач с помощью программного комплекса Hadoop, из которых следует, что время, затраченное на дисковый и сетевой ввод-вывод 10^8 байт данных в

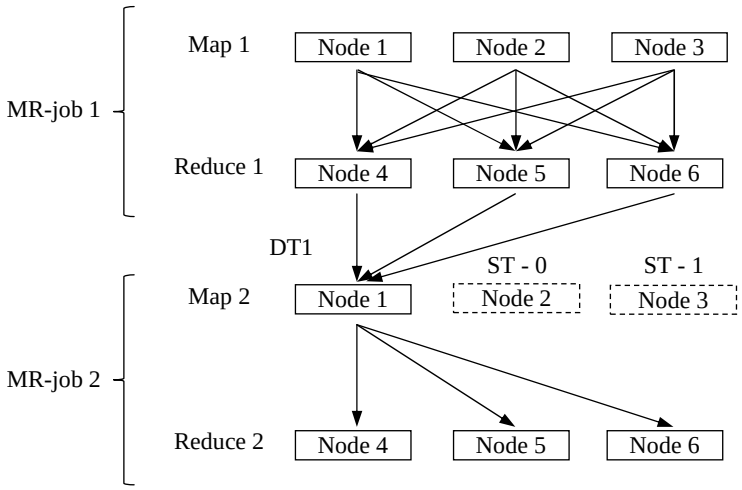

Рис. 3

10 000 раз превышает время на выполнение 10^8 операций умножения. Таким образом, дисковый и сетевой ввод-вывод в распределенных вычислениях является основной проблемой, влияющей на производительность распределенных алгоритмов.

Вторая проблема заключается в проверке условия останова. В большинстве итеративных алгоритмов критерием останова является неизменность данных между двумя последовательными итерациями. Организация такой проверки может требовать дополнительного выполнения MapReduce-задачи на каждой итерации, что вновь влечет за собой расходы из-за дискового и сетевого ввода-вывода.

Существуют различные подходы по устранению описанных проблем при реализации итеративных алгоритмов на модели MapReduce. Далее рассмотрим предлагаемые методы решения.

HaLoop. HaLoop [10] представляет собой программный каркас, основанный на открытой реализации модели MapReduce, – Hadoop. HaLoop позволяет разработчику реализовать итеративные алгоритмы на модели MapReduce, предоставляя ему больше возможностей по оптимизации производительности дискового и сетевого ввода-вывода. HaLoop может подойти для реализации произвольного итеративного алгоритма, однако выбор архитектурных решений при его проектировании был основан на отдельном подклассе итеративных алгоритмов, а именно алгоритмов анализа, в связи с чем дает существенный прирост в производительности в большей степени для алгоритмов данного класса. Основная идея данного фреймворка заключается в следующем. Для алгоритмов анализа данных характерна операция JOIN, т. е. соединение данных по общему ключу. Обеспечение производительности данной операции – ключевая особенность этого фреймворка.

Идеология HaLoop предполагает, что в рамках реализуемого итеративного алгоритма превалирует операция JOIN, т. е. соединение двух key-value-таблиц по общему столбцу key. При этом одна из таблиц является инвариантной в рамках итерационного процесса (ST – static table). Вторая же key-value-таблица (как правило, гораздо меньшего размера) изменяется от итерации к итерации (DT – dynamic table). Схема работы операции JOIN над данными таблицами в рамках модели MapReduce представлена на рис. 3.

Предположим, что имеется 3 вычислительных узла для Map-задач и 3 узла для Reduce-задач: node 1, node 2, node 3 и т. д. В node 1 загружена исходная таблица DT (версия 0). Так как таблица ST гораздо большего размера, чем DT, то ее данные делятся между узлами node 2 и node 3. Функция Map 1 в узле node 1 получает на входе ключ-значение из таблицы DT, а узлы node 2 и node 3 – ключ-значение из таблицы ST, но из разных ее частей (partitions). На выходе функции Map 1 получаются также пары ключ-значение, где ключом является то поле таблиц DT и ST, по которому происходит JOIN, а в качестве значения попадают элементы из множества значений DT и ST.

Затем результаты работы функций Map загружаются по соответствующим Reduce-узлам и склеиваются по JOIN-ключу. Таким образом, еще до начала выполнения самой Reduce-функции операция JOIN для двух таблиц уже завершена. Далее в ходе выполнения функции Reduce в качестве результатов возвращаются пары ключ-значение, которые в совокупности являются новой версией таблицы DT (версия 1). На этом заканчивается первая итерация. На следующей ите-

рации (MR-job 2) вновь необходимо выполнить JOIN таблицы ST с уже обновленной таблицей DT1. В этом месте и начинает работать HaLoop. HaLoop хранит структуру данных с информацией о том, в каком узле какие данные находятся, и старается запланировать Map- и Reduce-задачи над конкретными данными именно на тех узлах, где эти данные находятся. Благодаря наличию планировщика задач, знающего о местоположении закешированных в узлах данных, можно существенно сократить время выполнения алгоритма за счет отсутствия ненужной перезагрузки данных из распределенной файловой системы. Например, применительно к рис. 2 на второй итерации нет смысла запускать функцию Map в узлах node 2 и node 3 над данными из таблицы ST, так как их результат в связи с неизменностью таблицы ST будет таким же, как и в предыдущих итерациях, и уже есть в Reduce-узлах node 4, node 5 и node 6.

Таким образом, благодаря алгоритму планирования MapReduce-задач в HaLoop для всех последующих итераций необходим всего один узел для выполнения функции Map, что экономит не только количество задействованных узлов, но и время, которое в противном случае тратилось бы на бесполезную перезапись данных из распределенной файловой системы в Map-узлы, а затем из Map-узлов в Reduce-узлы.

С точки зрения разработчика HaLoop предоставляет специальный API, который позволяет задавать, какие данные будут инвариантными между итерациями, а какие следует кешировать в узлах. Для проверки условия останова HaLoop предоставляет 3 возможных сценария:

1. Останов после определенного количества итераций.

2. Останов в случае, если результирующие данные перестают изменяться от итерации к итерации.

3. Останов в случае, если разность результирующих данных между двумя итерациями меньше некоторого заданного E.

iMapReduce. iMapReduce [11] – это фреймворк, представляющий собой модифицированную версию открытой реализации MapReduce – Hadoop. Цель создания данного фреймворка – поддержка эффективного исполнения итеративных алгоритмов, реализованных на модели MapReduce.

iMapReduce вводит специальный вид MapReduce-задач – постоянные задачи (persistent tasks), которые продолжают работать не завершаясь в течение всего итеративного процесса. Другими словами, в отличие от классических Map- и Reduce-задач, которые после обработки всех доступных данных завершаются, задачи в iMapReduce просто переходят в режим ожидания новых данных, что позволяет сэкономить время на создание и планирование новых MapReduce-задач. iMapReduce поддерживает соответствие 1 к 1 между задачами map и reduce. Осуществляется это благодаря специальному разбиению исходных данных. Для этого используется специальная хеш-функция F, ставящая в соответствие каждому ключу номер MapReduce-пары, которая будет заниматься его обработкой:

$$mapID = reduceID = F(key, mapCount).$$

Благодаря этому каждый Map точно знает, какому конкретно Reduce-узлу следует послать текущие данные, и наоборот, каждый Reduce знает, куда следует послать данные по определенному ключу. Данная особенность позволяет некоторым Map-задачам приступить к обработке данных следующей итерации раньше, не дожидаясь всех остальных Map-задач, и тем самым частично избавиться от барьера в выполнении Map-Reduce-задач и повысить степень параллелизма. Схема работы iMapReduce приведена на рис. 4.

На рис. 4, *а* представлена схема работы iMapReduce с асинхронно выполняющимися Map-задачами. Для сравнения на рис. 4, *б* приводится обычная реализация выполнения на модели MapReduce. Благодаря тому, что одному Map-узлу (M) соответствует ровно один Reduce-узел (R), конкретному Map-узлу для того, чтобы начать следующую итерацию, нужно дождаться завершения не всех, а лишь конкретного Reduce-узла. В качестве дополнительной оптимизации iMapReudce предлагает Map-узлам возможность начать работу, как только соответствующий Reduce-узел обработает хотя бы одну запись. Осуществляется это благодаря наличию постоянного сокет-соединения между Map- и соответствующим ему Reduce-узлом.

Недостаток данной модели в том, что если Map- и Reduce-задачи имеют разные ключи, то выполненять асинхронные Map-задачи становится невозможным и процесс выполнения будет таким же, как и при классической реализации MapReduce. Таким образом, предложенная модель имеет явные ограничения на класс алгоритмов, которые на ней можно эффективно реализовать.

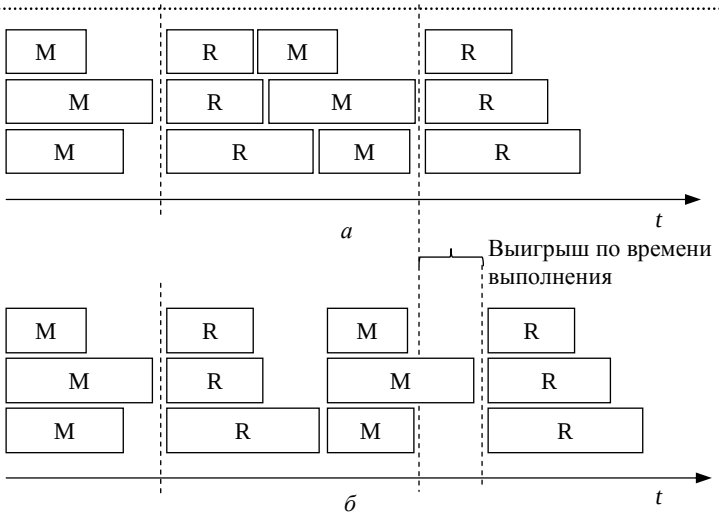

Рис. 4

С точки зрения разработчика iMapReduce предлагает программную модель, схожую с той, которая представлена ранее при описании HaLoop.

iHadoop. Реализация фреймворка iHadoop [12] также построена на базе Hadoop. iHadoop является идейным наследником фреймворка iMapReduce и развивает предложенные в нем концепции. В частности, как и iMapReduce, предлагает возможность исполнения асинхронных Map-здачач. Основным отличием и вкладом iHadoop является наличие динамического планировщика MapReduce-задач. В отличие от iMapReduce, в котором планирование задач является статическим и каждому Map-узлу в соответствие ставится конкретный Reduce-узел, iHadoop предлагает динамическое планирование, суть которого заключается в следующем. Выбор узла, на котором будет запланирована Reduce-задача, осуществляется таким образом, чтобы Map-задачи, которые будут обрабатывать результаты работы данной Reduce-задачи, оказалась на одном вычислительном узле. Целью данного планирования является минимизация времени, затрачиваемого на сетевой ввод-вывод.

Второй особенностью фреймворка iHadoop является подход к проверке условия останова. Идея данного подхода заключается в предположении, что большую часть итераций проверка условия останова выполняется впустую. Другими словами, если условие останова выполняется, например, на седьмой итерации, то 6 итераций проверка будет происходить «впустую» и будет тратиться лишнее время и задерживаться начало следующей итерации. В связи с этим iHadoop предлагает запускать MapReduce-задачу проверки условия останова параллельно с выполнением итераций и начинать следующую итерацию, не дожидаясь ее выполнения. Благодаря этому «лишние» проверки условия останова не задерживают выполнение самого алгоритма (не считая расходуемых при этом ресурсов кластера). Если параллельно запущенная проверка условия останова завершается положительно, то в качестве результата iHadoop выбирает не текущую, а предыдущую итерацию.

Twister. Twister [13] представляет основанную на потоковой обработке данных реализацию модели MapReduce, которая имеет поддержку итеративных алгоритмов. Для того чтобы избежать постоянной перезагрузки инвариантных между итерациями данных, Twister использует концепцию под названием «long-running» задач. Это означает, что фреймворк не запускает новую Map- и Reduce-задачу на каждой итерации, а использует уже существующие задачи с уже загруженными в узлах данными (аналог «постоянных» задач в iMapReduce). Помимо кеширования инвариантных данных Twister предлагает дополнительные средства для повышения производительности. Для того чтобы сократить время на локальном вводе-выводе, фреймворк предлагает инфраструктуру издатель/подписчики, построенную распределенно в оперативной памяти всех узлов кластера. Данная инфраструктура используется для доставки данных между Map-Reduce-узлами в обход файловой системы, доступ к которой является наиболее затратной операцией с точки зрения времени.

Главным узким местом данного фреймворка является допущение о том, что все необходимые для обработки данные должны поместиться в

распределенную память, построенную на узлах кластера. Данное допущение в условиях гомогенного кластера представляется весьма критичным в связи с тем, что в подобном кластере ресурсы узлов обычно весьма ограничены.

Таким образом, можно выделить следующие направления в модификации MapReduce-модели, которые позволяют повысить эффективность выполнения итеративных алгоритмов в распределенной среде:

1. Логическое и физическое разбиение исходных данных на 2 вида: статическую и динамическую части. Статическая часть данных является инвариантной между итерациями и благодаря этому может быть закеширована в соответствующих узлах. Динамическая же часть изменяется от итерации к итерации, и только она должна передаваться между Map- и Reduce-узлами.

2. Кеширование данных в Map- и Reduce-узлах, позволяющее предотвратить повторную загрузку данных из распределенной файловой системы между различными итерациями, что значительно снижает время на сетевой и дисковый ввод/вывод.

3. Оптимизация времени выполнения MapReduce-задач, отвечающих за проверку условия останова.

4. Введение дополнительного параллелизма благодаря введению асинхронных map- или reduce-задач. Однако данный способ оптимизации времени выполнения является более общим подходом и применим не только к итеративным алгоритмам.

Так или иначе все представленные направления оптимизации сводятся к одному – к минимизации дискового и сетевого ввода-вывода. Достигается эффективность разными способами: в случае сетевого ввода-вывод – кешированием данных в узлах, а в случае дискового – переносом данных в оперативную память системы.

СПИСОК ЛИТЕРАТУРЫ

1. Петров Д. Л., Кротов С. В., Митяков А. В. Модель системы MapReduce, основанная на инфраструктуре облачных вычислений // Изв. СПбГЭТУ «ЛЭТИ». 2011. № 10. С. 25–31.

2. Митяков А. В., Татаринов Ю. С. Проблема реализации итеративных алгоритмов на модели MapReduce // Тр. XX Всерос. науч.-методической конф. «Телематика'2013», СПб., 2013. С. 260–261.

3. Dean J., Ghemawat S. MapReduce: Simplified data processing on large clusters // OSDI, San Francisco, California, USA. 2004. Vol. 51. P. 137–140.

4. Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search / J. Lin, D. Metzler, T. Elsayed, L. Wang // In Proc. of the 18th Text REtrieval Conf. (TREC), Gaithersburg, Maryland, USA, 2009. P. 169–171.

5. Papadimitriou S., Sun J. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining // In Proc. of the Eighth IEEE ICDM, Pisa, Italy, 2008. P. 512–513.

6. Map-Reduce for Machine Learning on Multicore / Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, Kunle Olukotun // In Proc. of NIPS, Vancouver, B. C., Canada, 2007. P. 282–283.

7. Evaluating mapreduce for multi-core and multiprocessor system / C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis // In Proc. IEEE HPCA, Phoenix, Arizona, USA, 2007. P. 14–15.

8. Map-reduce for machine learning on multicore / C.-T Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. Bradski, A. Y. Ng, K. Olukotun // In Proc. of NIPS, Vancouver, B.C., Canada, 2007. P. 282–284.

9. Brin S., Page. L. The Anatomy of a Large-Scale Hypertextual Web Search Engine // Proc. of the seventh Intern. Conf. on World Wide Web (WWW), Brisbane, Australia, 1998. Vol. 30. P. 111–112.

10. Haloop: Ecient iterative data processing on large clusters / Y. Bu, B. Howe, M. Balazinska, D. M. Ernst // In VLDB '10, Singapore. 2010. Vol. 3. P. 287–289.

11. Imapreduce: A distributed computing framework for iterative computation / Y. Zhang, Q. Gao, L. Gao, C. Wang // In DataCloud '11, Anchorage, Alaska, 2011. Vol. 10. P. 52–54.

12. Elnikety E., Elsayed T., Ramadan H. E. iHadoop: Asynchronous Iterations for MapReduce // In Proc. the 2011 IEEE Third Intern. Conf. on Cloud Computing Technology and Science, Athens, Greece, 2011. P. 83–86.

13. Twister: a Runtime for Iterative MapReduce / J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, Seung-Hee Bae, J. Qiu, G. Fox // In Proc. of the 19th ACM Intern. Symp. on High Performance Distributed Computing (HPDC), Chicago, Illinois, USA, 2010. P. 812–814.