Чак Лэм

Hadoop в действии

Москва, 2012

УДК 004.6:004.42Apache Hadoop
ББК 32.973.26-018.2
Л92

Л92 **Чак Лэм**

Hadoop в действии. – М.: ДМК Пресс, 2012. – 424с.: ил.

ISBN 978-5-94074-785-7

Обработка больших массивов данных с помощью традиционных СУБД может оказаться трудным делом. Apache Hadoop — это каркас для разработки приложений, предназначенных для выполнения в распределенном кластере, без применения SQL. Такие приложения прекрасно масштабируются и могут обрабатывать гигантские массивы данных. Если вам требуется произвести анализ данных, то Hadoop – как раз то, что надо.

Прочитав эту книгу, вы познакомитесь с предметом и научитесь писать программы в стиле MapReduce. После нескольких простых примеров автор быстро переходит к вопросу об использовании Hadoop для решения более сложных задач анализа данных. Описываются рекомендованные приемы и паттерны проектирования, полезные при программировании для MapReduce.

Для чтения книги требуется знание основ языка Java. Некоторое знакомство с математической статистикой поможет разобраться в более сложных примерах.

УДК 004.6:004.42Apache Hadoop
ББК 32.973.26-018.2

ISBN 978-1-93518-219-1 (англ.)

ISBN 978-5-94074-785-7

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Уже давно данные очаровывают меня. Еще на младших курсах, изучая электротехнику, я открыл для себя цифровую обработку сигналов и прикипел к ней всей душой. Я понял, что музыку, фотографии и множество других вещей можно рассматривать как данные. Оказалось, что создавать и усиливать эмоциональные переживания можно с помощью вычислений. Ничего интереснее я даже представить себе не мог.

Со временем передо мной раскрылись новые, не менее удивительные аспекты данных. Вот уже несколько лет, как я занимаюсь социальными сетями и большими массивами данных. Именно большие массивы бросают вызов моему интеллекту. Статистический анализ данных я освоил уже давно, а для работы с новыми типами данных были нужны «всего лишь» новые математические методы. Конечно, это не просто, но по крайней мере я этому учился, а уж в ресурсах для пополнения знаний недостатка и вовсе нет. С другой стороны, для работы с большими объемами данных необходим совсем другой системный подход и новые способы программирования. Этому меня не учили, но – и это даже важнее – не один я оказался в таком положении. Знания о практических методах обработки больших массивов данных сродни черной магии. Существует немало инструментов и приемов масштабируемой обработки данных, в частности, кэширование (например, с помощью программы memcached), репликация, секционирование и, конечно, MapReduce/Hadoop. Последние несколько лет я потратил на приобретение и совершенствование навыков в этой области.

Лично для меня наиболее сложной оказалась средняя часть кривой обучения. Поначалу не так уж трудно найти блоги с вводным материалом и презентации, объясняющие, как написать пример уровня «Здравствуй, мир». А приобретя начальные знания, вы уже можете задавать дополнительные вопросы в списках рассылки, общаться со специалистами на различных встречах и конференциях и даже самостоятельно читать исходный код. Но вот в середине зияет гигантский

провал — аппетит уже разыгрался, а как сформулировать последующие вопросы, неясно. Эта проблема стоит особенно остро для таких новейших технологий, как Hadoop. Необходимо некое упорядоченное изложение, которое начиналось бы с примера «Здравствуй, мир» и доводило читателя до точки, в которой он смог бы без особых усилий применить Hadoop на практике. Именно так я и представляю себе эту книгу. К счастью, я обнаружил, что серия «In Action», выходящая в издательстве Manning, отлично согласуется с моей целью, и к тому же там есть прекрасные редакторы, помогавшие мне на всем протяжении работы.

Я получал удовольствие, когда писал книгу, и надеюсь, что она станет для вас началом увлекательного путешествия в мир Hadoop.

БЛАГОДАРНОСТИ

В эту книгу внесло вклад много людей. Прежде всего, я хочу поблагодарить Джеймса Уоррена (James Warren). Он возглавлял аналитический отдел в компании RockYou и вместе с ним мы стремились популяризировать Hadoop в этой организации. Я многому научился у него, он даже помогал мне писать ранние варианты текста.

На мое счастье нашлось немало людей, познакомивших меня с интересными примерами за рамками индустрии Web 2.0. В частности, я хочу выразить благодарность Чжи Гуо Люо (Zhiguo Luo), Мень Ху (Meng Xu), Шаолинь Сун (Shaoling Sun), Кену Макиннису (Ken MacInnis), Райану Роусону (Ryan Rawson), Вуку Эрчеговачу (Vuk Ercegovac), Раджасекару Кришнамурти (Rajasekar Krishnamurthy), Срираму Рагхавану (Sriram Raghavan), Фредерику Рейсу (Frederick Reiss), Юджину Шекита (Eugene Shekita), Сандипу Тата (Sandeep Tata), Шивакумару Вайтианатхану (Shivakumar Vaithyanathan) и Хуай Ю Чжу (Huaiyu Zhu).

Я также благодарен рецензентам этой книги. Они прислали ценные отзывы на первые варианты. В частности, технический редактор Пол О'Рорк (Paul O'Rorke) вышел далеко за пределы формальных обязанностей и внес ряд крайне полезных предложений о том, как улучшить рукопись. Я надеюсь, что когда-нибудь он сам напишет книгу. Кроме того, я получал истинное удовольствие от долгих бесед с Джонатаном Као (Jonathan Cao). Его опыт работы с базами данных и крупномасштабными системами позволил мне шире взглянуть на возможности Hadoop и лучше понять их.

По ходу работы текст книги многократно читали и другие рецензенты, которых я хотел бы поблагодарить за их бесценный труд: Пол Стусяк (Paul Stusiak), Филипп К. Дженерт (Philipp K. Janert), Амин Мохаммед-Коулмен (Amin Mohammed-Coleman), Джон С. Гриффин (John S. Griffin), Марко Угетти (Marco Ughetti), Рик Вагнер (Rick Wagner), Кеннет Делонг (Kenneth DeLong), Джош Паттерсон (Josh Patterson), Срини Пенчикала (Srini Penchikala), Костантино Кербо (Costantino Cerbo), Стив Логран (Steve Loughran), Ара Абрахамян

(Ara Abrahamian), Бен Холл (Ben Hall), Эндрю Зимер (Andrew Siemer), Роберт Хансон (Robert Hanson), Кит Ким (Keith Kim), Сопан Шивейл (Sopan Shewale), Марион Стуртеван (Marion Sturtevant), Крис Чендлер (Chris Chandler), Эрик Реймонд (Eric Raymond) и Джероен Бенкхейсен (Jeroen Benckhuijsen).

Мне посчастливилось работать с коллективом чудесных людей в издательстве Manning. Отдельное спасибо Трою Мотту (Troy Mott), который побудил меня взяться за это дело и терпеливо дожидался, пока я его закончу. Спасибо также Таре Уолш (Tara Walsh), Карен Тегтмейер (Karen Tegtmeyer), Марьян Бейс (Marjan Bace), Мэри Пирджис (Mary Piergies) Синтии Кэйн (Cynthia Kane), Стивену Хонгу (Steven Hong), Рашель Шредер (Rachel Schroeder), Кэти Теннант (Katie Tennant) и Морин Спенсер (Maureen Spencer). Поддержка с их стороны была просто феноменальной. Лучшей компании для работы я себе даже представить не могу.

Само собой, добрых слов заслуживают все, кто внес свой вклад в разработку каркаса Hadoop и помогает развиваться сформировавшейся вокруг него экосистеме. Начало положил Дуг Каттинг (Doug Cutting), а компания Yahoo оказалась достаточно прозорливой, чтобы поддержать его с самых первых шагов. Теперь компания Cloudera представляет Hadoop вниманию более широкой аудитории корпоративных пользователей. Сейчас самое подходящее время присоединиться к растущему сообществу Hadoop.

И наконец, я хочу сказать спасибо всем своим друзьям, семье и коллегам, которые поддерживали меня на протяжении работы над книгой.

ОБ ЭТОЙ КНИГЕ

Hadoop – это каркас с открытым исходным кодом, в котором реализован алгоритм распределения и редукции MapReduce, лежащий в основе подхода Google к организации запросов к распределенным наборам данных, которые и составляют Интернет. В связи с таким определением возникает очевидный вопрос: что такое *распределение* (map) и зачем нужна последующая *редукция* (reduce)? Зачастую массивные наборы данных с трудом поддаются анализу и опросу традиционными средствами, особенно когда сами запросы весьма сложны. По существу, алгоритм MapReduce разбивает запрос и набор данных на несколько частей – это этап распределения. Отдельные компоненты запроса можно обработать параллельно, а затем *свести* полученные результаты воедино (*редуцировать*).

Читатель этой книги узнает, как использовать Hadoop и писать MapReduce-программы. Книга предназначена для программистов, архитекторов и руководителей проектов, связанных с обработкой большого объема данных в офлайновом режиме. Будет описано, как получить копию Hadoop, как организовать кластер и как писать программы анализа. Мы начнем с применения Hadoop в конфигурации по умолчанию к решению нескольких простых задач, например, об анализе изменения частоты вхождения слов в корпус документов; это позволит уяснить основные идеи Hadoop и MapReduce. Далее мы перейдем к базовым концепциям MapReduce-приложений, разрабатываемых с помощью Hadoop, и по ходу дела изучим компоненты каркаса, применение Hadoop к широкому спектру задач анализа данных и многочисленным примерам Hadoop в действии.

Алгоритм MapReduce сложен как концептуально, так и в плане реализации, а от пользователей каркаса Hadoop требуется изучить все средства и хитрости, необходимые для работы с ним. В этой книге не просто рассказывается о том, как запустить Hadoop; прочитав ее, вы научитесь писать полезные программы с использованием MapReduce.

Предполагается, что читатель хотя бы немного владеет языком Java, поскольку именно на нем написано большинство примеров.

Знакомство с основами математической статистики (например, гистограммами и корреляцией) поможет разобраться в более сложных примерах обработки данных.

Структура книги

Эта книга состоит из 12 глав, разбитых на три части.

Часть 1 состоит из трех глав и представляет собой введение в каркас Hadoop. Здесь излагаются те базовые сведения, которые необходимо знать для понимания и использования каркаса. Описывается, из каких аппаратных компонентов состоит кластер Hadoop, рассказывается об установке и конфигурировании системы. Также в части 1 дается общее представление о каркасе MapReduce и приводится пример первой MapReduce-программы.

Часть 2 «Hadoop в действии» состоит из пяти глав, в которых описаны практические навыки, необходимые для составления и запуска программ обработки данных в среде Hadoop. Здесь мы рассмотрим многочисленные примеры применения Hadoop к анализу набора данных о патентах, в том числе и такие нетривиальные алгоритмы, как фильтр Блума. Мы также поговорим о приемах программирования и администрирования, чрезвычайно полезных при работе с Hadoop в производственной среде.

Часть 3 «Hadoop в реальной жизни», в которую входят последние четыре главы, посвящена обширной экосистеме, сложившейся вокруг Hadoop. Существуют облачные службы, которые позволяют обойтись без покупки собственного оборудования для создания кластера. Имеются также дополнительные пакеты, предлагающие высокоуровневые абстракции, надстроенные над MapReduce. Наконец, мы рассмотрим несколько примеров практического применения Hadoop к решению реальных задач бизнеса.

В приложении приведен список команд HDFS.

Графическое выделение и загрузка исходного кода

Исходный код в листингах и в основном тексте набран `моноширинным шрифтом`. Многие листинги сопровождаются аннотациями, в которых излагаются важные концепции. В некоторых случаях в листингах присутствуют нумерованные маркеры, с которыми соотносятся последующие пояснения.

Код всех приведенных в книге примеров можно скачать с сайта издательства по адресу www.manning.com/HadoopinAction.

АВТОР В СЕТИ

Приобретение книги «*Hadoop в действии*» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/HadoopinAction. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

ОБ АВТОРЕ

В настоящее время Чак Лэм занят организацией социальной сети для пользователей мобильных устройств под названием *RollCall*. Это будет социальный секретарь для активных людей.

Раньше Чак работал техническим руководителем проекта в компании RockYou. В этой должности он разрабатывал социальные приложения и инфраструктуру обработки данных, рассчитанную на сотни миллионов пользователей. Он применял A/B тестирование и статистический анализ для оптимизации скорости распространения социального приложения среди пользователей (его *виральности*). Ему удавалось значительно – иногда на порядок – повысить частоту посещения страниц.

Обработкой больших объемов данных Чак заинтересовался во время работы над кандидатской диссертацией в Стэнфордском университете. Он узнал о том, какой серьезный эффект большие массивы данных могут оказать на машинное обучение, и начал изучать последствия. В его диссертации «Вычислительные методы сбора данных» впервые были исследованы новые подходы к сбору данных для машинного обучения – с применением идей, заимствованных из программ с открытым исходным кодом и онлайновых игр.

Об иллюстрации на обложке

Рисунок на обложке книги «Hadoop в действии» называется «Юноша из селения Кистане в Далмации». Репродукция взята из альбома традиционной хорватской одежды середины девятнадцатого века, составленного Николой Арсеновичем и опубликованного этнографическим музеем города Сплит, Хорватия, в 2003 году. Рисунок был получен при содействии библиотекаря этнографического музея Сплита, который расположен в той части средневекового центра города, которая восходит еще к временам римского владычества: рядом с руинами дворца императора Диоклетиана, датируемыми примерно 304 годом до н. э. В альбоме собраны тщательно раскрашенные изображения людей из разных районов Хорватии, сопровождаемые описаниями костюмов и деталей повседневного быта.

Сейчас Кистане – небольшой городок в Буковице, одном из географических регионов Хорватии. Он расположен в северной Далмации, области с богатой историей времен римлян и венецианцев. Хорватское слово «mamok» означает «холостяк», «кавалер», «ухажер» – неженатый молодой человек брачного возраста. Изображенный на обложке юноша выглядит очень щеголевато в выглаженной белой полотняной рубашке и расшитом жилете – вероятно, это его лучший наряд, в котором он ходит в церковь и на праздники, или приоделся для свидания с девушкой.

Манера одеваться и общий уклад жизни за прошедшие 200 лет сильно изменились, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов, что уж говорить о деревеньках и городках, разделенных несколькими километрами. Мы обменяли культурное разнообразие на иное устройство личной жизни – основанное на многостороннем и стремительном технологическом развитии.

Издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на иллюстрациях в старых книгах и собраниях, – примером может служить обложка этой книги.

Часть 1

Hadoop – каркас распределенного программирования

В части 1 закладываются основы для понимания и использования Hadoop. Мы расскажем, из каких аппаратных компонентов состоит кластер Hadoop, а также опишем процедуру установки и конфигурирования, необходимую для создания работоспособной системы. Мы дадим общее представление о каркасе MapReduce, после чего вы напишете и запустите свою первую MapReduce-программу.

ГЛАВА 1.
Введение в Hadoop

В этой главе:

- Основы составления масштабируемых, распределенных программ для обработки данных.
- Что представляют собой Hadoop и MapReduce.
- Создание и запуск простой MapReduce-программы.

В современном мире данные окружают нас со всех сторон. Мы загружаем на серверы видео, делаем фотографии с помощью мобильных телефонов, посылаем текстовые сообщения друзьям, изменяем их статус в системе Facebook, оставляем комментарии на разных сайтах, щелкаем по рекламным баннерам и т. д. Компьютеры со своей стороны также порождают и хранят все больше и больше данных. Возможно, даже эта книга представлена у вас в электронном виде, и читаете вы ее на экране своего компьютера. И уж точно факт покупки этой книги отражен в базе данных какого-нибудь розничного продавца[1].

Проблемы, вызванные экспоненциальным ростом данных, первыми ощутили на себе такие находящиеся на переднем крае технологий компании, как Google, Yahoo, Amazon и Microsoft. Им приходилось просеивать терабайты и петабайты данных, чтобы понять, какие вебсайты популярны, какие книги пользуются спросом, какая реклама находит отклик у пользователей. Имеющиеся инструменты оказались не приспособлены к обработке столь больших объемов данных. Google стала первой компанией, начавшей пропагандировать систему *MapReduce*, которую применила для масштабирования обработки данных. Эта система вызвала большой интерес, так как многие другие компании уже столкнулись с аналогичными проблемами, а заново изобретать собственный инструментарий было нерентабельно.

[1] Вы ведь читаете легальную копию, правда?

Дуг Каттинг разглядел открывающиеся возможности и принялся за разработку версии MapReduce с открытым исходным кодом, которую назвал Hadoop. Вскоре Yahoo и другие компании объединили усилия в поддержку этого начинания. Ныне Hadoop составляет основную часть вычислительной инфраструктуры многих работающих в веб компаний, в частности, Yahoo, Facebook, LinkedIn и Twitter. Примериваются к ней и более традиционные организации, например, средства массовой информации и телекоммуникационные компании. В главе 12 описаны примеры использования Hadoop в таких компаниях, как New York Times, China Mobile и IBM.

Hadoop и, более общо, технологии распределенной обработки больших массивов данных, быстро становятся важным умением для широкого круга программистов. Сегодня программисту для эффективной работы необходимо знать о реляционных базах данных, сетях и безопасности, хотя еще пару десятков лет назад все это считалось факультативными навыками. Так и базовые знания о распределенной обработке данных скоро станут непременной частью багажа любого программиста. Ведущие университеты, например Стэнфордский и Карнеги-Меллона, уже ввели в программу факультетов информатики курс по изучению Hadoop. Эта книга поможет вам, программисту-практику, быстро освоить Hadoop и начать использовать его для обработки собственных наборов данных.

Эта глава содержит несколько более формальное введение в каркас Hadoop, рассматриваемый в контексте распределенных систем и систем обработки данных. Мы дадим обзор модели программирования MapReduce. Простой пример программы подсчета слов с помощью имеющихся инструментов поможет уяснить, какие проблемы возникают при обработке больших массивов данных. Затем вы напишете программу, основанную на использовании Hadoop, что позволит лучше оценить простоту этого каркаса. Мы также поговорим об истории Hadoop и некоторых перспективах парадигмы MapReduce. Но сначала позвольте мне вкратце объяснить, для чего я написал эту книгу и чем она может быть вам полезна.

1.1. Зачем написана книга «Hadoop в действии»?

На своем опыте я убедился, что поначалу Hadoop завораживает своими возможностями, но дело с трудом продвигается дальше написания элементарных примеров. Документация на официальном сайте

Hadoop довольна полная, но найти ясные ответы на прямо поставленные вопросы не всегда легко.

Решение данной проблемы и составляет задачу этой книги. Я не стану углубляться в скучные технические подробности, а поделюсь информацией, которая позволит вам быстро создавать полезный код. Попутно я расскажу о более сложных вопросах, часто возникающих на практике.

1.2. Что такое Hadoop?

Говоря формально, Hadoop – это каркас с открытым исходным кодом, предназначенный для создания и запуска распределенных приложений, обрабатывающих большие объемы данных. Распределенные вычисления – это широкая и многогранная область, но у Hadoop есть ряд важных отличительных особенностей, а именно:

- *Доступность* – Hadoop работает на крупных кластерах, собранных из стандартных компьютеров, или в вычислительном облаке, например на базе службы Elastic Compute Cloud (EC2), предлагаемой компанией Amazon.
- *Надежность* – поскольку Hadoop должен работать на стандартном оборудовании, его архитектура разработана с учетом возможности частых отказов. Большинство отказов можно обработать так, что характеристики кластера будут ухудшаться постепенно.
- *Масштабируемость* – Hadoop масштабируется линейно, то есть при увеличении объема данных достаточно добавить новые узлы в кластер.
- *Простота* – Hadoop позволяет пользователю быстро создавать эффективный параллельный код.

Доступность и простота Hadoop дают ему конкурентное преимущество в деле написания и запуска больших распределенных программ. Даже студент колледжа может быстро собрать собственный недорогой кластер Hadoop. С другой стороны, надежность и масштабируемость делают Hadoop подходящим средством даже для таких ответственных задач, которые решаются в компаниях Yahoo и Facebook. Поэтому Hadoop популярен как в академических кругах, так и в промышленности.

На рис. 1.1 показано, как пользователь взаимодействует с кластером Hadoop. Как видите, кластер Hadoop представляет собой на-

бор стандартных компьютеров, объединенных в сеть, физически расположенную в одном месте[2]. Как хранение, так и обработка данных происходят внутри этого «облака» машин. Разные пользователи отправляют кластеру Hadoop вычислительные «задания» с клиентских машин, например со своих настольных компьютеров, которые могут находиться далеко от кластера.

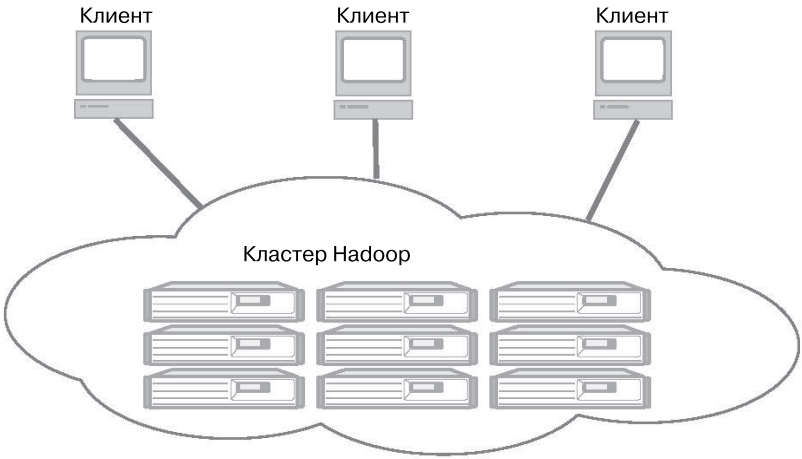


Рис. 1.1. Кластер Hadoop состоит из многих машин, которые хранят и параллельно обрабатывают большие наборы данных. Клиентские компьютеры посылают в это вычислительное облако задания и получают результаты.

Не все распределенные системы конфигурируются так, как показано на рис. 1.1. Краткое знакомство с другими распределенными системами поможет лучше понять философию Hadoop.

1.3. Сравнение Hadoop с другими распределенными системами

Закон Мура верно служил нам несколько десятилетий, но теперь создание все более и более мощных серверов не обязательно дает оптимальное решение крупномасштабных задач. Быстро набирает

[2] Хотя это и необязательно, обычно кластер Hadoop состоит из более-менее однородных машин на базе процессоров x86 под управлением ОС Linux. И почти всегда они находятся в одном центре обработки данных, часто даже в общем наборе стоек.

популярность альтернативный подход – объединение множества недорогих стандартных машин в функционально единую *распределенную систему*.

Чтобы понять, почему распределенные системы (масштабирование по горизонтали) оказались популярнее монолитных серверов (масштабирование по вертикали), надо принять во внимание производительность и цену современных технологий ввода/вывода. Высокопроизводительному компьютеру с четырьмя каналами ввода/вывода, производительностью 100 МБ/с каждый, потребуется три часа, чтобы только *прочитать* набор данных объемом 4 ТБ! Если же воспользоваться Hadoop, то такой же набор данных можно разбить на меньшие блоки (обычно размером 64 МБ), распределенные по нескольким машинам кластера с помощью распределенной файловой системы Hadoop Distributed File System (HDFS). Благодаря довольно скромной репликации кластерные машины могут читать этот набор данных параллельно, достигая гораздо более высокой пропускной способности. И при этом целый кластер, собранный из стандартных машин, оказывается дешевле одного высокопроизводительного сервера!

Сказанное выше объясняет, почему кластер Hadoop оказывается эффективнее монолитных систем. А теперь сравним Hadoop с другими архитектурами распределенных систем. Одним из примеров может служить система SETI@home, в которой хранители экрана, работающие на компьютерах, разбросанных по всему земному шару, совместно решают задачу поиска внеземных цивилизаций. В этой системе имеется центральный сервер, на котором хранятся полученные из космоса радиосигналы, и этот сервер по Интернету рассылает клиентским компьютерам задачи поиска аномальных признаков. При таком подходе данные перемещаются на места выполнения вычислений (хранители экрана в настольных компьютерах). По завершении вычислений результаты возвращаются в систему хранения данных.

Hadoop отличается от систем, подобных SETI@home, взглядом на данные. SETI@home основана на повторяющихся операциях пересылки данных между клиентами и серверами. Это хорошо работает для задач, требующих большого объема вычислений, но не годится, когда нужно обработать много данных, поскольку их объем настолько велик, что перемещение становится неэффективным. В основу Hadoop положена идея приближения кода к данным, а не наоборот. Взгляните еще раз на рис. 1.1 – как видите, хранение данных и выполнение вычислений происходит внутри кластера Hadoop. Клиенты

только посылают подлежащие исполнению MapReduce-программы, а они сравнительно невелики (часто порядка нескольких килобайтов). И, что еще важнее, идея приближения кода к данным применяется и в самом кластере Hadoop. Данные распределяются по кластеру, и обработка блока данных по возможности производится на той же машине, где этот блок находится.

Философия приближения кода к данным имеет смысл именно для того типа массивной обработки данных, для которого проектировался Hadoop. Размер исполняемых программ («кода») на много порядков меньше размера данных, поэтому перемещать их проще. Кроме того, на перемещение данных по сети ушло бы куда больше времени, чем на применение к ним вычислений. Так что пусть данные остаются там, где находятся, а на хост-машину мы будем пересылать исполняемый код.

Теперь, когда вы понимаете, какое место Hadoop занимает в ряду распределенных систем, посмотрим, как он соотносится с системами обработки данных, под которыми обычно понимаются СУБД на основе SQL.

1.4. Сравнение СУБД на основе SQL с Hadoop

Итак, Hadoop – это каркас для обработки данных. Так чем же он лучше стандартных реляционных СУБД, являющихся рабочей лошадкой в современных приложениях для обработки данных? Одна из причин заключается в том, что SQL (*структурированный* язык запросов) по природе своей ориентирован на работу со структурированными данными. А многие приложения Hadoop имеют дело с неструктурированными данными, например текстовыми. С этой точки зрения, Hadoop предлагает более общую парадигму, чем SQL.

Что касается структурированных данных, то тут сравнение оказывается более тонким. В принципе, SQL и Hadoop могут дополнять друг друга, поскольку SQL – это язык запросов, который можно реализовать поверх Hadoop, выступающего в роли подсистемы исполнения[3]. Однако на практике под словами «СУБД на основе SQL» принято понимать целый спектр унаследованных технологий, предлагаемых несколькими доминирующими производителями, причем эти технологии оптимизированы для исторически сложившегося на-

[3] На самом деле, в сообществе Hadoop эта тема активно обсуждается, и некоторые из передовых проектов в этой области мы рассмотрим в гл. 11.

бора приложений. Многие из существующих коммерческих СУБД не соответствуют требованиям, под которые проектировался Hadoop.

Помня об этом, попробуем все же провести более детальное сравнение Hadoop с типичными СУБД на основе SQL по некоторым параметрам.

Масштабирование по горизонтали, а не по вертикали

Масштабирование коммерческих реляционных баз данных обходится дорого. По своей природе, они более приспособлены для вертикального масштабирования. Чтобы развернуть более крупную базу данных, вы покупаете более мощный сервер. Вообще, стало уже привычным делом, что производители серверов позиционируют свои высокопроизводительные модели как «серверы класса базы данных». К сожалению, неизбежно настает такой момент, когда набор данных оказывается настолько большим, что достаточно мощного сервера для его обслуживания просто не существует. Но еще важнее тот факт, что высокопроизводительные серверы слишком дороги для многих приложений. Например, машина, которая в четыре раза мощнее стандартного ПК, обойдется гораздо дороже, чем объединение четырех ПК в кластер. Hadoop проектировался в расчете на горизонтально масштабируемую архитектуру, построенную на базе кластера стандартных ПК. Увеличение объема ресурсов сводится к добавлению новых машин в кластер Hadoop. Кластеры Hadoop, состоящие из десятков и сотен машин, считаются стандартными. На самом деле, запускать Hadoop на одном сервере имеет смысл только для разработки.

Пары ключ/значение вместо реляционных таблиц

Основополагающим принципом реляционных СУБД является размещение данных в таблицах, имеющих реляционную структуру, определяемую схемой. Реляционная модель обладает рядом замечательных формальных свойств, но многие современные приложения имеют дело с типами данных, которые плохо в эту модель укладываются. В качестве широко известных примеров упомянем текстовые документы, изображения и XML-файлы. Кроме того, большие наборы данных часто вообще не структурированы или слабо структурированы. В Hadoop в качестве основной единицы данных используется пара ключ/значение, и это достаточно гибкое решение для работы со слабо структурированными типами данных. Исходные данные в Hadoop могут быть представлены в любом формате, но в конечном итоге они преобразуются в пары ключ/значение, к которым и применяются функции обработки.

Функциональное программирование (MapReduce) вместо декларативных запросов (SQL)

По сути своей SQL является высокоуровневым декларативным языком. Запрашивая данные, вы говорите, какой результат хотели бы получить, и предоставляете СУБД решать, как добиться желаемого. В парадигме MapReduce предполагается, что вы сами описываете конкретные шаги обработки данных, что в какой-то мере напоминает порождаемый СУБД план выполнения SQL-запроса В SQL вы формулируете команды-запросы, в MapReduce пишете скрипты и программы. MapReduce допускает более общие способы обработки данных, чем SQL. Например, на основе данных можно строить сложные статистические модели или изменять формат изображений. SQL для таких задач приспособлен плохо.

С другой стороны, многие считают, что при работе с данными, которые хорошо укладываются в реляционные структуры, применение MapReduce выглядит неестественно. Программистам, привыкшим к парадигме SQL, бывает трудно перестроиться и мыслить так, как требует MapReduce. Надеюсь, что примеры и упражнения, приведенные в этой книге, сделают программирование в рамках MapReduce интуитивно более понятным. Отметим, однако, что существует много расширений, позволяющих получить все выгоды масштабируемости Hadoop, программируя в более привычных парадигмах. Есть даже такие, что позволяют писать запросы на SQL-подобном языке и затем автоматически компилируют их в исполняемый код для MapReduce. С некоторыми инструментами такого рода мы познакомимся в главах 10 и 11.

Автономная пакетная обработка вместо оперативных транзакций

Hadoop проектировался для автономной обработки и анализа больших объемов данных. Он не предназначен для произвольного считывания и обновления нескольких записей, то есть не может служить заменой системам оперативной обработки транзакций. На самом деле, сейчас и в обозримом будущем Hadoop лучше всего использовать для работы с хранилищами данных, в которых запись производится однократно, а чтение многократно. В этом смысле он напоминает «хранилища данных» в мире SQL.

Мы рассмотрели в общих чертах, как Hadoop соотносится с распределенными системами и с СУБД на основе SQL. Теперь научимся писать для него программы. Для этого необходимо понимать парадигму MapReduce, лежащую в основе Hadoop.

1.5. Знакомство с MapReduce

Вы, наверное, знакомы с такими моделями обработки данных, как конвейеры и очереди сообщений. Эти модели ориентированы на вполне определенные виды приложений. Наиболее хорошо известны конвейеры операционной системы Unix. Конвейер позволяет *повторно использовать* примитивы обработки; простое соединение существующих модулей в цепочку порождает новые модули. Очереди сообщений обеспечивают *синхронизацию* примитивов обработки. Программист создает задание для обработки данных в виде примитива, который может играть роль производителя или потребителя. Синхронизацию выполнения примитивов берет на себя система.

MapReduce также представляет собой модель обработки данных. Ее основное достоинство состоит в простоте масштабирования при наличии нескольких вычислительных узлов. В модели MapReduce примитивы обработки данных называются *распределителями* (mapper) и *редукторами* (reducer). Декомпозиция приложения обработки данных на распределители и редукторы иногда оказывается нетривиальной задачей. Но, коль скоро приложение представлено в форме, пригодной для MapReduce, его масштабирование на сотни, тысячи и даже десятки тысяч машин, объединенных в кластер, сводится к простому изменению конфигурации. Именно эта простота масштабирования и привлекла внимание многих программистов к модели MapReduce.

Есть много способов сказать MapReduce

Хотя о парадигме MapReduce написано немало, само название разные люди пишут по-разному. В оригинальной работе, опубликованной Google, и в статье в википедии фигурирует название *MapReduce*. Однако на некоторых страницах сайта Google встречается также написание *Map Reduce* (например, на странице http://research.google.com/roundtable/MR.html). На сайте официальной документации по Hadoop можно встретить ссылки на *Map-Reduce Tutorial*. Перейдя по такой ссылке, вы окажетесь на странице руководства, озаглавленного *Hadoop Map/Reduce Tutorial* (http://hadoop.apache.org/core/docs/current/mapred_tutorial.html), в котором объясняется, что такое каркас *Map/Reduce*. Различные варианты написания встречаются и для других компонентов Hadoop, например *NameNode* (*name node*, *namenode* и *namenode*), *DataNode*, *JobTracker* и *TaskTracker*. Для единообразия мы в этой книге всюду будем придерживаться ВерблюжьейНотации (то есть писать *MapReduce*, *NameNode*, *DataNode*, *JobTracker* и *TaskTracker*).

1.5.1. Масштабирование простой программы вручную

Прежде чем переходить к формальному рассмотрению MapReduce, попробуем масштабировать простенькую программу для обработки большого набора данных. Вы увидите, какие проблемы при этом возникают, и, следовательно, сможете по достоинству оценить преимущества использования каркасов типа MapReduce, которые берут на себя черную работу.

Нашей задачей будет подсчитать, сколько раз каждое слово встречается в наборе документов. В этом упражнении набор будет состоять всего из одного документа, содержащего единственного предложение:

Do as I say, not as I do.

(Делай, как я говорю, а не как я делаю.)

Мы должны будем получить счетчики слов, показанные в таблице.

Слово	Счетчик
as	2
do	2
I	2
not	1
say	1

Назовем это упражнение *подсчетом слов*. Если набор документов мал, то для решения задачи достаточно прямолинейной программы. Напишем ее на псевдокоде:

```
define wordCount as Multiset;
for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

Эта программа в цикле обходит все документы. Каждый документ разбивается на слова, и для каждого слова на единицу увеличивается соответствующий ему счетчик в мультимножестве `wordCount`. В конце функция `display()` распечатывает все элементы `wordCount`.

Примечание. Мультимножеством называется множество, в котором для каждого элемента хранится также счетчик. Создаваемая нами структура данных для подсчета слова – канонический пример мультимножества. На практике оно обычно реализуется в виде хеш-таблицы.

Эта программа прекрасно работает, пока интересующий нас набор документов не слишком велик. Но допустим, что нужно построить

фильтр спама, для чего требуется знать, какие слова часто встречаются в миллионах собранных вами «мусорных» почтовых сообщений. Перебор всех этих документов на одном компьютере займет чрезвычайно много времени. Сократить время можно, переписав программу так, чтобы она распределяла работу между несколькими машинами. Каждая машина будет обрабатывать небольшую часть всего набора документов. Когда все документы будут обработаны, наступит черед второй фазы – объединения полученных на каждой машине результатов. Псевдокод первой фазы в случае распределенной обработки выглядит так:

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

А псевдокод второй фазы так:

```
define totalWordCount as Multiset;
for each wordCount received from firstPhase {
    multisetAdd (totalWordCount, wordCount);
}
```

Не сложно, правда? Но есть несколько нюансов, из-за которых эта программа может работать не так, как ожидается. Прежде всего, мы не подумали о требовании к производительности чтения документов. Если все документы хранятся на одном центральном сервере, то узким местом станет пропускная способность этого сервера. Выделение нескольких машин для обработки поможет, но только до определенного предела – пока сервер хранения справляется с нагрузкой. Необходимо еще распределить сами документы между машинами, так чтобы каждая машина обрабатывала только хранящиеся на ней документы. Вот тогда мы сможем устранить узкое место, связанное с наличием центрального сервера хранения. Это еще раз подтверждает сделанное ранее замечание о тесной связи между хранением и обработкой в распределенных приложениях для обработки больших массивов данных.

Еще один недостаток приведенной выше программы заключается в том, что структуры `wordCount` (и `totalWordCount`) хранятся в памяти. Если набор документов велик, то количество уникальных

слов может превысить объем доступной оперативной памяти. В английском языке около миллиона слов, и они вообще-то могут легко уместиться в память iPod, но нашей программе придется иметь дело со словами, которых нет в стандартных словарях английского языка. Например, могут встречаться названия типа *Hadoop*. Мы должны будем подсчитывать и слова, написанные с грамматическими ошибками (например, *exampel*), и по отдельности считать различные формы слова (например, *eat*, *ate*, *eaten* и *eating*). И даже если для хранения уникальных слов в наборе документов памяти хватает, небольшое изменение в постановке задачи может привести к резкому увеличению потребности в памяти. Например, что если вместо слов мы захотим подсчитывать IP-адреса, встречающиеся в файле журнале, или частоту биграмм. В последнем случае нам придется работать с мультимножеством, содержащим миллиарды записей, а это уже превышает объем памяти большинства стандартных компьютеров.

Примечание. Биграммой называется пара соседних слов. В предложении «Do as I say, not as I do» можно выделить следующие биграммы: *Do as*, *as I*, *I say*, *say not*, *not as*, *as I*, *I do*. Аналогично триграммами называются группы из трех соседних слов. Биграммы и триграммы находят важные применения в обработке естественных языков.

Структура `wordCount` может не поместиться в память; нам придется переписать программу так, чтобы эта хеш-таблица хранилась на диске. А для реализации дисковой хеш-таблицы нужно будет написать достаточно сложный и объемный код.

Вспомним еще, что для второй фазы предусмотрена только одна машина, которой предстоит обработать структуры `wordCount`, передаваемые *всеми* машинами, участвующими в первой фазе. Сама по себе обработка одной структуры `wordCount` довольно громоздкая. После выделения достаточно большого количества машин для первой фазы наличие единственной машины для второй фазы становится узким местом. Возникает очевидный вопрос: можно ли переписать вторую фазу, так чтобы она тоже была распределенной и допускала масштабирование за счет добавления новых машин?

Ответ положительный. Чтобы сделать вторую фазу распределенной, необходимо как-то разделить нагрузку между несколькими машинами, работающими независимо. Нужно будет *разбить* таблицу `wordCount`, получившуюся после завершения первой фазы, на части, так чтобы каждая машина, принимающая участие во второй фазе, обрабатывала только одну часть. Предположим, что в нашем примере для

второй фазы выделено 26 машин. Каждой машине мы поручим обрабатывать только часть таблицы `wordCount`, содержащую слова, которые начинаются с одной и той же буквы. Например, машина A во второй фазе будет подсчитывать только слова, начинающиеся с буквы *a*. Чтобы обеспечить возможность такого разбиения на второй фазе, нам придется несколько модифицировать код первой фазы. Вместо одной дисковой хеш-таблицы `wordCount` мы заведем 26 таблиц: `wordCount-a`, `wordCount-b` и т. д. В каждой будут подсчитываться слова, начинающиеся с одной и той же буквы. По завершении первой фазы таблицы `wordCount-a` с каждой из участвующих в первой фазе машин будут посылаться машине A второй фазы, таблицы `wordCount-b` – машине B и т. д. Каждая машина первой фазы будет *тасовать* вычисленные результаты, передавая их машинам второй фазы.

Как видим, программа подсчета слов становится не такой уж простой. Чтобы она могла работать в кластере из нескольких машин, придется добавить целый ряд новых функций:

- хранить файлы на нескольких обрабатывающих машинах (участвующих в первой фазе);
- написать код дисковой хеш-таблицы, чтобы алгоритм обработки не был ограничен размером оперативной памяти;
- разбить на части промежуточные данные (таблицу `wordCount`), полученные на первой фазе;
- перетасовать образовавшиеся части, то есть раздать их машинам второй фазы.

Куча работы для такой простой задачи, как подсчет слов, а ведь мы еще даже не коснулись вопроса об отказоустойчивости. (Что если произойдет отказ машины в середине работы?) Вот потому-то и нужен такой каркас, как Hadoop. Если приложение написано на основе модели MapReduce, то Hadoop сам позаботится обо всей инфраструктуре масштабирования.

1.5.2. Масштабирование той же программы с помощью MapReduce

Процесс исполнения MapReduce-программы разбит на две основные фазы: *распределение* (mapping) и *редукция* (reducing). Для каждой фазы определяется функция обработки данных; эти функции называются соответственно *распределитель* и *редуктор*. На фазе распределения MapReduce принимает входные данные и передает каждый элемент

данных распределителю. На фазе редукции редуктор обрабатывает промежуточные результаты, полученные распределителями, и вычисляет окончательный результат.

Проще говоря, распределитель осуществляет *фильтрацию и преобразование* входных данных в нечто, что редуктор способен *агрегировать*. Бросается в глаза сходство этих двух фаз с тем, что нам предстояло бы разработать для программы подсчета слов. И это сходство не случайно. Каркас MapReduce проектировался с учетом обширного опыта написания масштабируемых распределенных программ. Двухфазная структура наблюдалась при масштабировании многих систем и была поэтому положена в основу каркаса.

Для масштабирования рассмотренной выше распределенной программы подсчета слов нам пришлось бы также написать функции разбиения и тасования. Разбиение (его также называют секционированием) и тасование – общеупотребительные паттерны проектирования, которые идут рука об руку с распределением и редукцией. Но в отличие от последних, разбиение и тасование – весьма общие функции, не зависящие от конкретного приложения. Каркас MapReduce предоставляет их реализацию по умолчанию, достаточную для большинства случаев.

Чтобы распределение, редукция, разбиение и тасование (а также еще несколько пока не упомянутых функций) могли органично работать совместно, необходимо принять некое соглашение о единой структуре обрабатываемых данных. Она должно быть достаточно гибкой и общей, отвечающей потребностям большинства приложений обработки данных, которые мы хотели бы поддержать. В MapReduce в качестве основных примитивов используются *списки* и *пары ключ/значение*. В роли ключей и значений часто выступают целые числа или строки, но это могут быть также фиктивные игнорируемые значения или составные объектные типы. Функции распределения и редукции должны соблюдать описанные ниже ограничения на типы ключей и значений.

В приложениях на основе каркаса MapReduce задаются распределитель и редуктор. Рассмотрим поток данных в целом:

	Вход	Выход
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3, v3>)

1 Входные данные для приложения должны быть представлены в виде списка пар ключ/значение, `list(<k1, v1>)`. Хотя может показаться, что этот формат слишком неопределенный, на практике обычно проблем не возникает. Входные

данные при обработке нескольких файлов обычно имеют вид `list(<String filename, String file_content>)`. Входные данные при обработке одного большого файла, например журнала, имеют вид `list(<Integer line_number, String log_event>)`.

2 Каждая пара ключ/значение из списка обрабатывается путем вызова функции map распределителя. На практике распределитель часто игнорирует ключ `k1`. Распределитель преобразует каждую пару `<k1, v1>` в список пар `<k2, v2>`. Специфика преобразования как раз и определяет семантику MapReduce-программы. Отметим, что пары ключ/значение обрабатываются в случайном порядке. Преобразование должно быть локальным, то есть его результат должен зависеть только от одной пары ключ/значение.

В программе подсчета слов распределитель принимает пару `<String filename, String file_content>` и попросту игнорирует ключ `filename`. Он мог бы выводить список пар `<String word, Integer count>`, но можно поступить и проще. Мы знаем, что на последующей стадии счетчики все равно будут агрегироваться, поэтому разумно выводить список пар вида `<String word, Integer 1>` с повторяющимися записями, отложив агрегирование до следующей стадии. Таким образом, в результирующем списке может либо один раз встречаться пара `<"foo", 3>`, либо три раза пара `<"foo", 1>`. Как вскоре станет ясно, последний вариант запрограммировать гораздо проще. Первый вариант может дать выигрыш в производительности, однако отложим подобную оптимизацию до момента, когда лучше познакомимся с каркасом MapReduce.

3 Результаты работы всех распределителей (концептуально) агрегируются в один гигантский список пар `<k2, v2>`. Все пары с одним и тем же ключом `k2` объединяются в новую пару `<k2, list(v2)>`. Каркас поручает редуктору обработать каждую такую агрегированную пару ключ/значение по отдельности. В программе подсчета слов на выходе распределителя для одного документа может получиться список, в котором пара `<"foo", 1>` встречается три раза, а для другого документа – список, в котором та же пара встречается дважды. Редуктор увидит агрегированную пару `<"foo", list(1,1,1,1,1)>`. В программе подсчета слов на выходе редуктора мы получим пару `<"foo", 5>`, в которой отражено, сколько всего раз сло-

во «foo» встретилось в наборе документов. Все редукторы работают с разными словами. Каркас MapReduce автоматически собирает все пары <k3, v3> и записывает их в один или несколько файлов. Отметим, что в программе подсчета слов типы данных k2 и k3 совпадают, как и типы v2 и v3. Но в других задачах может быть иначе.

Теперь перепишем программу подсчета слов на MapReduce, чтобы понять, как все части сопрягаются между собой. Псевдокод приведен в листинге 1.1.

Листинг 1.1.

```
map(String filename, String document) {
   List<String> T = tokenize(document);
   for each token in T {
       emit ((String)token, (Integer) 1);
   }
}
reduce(String token, List<Integer> values) {
   Integer sum = 0;
   for each value in values {
       sum = sum + value;
   }
   emit ((String)token, (Integer) sum);
}
```

Выше мы сказали, что результатами функций map и reduce являются списки. Как видно из псевдокода, на практике используется специальная функция каркаса emit(), которая генерирует элементы списка по одному. Функция emit() освобождает программиста от необходимости управлять большим списком.

Этот код похож на тот, что мы написали в разделе 1.5.1, но на этот раз он реально масштабируется. Hadoop действительно упрощает создание масштабируемых распределенных программ, не правда ли? А теперь настало время превратить этот псевдокод в программу для Hadoop.

1.6. Подсчет слов с помощью Hadoop – ваша первая программа

Итак, вы уже понимаете смысл каркаса Hadoop на основе MapReduce, теперь запустим его. В этой главе мы будем работать с Hadoop на одной-единственной машине, это может быть ваш настольный ПК или

ноутбук. А в следующей главе мы продемонстрируем запуск Hadoop в кластере машин, как то обычно делается при практическом развертывании. Запускать Hadoop на одной машине имеет смысл главным образом на этапе разработки.

Официальной платформой для разработки и промышленной эксплуатации Hadoop является Linux, хотя в качестве платформы разработки поддерживается и Windows. Если вы работаете на компьютере под управлением Windows, то нужно будет установить систему cygwin (http://www-cygwin.com/), которая даст оболочку и позволит запускать написанные для Unix скрипты.

Примечание. Есть много сообщений об успешном запуске Hadoop в режиме разработки и в других вариантах Unix, в частности в системах Solaris и Mac OS X. Более того, складывается впечатление, что многие разработчики Hadoop используют ноутбук MacBook Pro, поскольку они то и дело мелькают на конференциях по Hadoop и на встречах групп пользователей.

Для запуска Hadoop необходима среда Java (версии не ниже 1.6). Пользователи Mac должны запрашивать ее у компании Apple. Для других операционных систем последнюю версию комплекта JDK можно скачать с сайта компании Sun по адресу http://java.sun.com/javase/downloads/index.jsp. Установите его и запомните корневой каталог инсталляции Java, поскольку он потребуется в дальнейшем.

Чтобы установить Hadoop, сначала скачайте последнюю стабильную версию со страницы http://hadoop.apache.org/core/releases.html. Распаковав дистрибутив, измените скрипт conf/hadoopenv.sh, присвоив переменной JAVA_HOME путь к корню инсталляции Java. Например, в Mac OS X следует заменить строку

```
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
```

такой

```
export JAVA_HOME=/Library/Java/Home
```

Этим скриптом Hadoop вы будете пользоваться очень часто. Запустите его без аргументов, чтобы ознакомиться с документацией о порядке запуска:

```
bin/hadoop
```

Будет напечатано следующее[4]:

```
Порядок запуска: hadoop [--config confdir] COMMAND
где COMMAND может принимать следующие значения:
  namenode -format     разметить файловую систему DFS
```

[4] Справка печатается на английском языке, но для удобства читателя она переведена. *Прим. перев.*

```
  secondarynamenode    запустить демон Secondary NameNode на DFS
  namenode             запустить демон NameNode на DFS
  datanode             запустить демон DataNode на DFS
  dfsadmin             запустить административный клиент DFS
  fsck                 запустить утилиту проверки файловой
                       системы DFS
  fs                   запустить обобщенный пользовательский
                       клиент файловой системы
  balancer             запустить утилиту балансирования кластера
  jobtracker           запустить демон MapReduce Job Tracker
  pipes                запустить задание Pipes
  tasktracker          запустить демон MapReduce Task Tracker
  job                  управление заданиями MapReduce
  version              напечатать номер версии
  jar <jar>            выполнить jar-файл
  distcp <srcurl> <desturl>   копировать файлы или каталоги
                              (рекурсивно)
  archive -archiveName NAME <src>* <dest>   создать архив hadoop
  daemonlog            получить/установить уровень протоколирования
                       для каждого демона
или
  CLASSNAME            выполнить класс с именем CLASSNAME
Большинство команд печатают собственную справку при запуске без
параметров.
```

По ходу изложения мы будем рассматривать различные команды Hadoop. А пока достаточно знать, что для запуска программы для Hadoop (на языке Java) предназначена команда `bin/hadoop jar <jar>`. Как следует из названия команды, программы для Hadoop, написанные на Java, должны быть представлены в виде jar-архивов.

К счастью, нам не нужно сразу приступать к написанию программы для Hadoop; в дистрибутиве уже есть несколько примеров, которыми можно воспользоваться. Следующая команда показывает, что находится в файле examples.jar:

```
bin/hadoop jar hadoop-*-examples.jar
```

Из распечатки будет видно, что вместе с Hadoop поставляется десяток программ-примеров, и одна из них – программа подсчета узлов, которая называется... `wordcount`! Важные (внутренние) классы, составляющие эту программу, показаны в листинге 1.2. Мы увидим, как в этой программе реализованы функции map и reduce для подсчета слов, псевдокод которых был представлен в листинге 1.1. Мы модифицируем эту программу, чтобы понять, как можно изменить ее поведение. А пока примем, что она работает в соответствии с ожиданиями, и просто познакомимся с механизмом исполнения программ для Hadoop.

При запуске без аргументов программа `wordcount` выведет справку о порядке запуска:

```
bin/hadoop jar hadoop-*-examples.jar wordcount
```

а именно список допустимых аргументов:

```
wordcount [-m <maps>] [-r <reduces>] <input> <output>
```

Обязательными параметрами являются входной каталог (`<input>`), где находятся подлежащие анализу текстовые документы, и выходной каталог (`<output>`), куда программа поместит результаты. Перед тем как запускать `wordcount`, необходимо создать входной каталог:

```
mkdir input
```

и поместить в него какие-нибудь документы. Можете поместить в этот каталог произвольные текстовые документы. Для демонстрации возьмем текстовый вариант доклада президента конгрессу о положении в стране за 2002 год, находящийся по адресу http://www.gpoaccess.gov/sou/. Вычислим счетчики слов и выведем результаты:

```
bin/hadoop jar hadoop-*-examples.jar wordcount input output
more output/*
```

Будут выведены счетчики всех слов, встречающихся в документе, в алфавитном порядке. Неплохо, учитывая, что вы пока не написали ни строчки кода! Отметим, однако, ряд недостатков программы `wordcount`, включенной в дистрибутив. Разбиение на слова производится только по пробелам, но не по знакам препинания, из-за чего *States*, *States.* и *States:* оказываются разными словами. То же самое касается регистра букв, то есть *States* и *states* – тоже разные слова. Наконец, мы не хотели бы включать слова, которые встречаются в документе всего один или два раза.

К счастью, код `wordcount` открыт и находится в подкаталоге src/examples/org/apache/hadoop/examples/WordCount.java инсталляционного каталога. Мы можем подстроить его под свои требования. Создадим структуру каталогов, подходящую для экспериментов, и скопируем программу.

```
mkdir playground
mkdir playground/src
mkdir playground/classes
cp src/examples/org/apache/hadoop/examples/WordCount.java
➥ playground/src/WordCount.java
```

Прежде чем вносить изменения, откомпилируем и выполним только что созданную копию:

```
javac -classpath hadoop-*-core.jar -d playground/classes
➥ playground/src/WordCount.java
jar -cvf playground/wordcount.jar -C playground/classes/ .
```

Перед каждым запуском команды Hadoop необходимо удалять выходной каталог, поскольку он создается автоматически.

```
bin/hadoop jar playground/wordcount.jar
➥ org.apache.hadoop.examples.WordCount input output
```

Взгляните еще раз на файлы, оказавшиеся в выходном каталоге. Поскольку мы не изменяли код программы, то должен получиться такой же результат, как и раньше. Мы всего лишь откомпилировали свою копию вместо запуска заранее откомпилированной версии.

Теперь все готово к модификации программы `wordcount` с целью добавления новой функциональности. В листинге 1.2 приведена часть файла *WordCount.java*. Комментарии и вспомогательный код опущены.

Листинг 1.2.

```
public class WordCount extends Configured implements Tool {

  public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
```

1 **Разбиение на лексемы по пробелам**

2 **Привести лексему к типу объекта Text**

```
                        Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }
  ...
}
```

❸ Вывести счетчики для всех лексем

Основное функциональное различие между `WordCount.java` и нашим псевдокодом MapReduce-программы заключается в том, что в `WordCount.java` функция `map()` обрабатывает по одной строке текста за раз, тогда как наш псевдокод обрабатывает сразу весь документ. Но при взгляде на текст `WordCount.java` это различие незаметно, потому что такой режим принят в Hadoop по умолчанию.

Код в листинге 1.2 практически идентичен псевдокоду в листинге 1.1, хотя из-за синтаксиса Java он получился несколько длиннее. Функции map и reduce скрыты во внутренних классах `WordCount`. Вы, наверное, обратили внимание на использование специальных классов `LongWritable`, `IntWritable` и `Text` вместо более привычных классов Java `Long`, `Integer` и `String`. Пока будем считать это деталями реализации. В новые классы встроены дополнительные средства сериализации, необходимые для работы Hadoop.

Нетрудно понять, что именно требуется изменить. Мы видим ❶, что в `WordCount` используется класс `StringTokenizer` с настройками по умолчанию, при которых разбиение на лексемы производится по пустым символам (пробелы, знаки табуляции, возврата каретки и перевода строки). Если мы хотим игнорировать также знаки препинания, то должны добавить их в список символов-ограничителей, задаваемый в конструкторе:

```
StringTokenizer itr = new StringTokenizer(line, " \t\n\r\f,.:;?![]'");
```

В цикле обхода лексем мы преобразуем каждую лексему в объект типа `Text` ❷. (Напомним, что в Hadoop специальный класс `Text` используется вместо `String`.) Мы хотим игнорировать регистр букв при подсчете слов, поэтому перед преобразованием в объект `Text` должны перевести слово в нижний регистр:

```
word.set(itr.nextToken().toLowerCase());
```

Наконец, нас интересуют только слова, встречающиеся в документе более четырех раз. Поэтому мы изменим ❸, так чтобы слово

включалось в выходной результат лишь при выполнении этого условия. (Это принятый в Hadoop эквивалент функции `emit()`, которую мы употребили в псевдокоде.)

```
if (sum > 4) output.collect(key, new IntWritable(sum));
```

Внеся эти три изменения, снова скомпилируйте и запустите программу. Результаты показаны в табл. 1.1.

Таблица 1.1. Слова, встречающиеся более четырех раз в тексте доклада президента конгрессу о положении в стране за 2002 год

11th (5)	citizens (9)	its (6)	over (6)	to (123)
a (69)	congress (10)	jobs (11)	own (5)	together (5)
about (5)	corps (6)	join (7)	page (7)	tonight (5)
act (7)	country (10)	know (6)	people (12)	training (5)
afghanistan (10)	destruction (5)	last (6)	protect (5)	united (6)
all (10)	do (6)	lives (6)	regime (5)	us (6)
allies (8)	every (8)	long (5)	regimes (6)	want (5)
also (5)	evil (5)	make (7)	security (19)	war (12)
America (33)	for (27)	many (5)	september (5)	was (11)
American (15)	free (6)	more (11)	so (12)	we (76)
americans (8)	freedom (10)	most (5)	some (6)	we've (5)
an (7)	from (15)	must (18)	states (9)	weapons (12)
and (210)	good (13)	my (13)	tax (7)	were (7)
are (17)	great (8)	nation (11)	terror (13)	while (5)
as (18)	has (12)	need (7)	terrorist (12)	who (18)
ask (5)	have (32)	never (7)	terrorists (10)	will (49)
at (16)	health (5)	new (13)	than (6)	with (22)
be (23)	help (7)	no (7)	that (29)	women (5)
been (8)	home (5)	not (15)	the (184)	work (7)
best (6)	homeland (7)	now (10)	their (17)	workers (5)
budget (7)	hope (5)	of (130)	them (8)	world (17)
but (7)	i (29)	on (32)	these (18)	would (5)
by (13)	if (8)	one (5)	they (12)	yet (8)
camps (8)	in (79)	opportunity (5)	this (28)	you (12)
can (7)	is (44)	or (8)	thousands (5)	
children (6)	it (21)	our (78)	time (7)	

Как видим, в тексте оказалось 128 слов, встречающихся более 4 раз. Многие из них часто встречаются практически в любом англоязычном тексте. Например, *a* (69), *and* (210), *i* (29), *in* (79), *the* (184) и целый ряд других. Кроме того, мы видим слова, относящиеся к проблемам, с которым столкнулись США в то время: *terror* (13), *terrorist* (12), *terrorists* (10), *security* (19), *weapons* (12), *destruction* (5), *afghanistan* (10), *freedom* (10), *jobs* (11), *budget* (7) и другие.

1.7. История Hadoop

Первоначально Hadoop был подпроектом Nutch, который в свою очередь был подпроектом Apache Lucene. Основал все три проекта, каждый из которых является логическим продолжением предыдущего, Дуг Каттинг.

Lucene – это полнофункциональная библиотека для текстового индексирования и поиска. Lucene позволяет разработчику без труда добавить механизм поиска по набору текстовых документов. На базе Lucene уже созданы программы поиска по файлам, хранящимся в персональном компьютере, по корпоративным документам и ряд предметно-ориентированных поисковых систем. Nutch – самое амбициозное расширение Lucene. Это попытка построить полноценную систему поиска в веб с Lucene в качестве основного компонента. В Nutch имеются синтаксический анализатор HTML, робот для обхода веб, база данных, в которой хранится граф ссылок, и другие компоненты, необходимые для поисковой системы в веб. Дуг Каттинг мечтает, что Nutch станет открытой демократической альтернативой закрытым коммерческим технологиям типа Google.

Помимо добавления различных компонентов, в частности робота и синтаксического анализатора, поисковая система для веб отличается от простой системы поиска в документах масштабом. Если Lucene ориентирована на индексирование миллионов документов, то Nutch должна уметь обрабатывать миллиарды веб-страниц, не становясь при этом непомерно дорогой в эксплуатации. Nutch должна будет работать в распределенном кластере, собранном из стандартных машин. Перед командой разработчиков Nutch стоит головоломная задача – разрешить проблемы масштабируемости программным способом. Nutch нуждается в программном слое, который отвечал бы за распределенную обработку, резервирование, автоматический переход на резервный ресурс в случае отказа и балансирование нагрузки. Это далеко не тривиальные проблемы.

В 2004 году компания Google опубликовала две статьи, в которых описывалась файловая система Google File System (GFS) и каркас MapReduce. Google заявляла, что применяет обе эти технологии для масштабирования собственной поисковой системы. Дуг Каттинг тут же углядел возможность применить эти технологии в Nutch, и его команда реализовала новый каркас и перенесла на него Nutch. В результате масштабируемость Nutch резко возросла. Стало возможно обрабатывать несколько сотен миллионов веб-страниц при работе на кластере из десятков узлов. Дуг понял, что для углубленной разработки обеих технологий, необходимых для перехода к масштабу веб, нужен отдельный проект. Так и родился Hadoop. В январе 2006 года корпорация Yahoo! приняла Дуга на работу в должности руководителя группы, занимающейся только совершенствованием Hadoop в формате проекта с открытым исходным кодом.

Спустя два года Hadoop добился статуса «Проект Apache верхнего уровня». Позже, 19 февраля 2008 года корпорация Yahoo! объявила, что Hadoop, работающий в кластере из 10000 с лишним узлов под управлением Linux, стал основой ее промышленной системы индексирования веб (http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html). Hadoop таки вышел на уровень веб!

Что означают названия?

Выбирая названия для программных проектов, Дуг Каттинг, похоже, ищет идеи в собственной семье. *Lucene* – среднее имя его жены и первое имя ее бабушки по матери. Его сын, когда был еще совсем маленьким, называл словом *Nutch* любую еду, а позже придумал для своего желтого набивного слоника прозвище *Hadoop*. Дуг говорил, что «искал название, которое еще не засветилось в веб, как предметной области, и не было бы чьим-то торговым наименованием. Поэтому я рассматривал разные слова, которые употреблялись в моем узком кругу общения и более нигде. Дети – большие мастера по придумыванию новых слов».

1.8. Резюме

Hadoop – это гибкий инструмент, позволяющий пользователям задействовать всю мощь распределенных вычислений. За счет использования распределенной системы хранения данных и перемещения кода, а не данных, Hadoop удается избежать дорогостоящей пересылки при работе с большими наборами данных. Кроме того, избыточность хранения данных позволяет Hadoop восстанавливаться после

отказа одного узла. Вы видели, как просто создавать MapReduce-программы для Hadoop. Но не менее важно и то, о чем вам не придется беспокоиться, – секционирование данных, распределение задач между узлами, вопрос об организации взаимодействия между узлами. Все это Hadoop делает самостоятельно, давая вам возможность сосредоточиться на самом главном – ваших данных и операциях над ними.

В следующей главе мы будем более подробно говорить о внутреннем устройстве Hadoop и о настройке работоспособного кластера Hadoop.

1.9. Ресурсы

Официальный сайт Hadoop находится по адресу http://hadoop.apache.org/.

Заслуживают внимания оригинальные статьи о файловой системе Google File System и каркасе MapReduce. Оцените дизайн и архитектуру:

- *The Google File System* – http://labs.google.com/papers/gfs.html.
- *MapReduce: Simplified Data Processing on Large Clusters* – http://labs.google.com/papers/mapreduce.html.

ГЛАВА 2. Запуск Hadoop

В этой главе:

- Архитектурные компоненты Hadoop.
- Настройка Hadoop и три режима его работы: автономный, псевдораспределенный и полностью распределенный.
- Веб-инструменты для мониторинга настройки Hadoop.

Эта глава является справочником по настройке Hadoop. Если вы работаете с уже настроенным кластером Hadoop, то можете ее пропустить. Возможно, вы захотите получить сведения, достаточные для настройки собственной машины разработки, но детально разбираться в том, как конфигурируются связи между узлами и координируется их работа, необязательно.

Рассмотрев в разделе 2.1 физические компоненты Hadoop, мы перейдем к настройке кластера – этому посвящены разделы 2.2 и 2.3. В разделе 2.3 мы обсудим три режима работы Hadoop. Из раздела 2.4 вы узнаете о веб-инструментах для мониторинга кластера.

2.1. Структурные элементы Hadoop

В предыдущей главе мы обсудили концепции распределенного хранения данных и распределенных вычислений. Теперь посмотрим, как эти идеи реализуются в Hadoop. В полностью сконфигурированном кластере под «запуском Hadoop» понимается запуск набора демонов, или резидентных программ на различных сетевых серверах. Каждый демон играет свою роль; некоторые запускаются только на одном сер-

вере, другие - на нескольких. Существуют следующие демоны:

- NameNode;
- DataNode;
- Secondary NameNode;
- JobTracker;
- TaskTracker.

Мы поговорим о роли каждого из них в каркасе Hadoop.

2.1.1. NameNode

Начнем с, пожалуй, самого важного из демонов Hadoop – NameNode. В Hadoop применяется архитектура главный/подчиненный как для распределенного хранения, так и для распределенных вычислений. Распределенная система хранения называется *файловой системой Hadoop*, или HDFS (Hadoop File System). NameNode представляет собой главный демон HDFS, который распределяет низкоуровневые задачи ввода/вывода между подчиненными демонами DataNode. NameNode – это диспетчер HDFS; он ведет учет разбиению файлов на блоки, хранит информацию о том, на каких узлах эти блоки находятся, и следит за общим состоянием распределенной файловой системы.

Для демона NameNode характерно высокое потребление памяти и большой объем операций ввода/вывода. Поэтому на сервере, где работает NameNode, обычно не хранятся пользовательские данные и не выполняются вычисления, связанные с MapReduce-программами, что позволяет снизить нагрузку на машину. Это означает, что сервер NameNode не дублируется, как DataNode или TaskTracker. Таким образом, важность NameNode имеет оборотную сторону – его отказ приводит к отказу системы в целом. При отказе – программном или аппаратном – узла, на котором работает любой другой демон, кластер Hadoop, скорее всего, успешно продолжит работу, а отказавший узел можно будет быстро перезапустить. Для NameNode дело обстоит иначе.

2.1.2. DataNode

На любой подчиненной машине в кластере работает демон DataNode, на который возложена основная работа распределенной файловой системы – считывание и запись блоков HDFS в физические файлы,

находящиеся в локальной файловой системе. Когда клиент хочет прочитать или записать HDFS-файл, демон NameNode сообщает ему, на каком узле DataNode находится каждый блок файла. Далее клиент напрямую общается с демонами DataNode, работая с локальными файлами, соответствующими блокам. Кроме того, демон DataNode может взаимодействовать с другими таким же демонами, осуществляя репликацию блоков данных для обеспечения резервирования.

На рис. 2.1 показаны роли демонов NameNode и DataNode. Здесь мы видим два файла: /user/chuck/data1 и /user/james/data2. В файле data1 размещены три блока, обозначенные 1, 2 и 3, а в файле data2 – блоки 4 и 5. Содержимое файлов распределено между узлами DataNode. На рисунке для каждого блока имеются три реплики. Например, реплики блока 1 (из файла data1) хранятся на трех правых узлах DataNode. Таким образом, если любой узел DataNode выйдет из строя или станет недоступен по сети, то файлы все равно можно будет прочесть.

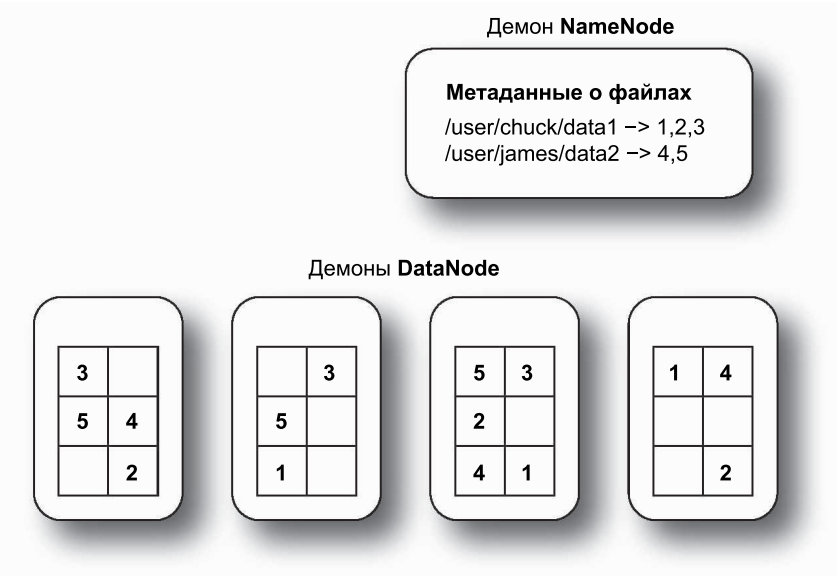


Рис. 2.1. Взаимодействие демонов NameNode и DataNode в HDFS. NameNode хранит метаданные о файлах: какие файлы присутствуют в системе и как каждый файл разбит на блоки. Демоны DataNode обеспечивают резервирование блоков и периодически передают NameNode информацию о состоянии, чтобы метаданные оставались актуальными.

Демоны DataNode периодически передают NameNode информацию о состоянии. На стадии инициализации каждый DataNode сообщает NameNode о блоках, которые хранятся на нем в настоящий момент. Впоследствии демоны DataNode продолжают взаимодействовать с NameNode, передавая информацию о локальных изменениях и получая инструкции о создании, перемещении и удалении блоков с локального диска.

2.1.3. Secondary NameNode

Secondary NameNode (SNN) – это вспомогательный демон, который занимается мониторингом состояния кластера HDFS. В кластере может присутствовать только один SNN, и обычно для него выделяется специальный сервер. На этом сервере не может работать ни DataNode, ни TaskTracker. От NameNode демон SNN отличается тем, что не получает и не протоколирует информацию об изменениях HDFS в режиме реального времени. Вместо этого он взаимодействует с NameNode с целью создания мгновенных снимков метаданных HDFS; интервал времени между операциями создания снимков определяется конфигурацией кластера.

Выше уже отмечалось, что NameNode является точкой общего отказа кластера Hadoop, а снимки, создаваемые SNN, позволяют уменьшить время простоя и свести к минимуму потерю данных. Тем не менее, отказ NameNode требует вмешательства со стороны человека – необходимо переконфигурировать кластер так, чтобы узел SNN стал основным узлом NameNode. Процедуру восстановления мы обсудим в главе 8 при рассмотрении рекомендаций по управлению кластером.

2.1.4. JobTracker

Демон JobTracker – это посредник между Hadoop и приложением. Когда вы передаете свой код кластеру, JobTracker строит план выполнения, то есть определяет, какие файлы обрабатывать, назначает узлы различным задачам и следит за ходом исполнения этих задач. Если задача завершится неудачно, то JobTracker автоматически перезапустит ее, возможно на другом узле; количество таких попыток определено в конфигурации кластера.

В кластере Hadoop может быть только один демон JobTracker. Обычно он работает на отдельном сервере, играющем роль главного узла кластера.

2.1.5. TaskTracker

Демоны вычислений, как и демоны хранения, также построены на базе архитектуры главный/подчиненный: JobTracker – это главный демон, организующий общее выполнение задачи MapReduce, а демоны TaskTracker управляют исполнением отдельных заданий на подчиненных узлах. Это взаимодействие показано на рис. 2.2.

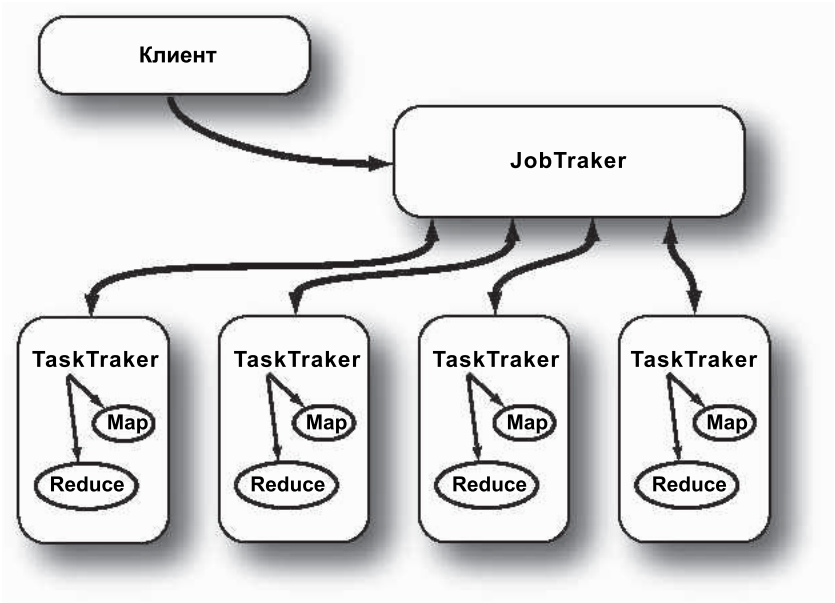


Рис. 2.2. Взаимодействие JobTracker и TaskTracker. После того как клиент вызвал JobTracker, чтобы начать задачу обработки, JobTracker разбивает задачу на части и назначает задания map и reduce различным узлам TaskTracker в кластере.

Каждый TaskTracker отвечает за исполнение отдельных заданий, которые ему назначил JobTracker. Хотя в одном подчиненном узле может работать только один демон TaskTracker, ему разрешено запускать несколько виртуальных машин Java (JVM) для параллельного исполнения нескольких заданий распределения или редукции.

Одна из обязанностей TaskTracker – периодически докладывать о себе демону JobTracker. Если JobTracker в течение заданного промежутка времени не получает контрольного сообщения от TaskTracker, то он предполагает, что узел TaskTracker вышел из строя, и передает ранее назначенные ему задания другим узлам кластера.

Рассмотрев все демоны Hadoop, мы можем изобразить топологию типичного кластера Hadoop (рис. 2.3).

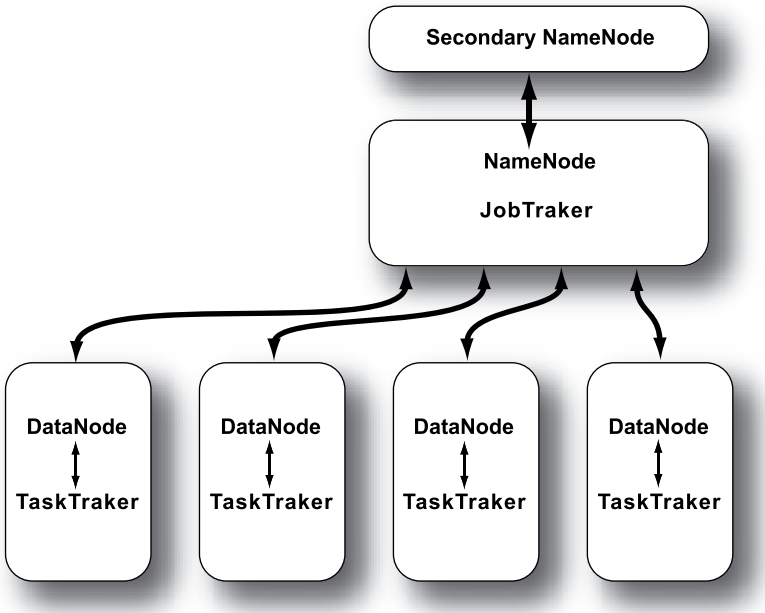


Рис. 2.3. Топология типичного кластера Hadoop. Это архитектура главный/подчиненный, в которой NameNode и JobTracker – главные узлы, а DataNode и TaskTracker – подчиненные.

В этой топологии имеется главный узел, на котором работают демоны NameNode и JobTracker, и отдельный узел с демоном SNN на случай отказа главного узла. Для небольших кластеров SNN может располагаться на каком-либо подчиненном узле. Но в большом кластере размещайте NameNode и JobTracker на разных машинах. На каждой из подчиненных машин работает по одному экземпляру DataNode и TaskTracker, чтобы задания можно было исполнять на том же узле, где находятся обрабатываемые ими данные.

Далее мы займемся настройкой такого кластера Hadoop, для чего сначала определим главный узел и каналы управления между узлами. Если у вас уже есть сконфигурированный кластер Hadoop, то можете пропустить следующий раздел, где рассказывается о настройке каналов между узлами на базе протокола Secure Shell (SSH). Существуют также два способа запустить Hadoop на одной машине: автономный

и псевдораспределенный режим. Они полезны для разработки. Конфигурирование этих режимов, а также стандартного кластера (полностью распределенный режим) рассматривается в разделе 2.3.

2.2. Настройка SSH для кластера Hadoop

При настройке кластера Hadoop необходимо назначить какой-нибудь узел в качестве главного. Как показано на рис. 2.3, на соответствующем сервере обычно работают демоны NameNode и JobTracker. Он также служит отправной точкой для активации демонов DataNode и TaskTracker на всех подчиненных узлах и последующего взаимодействия с ними. Поэтому необходимы средства удаленного доступа со стороны главного узла ко всем остальным.

Для этой цели в Hadoop используется протокол SSH без парольной фразы. В SSH применяются стандартные методы асимметричной криптографии, то есть создаются два ключа для аутентификации пользователей: открытый и закрытый. Открытый ключ хранится локально на каждом узле кластера, а закрытым ключом главный узел шифрует информацию при доступе к удаленной машине. Имея оба ключа, целевая машина может аутентифицировать попытку удаленного входа.

2.2.1. Определение общей учетной записи

До сих пор мы говорили просто о доступе к одному узлу со стороны другого; на самом деле речь идет о доступе учетной записи на одном узле к учетной записи на другом узле. В случае Hadoop обе учетные записи должны называться одинаково на всех узлах (в этой книге используется учетная запись пользователя hadoop), и безопасности ради мы рекомендуем создавать эту учетную запись на уровне пользователя. Она предназначена исключительно для управления кластером Hadoop. После того как демоны запущены, MapReduce-задачи можно исполнять от имени других учетных записей.

2.2.2. Проверка правильности установки SSH

Первым делом следует проверить, что на всех узлах установлен SSH. Это легко сделать с помощью команды Unix `which`:

```
[hadoop-user@master]$ which ssh
/usr/bin/ssh
```

```
[hadoop-user@master]$ which sshd
/usr/bin/sshd
[hadoop-user@master]$ which ssh-keygen
/usr/bin/ssh-keygen
```

Если в ответ вы получаете сообщение об ошибке, например:

```
/usr/bin/which: no ssh in (/usr/bin:/bin:/usr/sbin...
```

то установите OpenSSH (www.openssh.com) с помощью менеджера пакетов Linux или скачав исходный код напрямую. (А еще лучше, попросите об этом системного администратора.)

2.2.3. Генерация пары ключей

Убедившись, что SSH корректно установлен на всех узлах кластера, выполним команду `sshkeygen` на главном узле, чтобы сгенерировать пару RSA-ключей. Не задавайте парольную фразу, иначе вам придется вручную вводить ее каждый раз, как главный узел попытается обратиться к другому узлу.

```
[hadoop-user@master]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop-user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hadoop-user/.ssh/id_rsa.
Your public key has been saved in /home/hadoop-user/.ssh/id_rsa.pub.
```

Созданный открытый ключ будет иметь вид:

```
[hadoop-user@master]$ more /home/hadoop-user/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA1WS3RG8LrZH4zL2/1oYgkV1OmVclQ2OO5vRi0Nd
K51Sy3wWpBVHx82F3x3ddoZQjBK3uvLMaDhXvncJG31JPfU7CTAfmtgINYv0kdUbDJq4TKG/fuO5q
J9CqHV71thN2M310gcJ0Y9YCN6grmsiWb2iMcXpy2pqg8UM3ZKApyIPx99O1vREWm+4moFTg
YwIl5be23ZCyxNjgZFWk5MRlT1p1TxB68jqNbPQtU7fIafS7Sasy7h4eyIy7cbLh8x0/V4/mcQsY
5dvReitNvFVte6onl8YdmnMpAh6nwCvog3UeWWJjVZTEBFkTZuV1i9HeYHxpm1wAzcnf7az78jT
IRQ== hadoop-user@master
```

и теперь мы должны распространить его на все узлы кластеры.

2.2.4. Распространение открытого ключа и проверка возможности входа в систему

Операция копирования открытого ключа на все подчиненные и главные узлы несколько утомительна, но проделать ее необходимо:

```
[hadoop-user@master]$ scp ~/.ssh/id_rsa.pub hadoop-user@target:~/master_key
```

Зайдите на целевой узел локально и сделайте главный ключ ав-

торизованным (или добавьте в список авторизованных ключей, если ранее уже были определены другие ключи):

```
[hadoop-user@target]$ mkdir ~/.ssh
[hadoop-user@target]$ chmod 700 ~/.ssh
[hadoop-user@target]$ mv ~/master_key ~/.ssh/authorized_keys
[hadoop-user@target]$ chmod 600 ~/.ssh/authorized_keys
```

После генерации и копирования ключа можно проверить правильность конфигурации, попытавшись зайти на целевой узел с главного:

```
[hadoop-user@master]$ ssh target
The authenticity of host 'target (xxx.xxx.xxx.xxx)' can't be established.
RSA key fingerprint is 72:31:d8:1b:11:36:43:52:56:11:77:a4:ec:82:03:1d.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'target' (RSA) to the list of known hosts.
Last login: Sun Jan 4 15:32:22 2009 from master
```

После того как подлинность целевого узла подтверждена, главный узел не будет задавать вопрос при последующих попытках входа.

```
[hadoop-user@master]$ ssh target
Last login: Sun Jan 4 15:32:49 2009 from master
```

Итак, основа для работы Hadoop в вашем собственном кластере заложена. Теперь обсудим различные режимы Hadoop, которые могут пригодиться при работе над вашими проектами.

2.3. Запуск Hadoop

Перед тем как запускать Hadoop, необходимо еще кое-что настроить. Познакомимся поближе с конфигурационным каталогом Hadoop:

```
[hadoop-user@master]$ cd $HADOOP_HOME
[hadoop-user@master]$ ls -l conf/
total 100
-rw-rw-r-- 1 hadoop-user hadoop 2065 Dec 1 10:07 capacity-scheduler.xml
-rw-rw-r-- 1 hadoop-user hadoop  535 Dec 1 10:07 configuration.xsl
-rw-rw-r-- 1 hadoop-user hadoop 49456 Dec 1 10:07 hadoop-default.xml
-rwxrwxr-x 1 hadoop-user hadoop 2314 Jan 8 17:01 hadoop-env.sh
-rw-rw-r-- 1 hadoop-user hadoop 2234 Jan 2 15:29 hadoop-site.xml
-rw-rw-r-- 1 hadoop-user hadoop 2815 Dec 1 10:07 log4j.properties
-rw-rw-r-- 1 hadoop-user hadoop  28 Jan 2 15:29 masters
-rw-rw-r-- 1 hadoop-user hadoop  84 Jan 2 15:29 slaves
-rw-rw-r-- 1 hadoop-user hadoop  401 Dec 1 10:07 sslinfo.xml.example
```

Прежде всего необходимо задать местоположение Java на всех узлах, включая главный. В файле hadoop-env.sh присвойте переменной окружения JAVA_HOME значение, указывающее на инсталяци-

онный каталог Java. На наших серверах она определена следующим образом:

```
export JAVA_HOME=/usr/share/jdk
```

(Если вы выполняли примеры из главы 1, то это уже сделано.)

В файле hadoop-env.sh есть и другие переменные, описывающие окружение Hadoop, но `JAVA_HOME` – единственная, которую нужно задать перед началом работы. Значения других переменных, подразумеваемые по умолчанию, скорее всего, вам подойдут. Позже, познакомившись с Hadoop поближе, вы сможете настроить этот файл под свои требования (местоположение каталога журналов, путь к классам Java и т. д.).

Большинство параметров Hadoop находятся в конфигурационных XML-файлах. До выхода версии 0.20 они назывались hadoop-default.xml и hadoop-site.xml. Из названий понятно, что hadoop-default.xml содержит параметры по умолчанию, действующие в случае, когда они явно не переопределены в файле hadoop-site.xml. На практике вы будете иметь дело только с файлом hadoop-site.xml. В версии 0.20 этот файл разбили на три: core-site.xml, hdfs-site.xml и mapred-site.xml. Это позволило лучше соотнести конфигурационные параметры с той подсистемой Hadoop, которой они управляют. Далее в этой главе мы, как правило, будем отмечать, в каком из трех конфигурационных файлов находится интересующий нас параметр. Если вы работаете с ранней версией Hadoop, имейте в виду, что все конфигурационные параметры хранятся в файле hadoop-site.xml.

В следующих подразделах мы подробнее опишем режимы работы Hadoop и приведем конфигурационные файлы для каждого из них.

2.3.1. Локальный (автономный) режим

Автономный режим работы Hadoop подразумевается по умолчанию. Сразу после распаковки дистрибутива Hadoop еще ничего не знает об имеющемся у вас оборудовании. Поэтому на всякий случай предполагается, что конфигурация минимальна. В этом режиме все три XML-файла (или hadoop-site.xml в версиях младше 0.20) пусты:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
```

```
</configuration>
```

Если конфигурационные файлы пусты, то Hadoop будет работать целиком на локальной машине. Поскольку в этом случае во взаимодействии с другими узлами нет нужды, то в автономном режиме не используется HDFS и не запускается ни один из демонов Hadoop. В основном этот режим применяется для разработки и отладки логики MapReduce-программы без дополнительных сложностей, связанных с взаимодействием демонов. В примере из главы 1 MapReduce-программа запускалась именно в автономном режиме.

2.3.2. Псевдораспределенный режим

В псевдораспределенном режиме Hadoop работает в «вырожденном кластере», когда все демоны запущены на одной машине. Это дополнение к автономному режиму, позволяющее отлаживать код, наблюдать за потреблением памяти, изучать проблемы ввода/вывода в файловой системе HDFS и другие взаимодействия демонов. В листинге 2.1 показаны простые XML-файлы для настройки одиночного сервера в этом режиме.

Листинг 2.1.
Примеры трех конфигурационных файлов для псевдораспределенного режима

core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- В этот файл помещаются свойства, специфичные для конкретной
инсталляции, отменяющие умолчания. -->

<configuration>

<property>
 <name>fs.default.name</name>
 <value>hdfs://localhost:9000</value>
 <description>Имя файловой системы по умолчанию. Задается в виде URI,
 схема и адрес которого определяют реализацию файловой системы.
 </description>
</property>

</configuration>
```

mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
```

```
<!-- В этот файл помещаются свойства, специфичные для конкретной
инсталляции, отменяющие умолчания. -->

<configuration>

<property>
 <name>mapred.job.tracker</name>
 <value>localhost:9001</value>
 <description>Имя хоста и порт, определяющие узел, на котором
 работает демон MapReduce JobTracker.</description>
</property>

</configuration>
```

hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- В этот файл помещаются свойства, специфичные для конкретной
инсталляции, отменяющие умолчания. -->

<configuration>

<property>
 <name>dfs.replication</name>
 <value>1</value>
 <description>При создании файла можно задать фактический
 коэффициент репликации.</description>
 </property>

</configuration>
```

В файлах `core-site.xml` и `mapred-site.xml` задаются соответственно имена хостов и порты демонов NameNode и JobTracker. В файле `hdfs-site.xml` задается подразумеваемый по умолчанию коэффициент репликации для HDFS, который должен быть равен единице, потому что у нас есть всего один узел. Необходимо также задать местоположение узла Secondary NameNode в файле `masters` и подчиненных узлов в файле `slaves`:

```
[hadoop-user@master]$ cat masters
localhost
[hadoop-user@master]$ cat slaves
localhost
```

Хотя все демоны работают на одной машине, они все же общаются между собой по тому же протоколу SSH, что и в распределенном кластере. В разделе 2.2 подробно обсуждается настройка SSH-кана-

лов, однако для работы на одном узле достаточно проверить, позволяет ли машина открыть SSH-соединение с собой же:

```
[hadoop-user@master]$ ssh localhost
```

Если да, то все хорошо. В противном случае для настройки следует выполнить такие две команды:

```
[hadoop-user@master]$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
[hadoop-user@master]$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Теперь почти все готово к запуску Hadoop. Но сначала необходимо разметить файловую систему HDFS командой

```
[hadoop-user@master]$ bin/hadoop namenode -format
```

Далее можно запустить демоны скриптом `start-all.sh`. Затем убедитесь, что настройка прошла успешно, выполнив команду `jps`, которая выводит список всех работающих демонов.

```
[hadoop-user@master]$ bin/start-all.sh
[hadoop-user@master]$ jps
26893 Jps
26832 TaskTracker
26620 SecondaryNameNode
26333 NameNode
26484 DataNode
26703 JobTracker
```

Закончив работать с Hadoop, вы можете остановить его демоны командой

```
[hadoop-user@master]$ bin/stop-all.sh
```

Автономный и псевдораспределенный режимы предназначены для разработки и отладки. Реальный кластер Hadoop работает в третьем режиме – полностью распределенном.

2.3.3. Полностью распределенный режим

Мы неоднократно подчеркивали достоинства распределенной системы хранения и распределенных вычислений. Пора бы уже настроить настоящий кластер. Ниже мы будем использовать следующие имена серверов:

- *master* – главный узел кластера, на котором работают демоны NameNode и JobTracker;
- *backup* – сервер, на котором работает демон Secondary NameNode;

- *hadoop1, hadoop2, hadoop3, ...* – подчиненные узлы, на которых работают демоны DataNode и TaskTracker.

В листинге 2.2 приведены модифицированные версии файлов для псевдораспределенного режима (см. листинг 2.1), следующие вышеописанному соглашению об именах. Можете использовать их как заготовки для настройки своего кластера.

Листинг 2.2.
Примеры конфигурационных файлов для полностью распределенного режима

core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- В этот файл помещаются свойства, специфичные для конкретной
инсталляции, отменяющие умолчания. -->

<configuration>

<property>
 <name>fs.default.name</name>
 <value>hdfs://master:9000</value>
 <description>Имя файловой системы по умолчанию. Задается в виде URI,
 схема и адрес которого определяют реализацию файловой системы.
 </description>
</property>

</configuration>
```

1 Демон NameNode для файловой системы

mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- В этот файл помещаются свойства, специфичные для конкретной
инсталляции, отменяющие умолчания. -->

<configuration>

<property>
 <name>mapred.job.tracker</name>
 <value>master:9001</value>
 <description>Имя хоста и порт, определяющие узел, на котором
 работает демон MapReduce JobTracker.</description>
</property>

</configuration>
```

2 Главный узел JobTracker

hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- В этот файл помещаются свойства, специфичные для конкретной
инсталляции, отменяющие умолчания. -->

<configuration>

<property>
 <name>dfs.replication</name>
 <value>3</value>
 <description>При создании файла можно задать фактический
 коэффициент репликации.</description>
</property>

</configuration>
```

❸ Увеличен коэффициент репликации HDFS

Основные различия таковы:

- мы явно задали имена хостов, на которых работают демоны NameNode ❶ и JobTracker ❷;
- мы увеличили коэффициент репликации HDFS, чтобы воспользоваться преимуществами распределенной системы хранения ❸. Напомним, что данные реплицируются для повышения доступности и надежности.

Необходимо также изменить файлы `masters` и `slaves`, отразив в них местоположение других демонов.

```
[hadoop-user@master]$ cat masters
backup
[hadoop-user@master]$ cat slaves
hadoop1
hadoop2
hadoop3
...
```

Скопировав эти файлы на все узлы кластеры, не забудьте разметить HDFS, то есть подготовить ее к хранению данных:

```
[hadoop-user@master]$ bin/hadoop namenode-format
```

Теперь можно запустить демоны Hadoop:

```
[hadoop-user@master]$ bin/start-all.sh
```

и убедиться, что узлы выполняют назначенные им роли.

```
[hadoop-user@master]$ jps
30879 JobTracker
30717 NameNode
```

```
30965 Jps
[hadoop-user@backup]$ jps
2099 Jps
1679 SecondaryNameNode
[hadoop-user@hadoop1]$ jps
7101 TaskTracker
7617 Jps
6988 DataNode
```

Вот теперь у нас есть функционирующий кластер!

Переключение режимов

При работе с Hadoop мне кажется удобным использовать для переключения режимов символические ссылки вместо того, чтобы каждый раз редактировать XML-файлы. Для этого создайте свой каталог для каждого режима и поместите в него соответствующие этому режиму XML-файлы. Ниже приведен пример списка файлов в одном каталоге:

[hadoop@hadoop_master hadoop]$ ls -l

total 4884

drwxr-xr-x 2 hadoop-user hadoop 4096 Nov 26 17:36 bin

-rw-rw-r- - 1 hadoop-user hadoop 57430 Nov 13 19:09 build.xml

drwxr-xr-x 4 hadoop-user hadoop 4096 Nov 13 19:14 c++

-rw-rw-r- - 1 hadoop-user hadoop 287046 Nov 13 19:09 CHANGES.txt

lrwxrwxrwx 1 hadoop-user hadoop 12 Jan 5 16:06 conf -> conf.cluster

drwxr-xr-x 2 hadoop-user hadoop 4096 Jan 8 17:05 conf.cluster

drwxr-xr-x 2 hadoop-user hadoop 4096 Jan 2 15:07 conf.pseudo

drwxr-xr-x 2 hadoop-user hadoop 4096 Dec 1 10:10 conf.standalone

drwxr-xr-x 12 hadoop-user hadoop 4096 Nov 13 19:09 contrib

drwxrwxr-x 5 hadoop-user hadoop 4096 Jan 2 09:28 datastore

drwxr-xr-x 6 hadoop-user hadoop 4096 Nov 26 17:36 docs

...

Переключаться между конфигурациями можно командой Linux `ln` (например, `ln -s conf.cluster conf`). Этот способ полезен также, когда нужно временно изъять какой-то узел из кластера, чтобы отладить MapReduce-программу в псевдораспределенном режиме. Однако позаботьтесь о том, чтобы в разных режимах местоположения файлов для HDFS были различны, и перед изменением конфигурации остановите все демоны.

Итак, мы рассмотрели все настройки, необходимые для успешного запуска кластера Hadoop, и можем перейти к знакомству с веб-интерфейсом для мониторинга основных характеристик состояния кластера.

2.4. Веб-интерфейс для мониторинга кластера

Обсудив режимы работы Hadoop, мы можем перейти к рассмотрению веб-интерфейсов, предоставляемых для мониторинга состояния кластера. Работающий в браузере интерфейс позволяет получить нужную информацию гораздо быстрее, чем с помощью перелопачивания различных журналов и каталогов.

Демон NameNode возвращает общий отчет при обращении к порту 50070. Из этого отчета можно составить представление о состоянии файловой системы HDFS кластера. На рис. 2.4 показано, как выглядит отчет для кластера из двух узлов. Интерфейс позволяет произвести обзор файловой системы, узнать о состоянии каждого узла `DataNode` и просмотреть журналы демонов, чтобы удостовериться в правильности их функционирования.

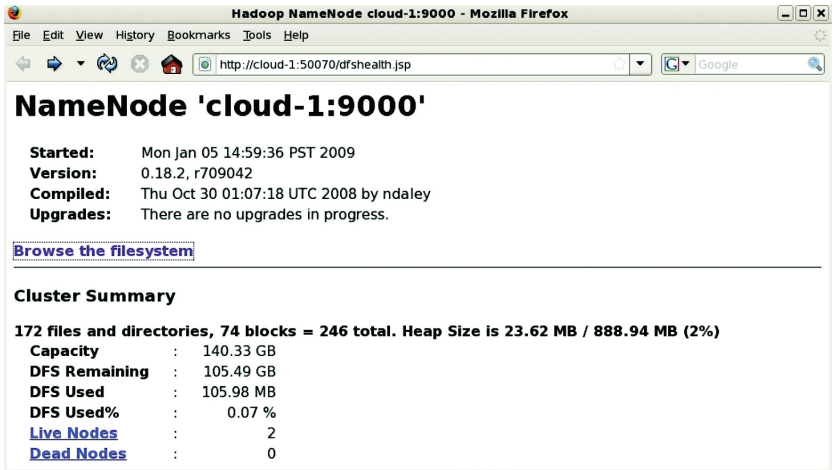


Рис. 2.4. Веб-интерфейс к системе HDFS. Отсюда можно произвести обзор файловой системы HDFS, узнать, сколько места для хранения есть в каждом узле, и следить за общим состоянием кластера.

Hadoop также представляет аналогичный сводный отчет о выполняющихся задачах MapReduce. На рис. 2.5 показан этот отчет, возвращаемый при обращении к порту 50030 узла JobTracker.

В этом интерфейсе также присутствует много информации. Можно узнать о состоянии каждой выполняющейся задачи MapReduce и

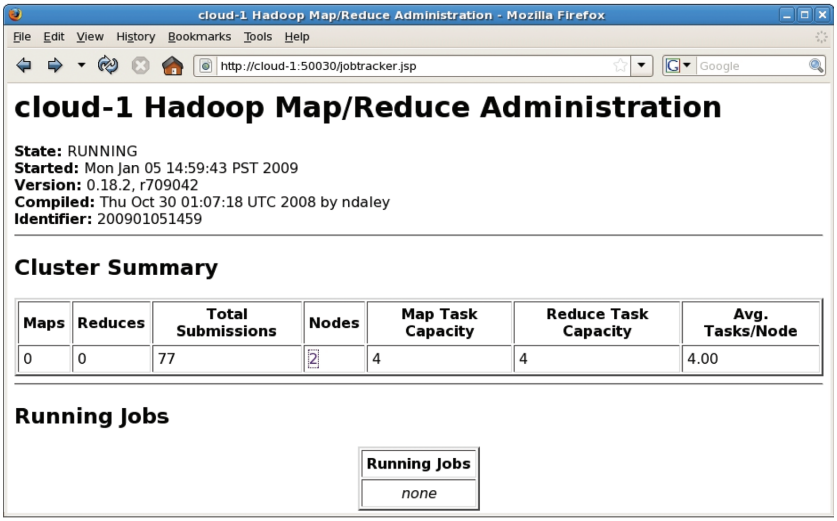


Рис. 2.5. Веб-интерфейс к информации о MapReduce-программах. Позволяет вести мониторинг активных задач MapReduce и просматривать журналы каждого задания map и reduce. Доступны также журналы уже завершившихся задач, что очень полезно для отладки программ.

просмотреть детальные сведения о завершившихся задачах. Последнее особенно важно – эти журналы показывают, какие узлы были заняты исполнением заданий и сколько времени и ресурсов было затрачено на каждое задание. Наконец, доступна также конфигурация для каждой задачи (рис. 2.6). Располагая всей этой информацией, вы можете настроить свои MapReduce-программы так, чтобы ресурсы кластера использовались наиболее эффективно.

На данном этапе полезность этих инструментов, возможно, еще не очевидна, но, когда вы начнете решать в кластере более сложные задачи, то они очень пригодятся. Вы еще будете иметь возможность оценить их важность по мере более тесного знакомства с Hadoop.

2.5. Резюме

В этой главе мы рассмотрели основные узлы кластера и их роли в архитектуре Hadoop. Вы научились конфигурировать кластер, а также познакомились с некоторыми инструментами для мониторинга его состояния.

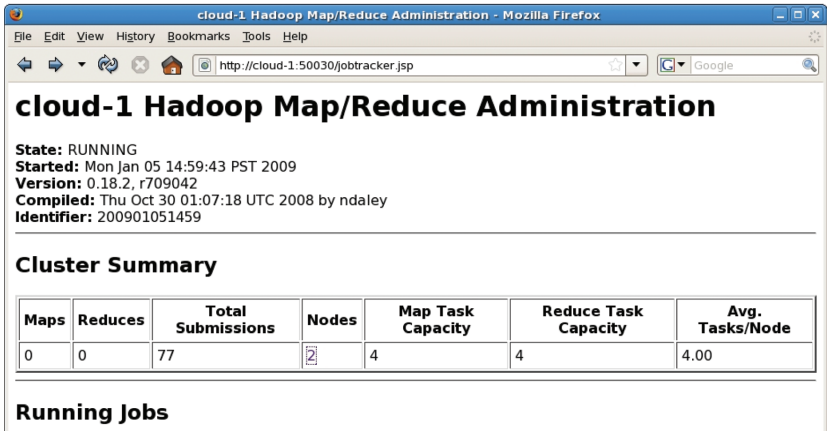


Рис. 2.6. Детальная конфигурация одной задачи MapReduce. Эта информация может оказаться полезной при настройке параметров для оптимизации производительности программ.

Вообще, в этой главе мы обсуждали в основном одноразовые операции. Один раз разметив узел NameNode для кластера, вы уже (хотелось бы надеяться) не будете возвращаться к этому занятию. Аналогично нет нужды многократно изменять конфигурационный файл hadoop-site.xml или распределять демонов по узлам. В следующей главе вы узнаете об аспектах Hadoop, с которыми приходится сталкиваться ежедневно, например, об управлении файлами в системе HDFS. После этого вы сможете приступить к написанию собственных MapReduce-программ и по-настоящему раскрыть для себя потенциал, заложенный в Hadoop.

ГЛАВА 3.
Компоненты Hadoop

В этой главе:

- Управление файлами в HDFS.
- Анализ компонентов каркаса MapReduce.
- Считывание входных и запись выходных данных.

В предыдущей главе мы рассматривали установку и настройку Hadoop. Мы говорили о назначении различных узлов и о том, как сконфигурировать их для совместной работы. А теперь, когда Hadoop запущен и работает, взглянем на него с точки зрения программиста. Если предыдущую главу сравнить с руководством по соединению проигрывателя, микшера, усилителя и динамиков между собой, то эта будет посвящена технике микширования музыки.

Сначала мы рассмотрим файловую систему HDFS, в которой хранятся данные, обрабатываемые приложениями для Hadoop. Затем займемся деталями каркаса MapReduce. В главе 1 мы уже видели MapReduce-программу, но ограничились концептуальным обсуждением ее логики. Теперь мы познакомимся с некоторыми классами и методами Java, а также стоящими за ними шагами обработки. Мы также научимся считывать и записывать данные в разных форматах.

3.1. Работа с файлами в системе HDFS

HDFS – это файловая система, предназначенная для обработки больших объемов распределенных данных с помощью таких технологий, как MapReduce. В HDFS можно сохранить гигантский набор данных (скажем, 100 ТБ) в виде одного файла; большинству других файловых

систем это не под силу. В главе 2 мы говорили о репликации данных с целью повышения доступности и распределения их по нескольким машинам для параллельной обработки. HDFS абстрагирует эти детали и создает иллюзию работы с одним файлом.

Поскольку HDFS не является файловой системой в смысле Unix, стандартные инструменты типа команд `ls` и `cp` в ней не работают[1], как и стандартные функции работы с файлами, например `fopen()` и `fread()`. С другой стороны, в состав Hadoop входит набор командных утилит, аналогичных командам Linux для работы с файлами. Ниже мы обсудим эти команды, составляющие основной интерфейс с системой HDFS. В разделе 3.1.2 рассматриваются написанные на Java библиотеки Hadoop для работы с файлами HDFS из программы.

Примечание. Обычно при работе с Hadoop где-то создаются файлы (например, журналы), которые затем копируются в HDFS с помощью одной из команд, рассматриваемых в следующем разделе. Далее эти данные обрабатываются MapReduce-программами. Но обычно они читают HDFS-файлы не напрямую, а обращаются к службам каркаса MapReduce, которые разбивают файлы на отдельные записи (пары ключ/значение) – элементарные блоки, с которыми работают MapReduce-программы. Вам редко придется программно считывать или записывать HDFS-файлы, разве что для нестандартного импорта или экспорта данных.

3.1.1. Основные команды для работы с файлами

Любая команда Hadoop для работы с файлами имеет вид

```
hadoop fs -cmd <args>
```

где `cmd` – имя конкретной команды, а `<args>` – переменное число аргументов.

Команда `cmd` обычно имеет такое же имя, как ее аналог в Unix. Например, команда вывода списка файлов записывается так[2]:

```
hadoop fs –ls
```

---

[1] Существует несколько проектов, авторы которых стремятся сделать HDFS монтируемой файловой системой Unix. Подробности см. по адресу http://wiki.apache.org/hadoop/MountableHDFS. На момент работы над этой книгой эти проекты еще не стали официальной частью Hadoop и, возможно, их надежность не соответствует требованиям, предъявляемым к промышленной системе.

[2] В некоторых старых документах команды для работы с файлами записываются в виде `hadoop dfs -cmd <args>`. `dfs` и `fs` обозначают одно и то же, но в настоящее время предпочтение отдается `fs`.

Рассмотрим наиболее употребительные операции управления файлами в каркасе Hadoop, а именно:

- добавление файлов и каталогов;
- извлечение файлов;
- удаление файлов.

URI для задания точного местоположения файла и каталога

Команды Hadoop могут работать как с файловой системой HDFS, так и с локальной файловой системой. (А в главе 9 мы увидим, что и со службой Amazon S3, как с файловой системой.) Местоположение файла или каталога задается с помощью URI-адреса. Полный формат URI имеет вид scheme://authority/path. Часть *scheme* – аналог протокола. Она может принимать значения `hdfs` (файловая система HDFS) или `file` (локальная файловая система). В случае HDFS *authority* – это адрес узла NameNode, а *path* – путь к интересующему файлу или каталогу. Например, в стандартной псевдораспределенной конфигурации, когда HDFS работает на локальной машине, прослушивая порт 9000, URI для доступа к файлу example.txt в каталоге user/chuck имеет вид hdfs://localhost:9000/user/chuck/example.txt. Для вывода содержимого этого файла можно воспользоваться командой Hadoop `cat`:

hadoop fs -cat hdfs://localhost:9000/user/chuck/example.txt

Ниже мы увидим, что в большинстве случаев задавать часть *scheme://authority* нет необходимости. При работе с локальной файловой системой предпочтительнее использовать не команды Hadoop, а стандартные команды Unix. Для копирования файлов между локальной файловой системой и HDFS команды Hadoop типа `put` и `get` считают локальную файловую систему соответственно источником или местом назначения, не требуя задания схемы *file://*. В других командах, если опустить часть URI scheme://authority, то будет использовано умолчание из конфигурации Hadoop. Например, в файле conf/core-site.xml для псевдораспределенной конфигурации свойство fs.default.name должно быть задано так:

```
<property>
   <name>fs.default.name</name>
   <value>hdfs://localhost:9000</value>
</property>
```

При такой конфигурации URI hdfs://localhost:9000/user/chuck/example.txt можно сокращать до /user/chuck/example.txt. Кроме того, HDFS по умолчанию принимает в качестве рабочего каталога /user/$USER, где $USER – имя текущего пользователя. Если вы вошли под именем chuck, то URI hdfs://localhost:9000/user/chuck/example.txt можно сократить до example.txt. Команда Hadoop `cat` для вывода содержимого этого файла будет иметь вид:

```
hadoop fs -cat example.txt
```

Добавление файлов и каталогов

Перед тем как применять Hadoop-программы к данным, хранящимся в системе HDFS, эти данные необходимо туда поместить. Предположим, что файловая система HDFS уже размечена и запущена (экспериментировать рекомендуем в псевдораспределенном режиме). Давайте создадим каталог и поместим в него файл.

По умолчанию рабочим каталогом HDFS является /user/$USER, где $USER – имя текущего пользователя. Но этот каталог не создается автоматически, поэтому создайте его командой `mkdir`. Для определенности я назову его *chuck*. А вы должны подставить в следующих примерах свое имя пользователя.

```
hadoop fs –mkdir /user/chuck
```

Команда Hadoop `mkdir` автоматически создает родительские каталоги, если они еще не существуют, – аналогично команде Unix `mkdir`, запущенной с флагом `-p`. Поэтому показанная выше команда создаст заодно и каталог /user. Убедимся в этом с помощью команды `ls`:

```
hadoop fs -ls /
```

В ответ будет выведен список, содержащий подкаталог /user внутри корневого каталога /:

```
Found 1 items
drwxr-xr-x - chuck supergroup 0 2009-01-14 10:23 /user
```

Чтобы увидеть все подкаталоги – по аналогии с командой Unix `ls`, запущенной с флагом `-r`, – выполните команду Hadoop `lsr`:

```
hadoop fs -lsr /
```

Будут рекурсивно выведены все файлы и каталоги:

```
drwxr-xr-x  - chuck supergroup      0 2009-01-14 10:23 /user
drwxr-xr-x  - chuck supergroup      0 2009-01-14 10:23 /user/chuck
```

Имея рабочий каталог, мы можем поместить в него файл. Создайте в локальной файловой системе какой-нибудь текстовый файл и назовите его example.txt. Для копирования файла из локальной файловой системы в HDFS применяется команда Hadoop `put`:

```
hadoop fs -put example.txt .
```

Обратите внимание на точку (.), переданную в качестве последнего аргумента. Она означает, что файл копируется в текущий рабочий каталог. Эта команда эквивалентна такой:

```
hadoop fs -put example.txt /user/chuck
```

Снова воспользуемся командой рекурсивного вывода списка файлов и убедимся, что новый файл действительно добавлен в систему HDFS:

```
$ hadoop fs -lsr /
drwxr-xr-x   - chuck supergroup       0 2009-01-14 10:23 /user
drwxr-xr-x   - chuck supergroup       0 2009-01-14 11:02 /user/chuck
-rw-r--r--   1 chuck supergroup     264 2009-01-14 11:02
➥/user/chuck/example.txt
```

На практике необязательно выводить все файлы рекурсивно, можно ограничиться своим рабочим каталогом. Для этого следует выполнить команду Hadoop `ls` в простейшей форме:

```
$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 chuck supergroup     264 2009-01-14 11:02
➥/user/chuck/example.txt
```

В списке показаны различные свойства: разрешения, владелец, группа, размер файла и дата последнего изменения – все как в стандартной команде Unix. В столбце, где сейчас находится «1», отображается коэффициент репликации файла. В псевдораспределенном режиме этот столбец всегда должен содержать 1. В производственном же кластере коэффициент репликации обычно равен 3, но может быть произвольным положительным числом. Для каталогов коэффициент репликации не имеет смысла, поэтому вместо него отображается дефис (-).

Поместив данные в систему HDFS, мы можем обработать их Hadoop-программой. Результатом обработки будет новый набор файлов в HDFS, и, возможно, вы захотите прочитать или извлечь результаты.

Извлечение файлов

Команда Hadoop `get` противоположна `put`. Она копирует файлы из системы HDFS в локальную файловую систему. Предположим, что локальный файл example.txt стерся, и мы хотели бы извлечь его из HDFS; для этого запустим команду

```
hadoop fs -get example.txt .
```

которая скопирует запрошенный файл в текущий рабочий каталог.

Получить доступ к данным можно и по-другому – отобразить их. Для этого предназначена команда Hadoop `cat`:

```
hadoop fs -cat example.txt
```

Команды Hadoop для работы с файлами можно использовать совместно с конвейерами Unix, чтобы направить выход одной команды на вход другой. Например, если файл очень велик (как

обычно и бывает в Hadoop), а вам нужно быстро узнать, что в нем содержится, то можно направить выход команды Hadoop `cat` на вход команды Unix `head`.

```
hadoop fs -cat example.txt | head
```

В Hadoop имеется собственная команда `tail` для просмотра последнего килобайта файла:

```
hadoop fs -tail example.txt
```

По завершении работы с файлами в HDFS их можно удалить, освободив тем самым место на диске.

Удаление файлов

Вряд ли вас удивит, что команда Hadoop для удаления файлов называется `rm`:

```
hadoop fs –rm example.txt
```

Команду `rm` можно использовать также для удаления пустых каталогов.

Получение справки

Полный список команд Hadoop для работы с файлами, вместе с порядком вызова и описанием, приведен в приложении. Как правило, команды построены по образцу аналогов в Unix. Для получения списка всех команд, поддерживаемых в вашей версии Hadoop, выполните команду `hadoop fs` (без параметров). Имеется также параметр `help` для получения справки по конкретной команде. Например, чтобы получить справку по команде `ls`, введите

```
hadoop fs –help ls
```

Будет выдано такое описание[3]:

```
-ls <path>:    вывести список файлов и каталогов, имена которых
               соответствуют заданному образцу. Если путь path
               не задан, выводится содержимое каталога
               /user/<currentUser>. Имена каталогов печатаются в виде
                     dirName (полный путь) <dir>
               а имена файлов в виде
                     fileName(полный путь) <r n> size
               где n – число реплик для данного файла, а size - размер
               файла в байтах.
```

Хотя набор командных утилит достаточен для большинства задач взаимодействия с файловой системой HDFS, он все же не является

[3] Для удобства читателя описание переведено на русский язык. *Прим. перев.*

исчерпывающим; бывают ситуации, когда необходим доступ с помощью программного интерфейса HDFS API. В следующем разделе мы покажем, как это делается.

3.1.2. Чтение и запись в HDFS из программы

Для обоснования нашего интереса к HDFS Java API разработаем программу PutMerge, которая будет объединять файлы при копировании их в HDFS. Командные утилиты такую операцию не поддерживают, нам придется прибегнуть к помощи API.

Эта программа может оказаться полезной для анализа журналов Apache, поступающих от нескольких веб-серверов. Можно, конечно, скопировать каждый журнал в HDFS, но, вообще говоря, Hadoop работает эффективнее с одним большим файлом, чем с множеством мелких. (Понятие «мелкий» здесь относительно, речь может идти о десятках и сотнях гигабайтов.) Кроме того, с точки зрения анализа нам хотелось бы рассматривать данные в журналах как единый большой файл. Тот факт, что он распределен по нескольким файлам, – побочный результат физической архитектуры фермы веб-серверов. Одно из возможных решений состоит в том, чтобы сначала объединить все файлы, а затем скопировать получившийся результат в систему HDFS. К сожалению, для объединения файлов на локальной машине должно быть очень много места на дисках. Гораздо проще было бы объединять файлы «на лету» – по мере копирования в HDFS.

Поэтому нам необходима операция типа PutMerge. В Hadoop имеется командная утилита `getmerge`, позволяющая объединять несколько HDFS-файлов перед копированием в локальную файловую систему. А нам нужна прямо противоположная команда, которой в Hadoop нет. Поэтому мы напишем ее с помощью HDFS API.

Основные классы для работы с файлами в Hadoop находятся в пакете org.apache.hadoop.fs. В состав примитивных операций входят хорошо знакомые функции open, read, write и close. На самом деле API для работы с файлами в Hadoop весьма общий и применим к файловым системам, отличным от HDFS. Для программы PutMerge мы воспользуемся этим API, чтобы читать файлы из локальной файловой системы и записывать в HDFS.

Точка входа в файловый API Hadoop – класс FileSystem. Это абстрактный класс, описывающий интерфейс с файловой системой, а для работы с HDFS и локальной файловой системой имеются его конк-

ретные подклассы. Для получения нужного экземпляра `FileSystem` следует вызвать фабричный метод `FileSystem.get(Configuration conf)`. `Configuration` – это специальный класс для хранения конфигурационных параметров в виде пар ключ/значение. Его подразумеваемая по умолчанию реализация основана на конфигурации ресурсов для системы HDFS. Получить объект `FileSystem` для работы с HDFS можно следующим образом:

```
Configuration conf = new Configuration();
FileSystem hdfs = FileSystem.get(conf);
```

Для получения объекта `FileSystem`, ориентированного на локальную файловую систему, предназначен фабричный метод `FileSystem.getLocal(Configuration conf)`:

```
FileSystem local = FileSystem.getLocal(conf);
```

В API Hadoop для работы с файлами объекты `Path` служат для представления имен файлов и каталогов, а объекты `FileStatus` – для хранения метаданных о файлах и каталогах. Наша программа PutMerge будет объединять все файлы из некоторого локального каталога. Чтобы получить список файлов в каталоге, воспользуемся методом `listStatus()` класса `FileSystem`:

```
Path inputDir = new Path(args[0]);
FileStatus[] inputFiles = local.listStatus(inputDir);
```

Размер массива `inputFiles` равен числу файлов в указанном каталоге. В каждом объекте `FileStatus` из массива `inputFiles` хранятся метаданные: размер файла, права доступа, время модификации и т. д. Программе PutMerge интересен объект `Path`, соответствующий каждому файлу; его можно получить методом `inputFiles[i].getPath()`. У этого объекта можно запросить объект `FSDataInputStream` для чтения файла:

```
FSDataInputStream in = local.open(inputFiles[i].getPath());
byte buffer[] = new byte[256];
int bytesRead = 0;
while( (bytesRead = in.read(buffer)) > 0) {
   ...
}
in.close();
```

`FSDataInputStream` – это подкласс стандартного класса Java `java.io.DataInputStream` с дополнительной поддержкой для произвольной выборки. Для записи в HDFS-файл имеется аналогичный объект `FSDataOutputStream`.

```
Path hdfsFile = new Path(args[1]);
FSDataOutputStream out = hdfs.create(hdfsFile);
out.write(buffer, 0, bytesRead);
out.close();
```

Для завершения программы PutMerge напишем цикл обхода всех файлов в массиве `inputFiles`, в котором будем читать файл из локальной системы и записывать в HDFS. Полный текст программы приведен в листинге 3.1.

Листинг 3.1.
Программа PutMerge

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class PutMerge {
   public static void main(String[] args) throws IOException {

      Configuration conf = new Configuration();
      FileSystem hdfs = FileSystem.get(conf);
      FileSystem local = FileSystem.getLocal(conf);

      Path inputDir = new Path(args[0]);
      Path hdfsFile = new Path(args[1]);

      try {
         FileStatus[] inputFiles = local.listStatus(inputDir);
         FSDataOutputStream out = hdfs.create(hdfsFile);

         for (int i=0; i<inputFiles.length; i++) {
            System.out.println(inputFiles[i].getPath().getName());
            FSDataInputStream in =
          ➥local.open(inputFiles[i].getPath());
            byte buffer[] = new byte[256];
            int bytesRead = 0;
            while( (bytesRead = in.read(buffer)) > 0) {
               out.write(buffer, 0, bytesRead);
            }
            in.close();
         }
         out.close();
      } catch (IOException e) {
```

1. Задаем входной каталог и выходной файл
2. Получаем список локальных файлов
3. Создаем поток вывода в HDFS
4. Открываем локальный поток ввода

```
            e.printStackTrace();
        }
    }
}
```

Сначала мы задаем локальный каталог и результирующий HDFS-файл, исходя из указанных пользователем аргументов ❶. В ❷ мы получаем информацию о каждом файле в локальном входном каталоге. Затем в ❸ создаем поток вывода для записи в HDFS-файл. Далее мы перебираем все файлы в локальном каталоге и в ❹ открываем поток ввода для чтения файла. Остальное – стандартный код для копирования файла.

В классе `FileSystem` имеются также методы `delete()`, `exists()`, `mkdirs()` и `rename()` для выполнения других стандартных операций с файлами. Актуальную документацию об API Hadoop для работы с файлами в формате Javadoc можно найти по адресу http://hadoop.apache.org/core/docs/current/api/org/ apache/hadoop/fs/package-summary.html.

Мы показали, как работать с файлами в системе HDFS. Теперь вы знаете несколько способов поместить данные в HDFS и извлечь их оттуда. Но какой интерес в том, чтобы просто иметь данные? Вам ведь хочется обрабатывать их, анализировать и подвергать иным манипуляциям. Поэтому закончим на этом рассмотрение HDFS и перейдем к следующему важному компоненту Hadoop – каркасу MapReduce. Наша задача – научиться программировать для него.

3.2. Анатомия MapReduce-программы

Выше мы отмечали, что MapReduce-программа обрабатывает данные посредством манипулирования парами ключ/значение. В общем виде это выглядит так:

map: (K1,V1) → list(K2,V2)
reduce: (K2,list(V2)) → list(K3,V3)

Конечно, такое представление потока данных слишком общее. В этом разделе мы подробно рассмотрим каждый этап типичной MapReduce-программы. На рис. 3.1 представлена высокоуровневая диаграмма всего процесса. По мере обсуждения мы детализируем структуру каждого компонента.

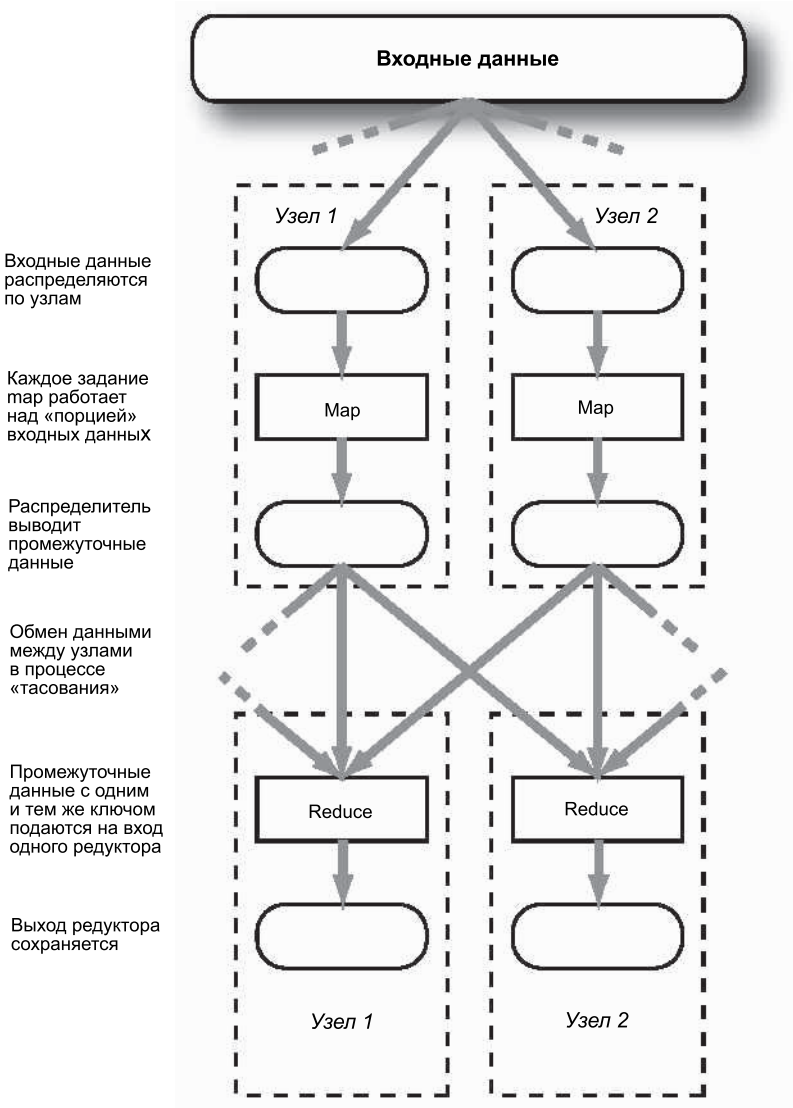


Рис. 3.1. Общий поток данных в MapReduce-программе. Обратите внимание, что после того, как распределение входных данных по различным узлам состоялось, в следующий раз узлы взаимодействуют между собой только на этапе «тасования» . Такое ограничение на взаимодействие весьма благотворно сказывается на масштабируемости.

Перед тем как начинать анализ обработки данных на каждом этапе, следует познакомиться с типами данных, которые поддерживает Hadoop.

3.2.1. Типы данных в Hadoop

Мы уже не раз говорили о ключах и значениях, но еще ничего не сказали об их типах. Каркас MapReduce не позволяет задавать в качестве типа произвольный класс. Например, хотя мы часто говорим, что ключи и значения являются целыми числами, строками и т. д., это не означает, что речь идет о стандартных классах Java: Integer, String и других. Дело в том, что в каркасе MapReduce определен некий способ сериализации пар ключ/значение для передачи их через границы машин в кластере, и только классы, поддерживающие эту сериализацию, могут выступать в роли типов ключей и значений.

Точнее говоря, классы, реализующие интерфейс `Writable`, могут быть типами значений, а классы, реализующие интерфейс `WritableComparable<T>`, – типами как ключей, так и значений. Отметим, что интерфейс `WritableComparable<T>` представляет собой комбинацию интерфейсов `Writable` и `java.lang.Comparable<T>`. Требование сравнимости ключей выдвигается потому, что на этапе редукции они сортируются, тогда как значения просто передаются напрямую.

В состав Hadoop входит ряд готовых классов, реализующих интерфейс `WritableComparable`, в том числе классы-обертки для всех простых типов данных (см. табл. 3.1).

Таблица 3.1. Наиболее употребительные типы ключей и значений. Все эти классы реализуют интерфейс WritableComparable

Класс	Описание
`BooleanWritable`	Обертка стандартного типа Boolean
`ByteWritable`	Обертка одного байта
`DoubleWritable`	Обертка типа Double
`FloatWritable`	Обертка типа Float
`IntWritable`	Обертка типа Integer
`LongWritable`	Обертка типа Long
`Text`	Обертка для хранения текста в кодировке UTF8
`NullWritable`	Заглушка для случая, когда ключ или значение не нужны

Ключи и значения могут принадлежать не только типам, которые Hadoop поддерживает по умолчанию. Можно самостоятельно создать тип, реализующий интерфейс `Writable` (или `WritableComparable<T>`). Например, в листинге 3.2 приведен класс для представления ребер графа. Его можно использовать для описания маршрута полета между двумя городами.

Листинг 3.2.
Пример класса, реализующего интерфейс WritableComparable

```
public class Edge implements WritableComparable<Edge>{

   private String departureNode;
   private String arrivalNode;

   public String getDepartureNode() { return departureNode;}

   @Override
   public void readFields(DataInput in) throws IOException {          ❶
      departureNode = in.readUTF();                                   Определяем,
      arrivalNode = in.readUTF();                                     как читать
   }                                                                  данные

   @Override
   public void write(DataOutput out) throws IOException {             ❷
      out.writeUTF(departureNode);                                    Определяем,
      out.writeUTF(arrivalNode);                                      как выводить
   }                                                                  данные

   @Override
   public int compareTo(Edge o) {
      return (departureNode.compareTo(o.departureNode) != 0)          ❸
         ? departureNode.compareTo(o.departureNode)                   Определяем
         : arrivalNode.compareTo(o.arrivalNode);                      упорядочение
   }                                                                  данных
}
```

В классе `Edge` реализованы методы `readFields()` ❶ и `write()` ❷ интерфейса `Writable`. Они работают с классами Java `DataInput` и `DataOutput` для сериализации содержимого объекта. Реализован также метод `compareTo()` ❸ интерфейса `Comparable`. Он возвращает –1, 0 или 1, если объект `Edge`, от имени которого метод вызван, соответственно меньше, равен или больше объекта `Edge`, переданного в качестве аргумента.

Определив необходимые интерфейсы, мы можем теперь приступить к рассмотрению первого этапа обработки данных: распределителя (см. рис. 3.1).

3.2.2. Распределитель

Класс, претендующий на роль распределителя, должен реализовывать интерфейс `Mapper` и наследовать классу `MapReduceBase`. Неудивительно, что класс `MapReduceBase` является базовым как для распределителей, так и для редукторов. Он содержит два метода, выступающие в качестве конструктора и деструктора.

- *void configure* (*JobConf job*) – в этом методе можно извлечь параметры, заданные либо в конфигурационных XML-файлах, либо в главном классе приложения. Этот метод следует вызвать перед началом обработки данных.
- *void close* () – этот метод, вызываемый непосредственно перед завершением задания `map`, выполняет очистку: закрывает соединения с базой данных, открытые файлы и т. п.

Интерфейс `Mapper` отвечает за шаг обработки данных. В нем используется обобщенный тип Java вида `Mapper<K1,V1,K2,V2>`, где классы ключей и значений реализуют соответственно интерфейсы `WritableComparable` и `Writable`. Его единственный метод предназначен для обработки одной пары ключ/значение:

```
void map(K1 key,
         V1 value,
         OutputCollector<K2,V2> output,
         Reporter reporter
        ) throws IOException
```

Этот метод порождает (возможно, пустой) список пар (K2, V2) для заданной входной пары (K1, V1). Объект `OutputCollector` получает результат работы распределителя, а объект `Reporter` предоставляет средства для получения дополнительной информации о ходе работы распределителя.

В состав Hadoop входят несколько полезных реализаций распределителей. Некоторые из них представлены в табл. 3.2.

Таблица 3.2. Некоторые полезные готовые реализации интерфейса Mapper

Класс	Описание
`IdentityMapper<K,V>`	Реализует интерфейс Mapper<K,V,K,V>, отображая вход на выход без какого-либо преобразования
`InverseMapper<K,V>`	Реализует интерфейс Mapper<K,V,V,K>, меняя ключи и значения местами

Таблица 3.2. (окончание)

Класс	Описание
`RegexMapper<K>`	Реализует интерфейс Mapper<K,Text,Text,LongWritable>, порождая пару (match, 1) для каждой входной пары, удовлетворяющей регулярному выражению
`TokenCountMapper<K>`	Реализует интерфейс Mapper<K,Text,Text,LongWritable>, порождая пары (token, 1), соответствующие лексемам, выделенным из входного значения

Как следует из самого названия MapReduce, вслед за фазой распределения map идет фаза редукции reduce, показанная в нижней части рис. 3.1.

3.2.3. Редуктор

Как и в случае распределителя, реализация редуктора должна расширять базовый класс MapReduce, что позволяет выполнить конфигурирование и очистку. Кроме того, она должна реализовывать интерфейс `Reducer`, в котором есть всего один метод:

```
void reduce(K2 key,
            Iterator<V2> values,
            OutputCollector<K3,V3> output,
            Reporter reporter
          ) throws IOException
```

Получив на входе результаты работы различных распределителей, редуктор сортирует входные данные по ключам пар ключ/значение и группирует значения с одинаковыми ключами. Затем вызывается метод `reduce()`, который порождает (возможно, пустой) список пар (K3, V3) путем обхода значений, ассоциированных с данным ключом. Объект `OutputCollector` получает результат работы редуктора и записывает его в выходной файл. Объект предоставляет средства для получения дополнительной информации о ходе работы редуктора.

В табл. 3.3 приведены две реализации простых редукторов, входящие в состав Hadoop.

Таблица 3.3. Некоторые полезные готовые реализации интерфейса Reducer

Класс	Описание
`IdentityMapper<K,V>`	Реализует интерфейс Reducer<K,V,K,V>, отображая вход на выход без какого-либо преобразования

Таблица 3.3. (окончание)

Класс	Описание
`LongSumReducer<K>`	Реализует интерфейс Reducer<K,LongWritable,K, LongWritable>, подсчитывая сумму значений с данным ключом

Хотя мы называем программы для Hadoop MapReduce-приложениями, существует еще один важный промежуточный шаг: передача результатов работы распределителей различным редукторам. Ответственность за него возлагается на разбиватель.

3.2.4. Разбивка – направление выхода распределителя

Программисты, только начинающие осваивать MapReduce, часто допускают одну и ту же ошибку – используют только один редуктор. Ну действительно, ведь единственный редуктор сортирует все данные перед обработкой, а кому не нравятся отсортированные данные? Предыдущие наши рассуждения о каркасе MapReduce должны были убедить вас в нелепости такого подхода. В нем не задействованы достоинства параллельных вычислений. С одним редуктором наше вычислительное облако превращается в дождевую каплю.

Но при наличии нескольких редукторов необходимо как-то определить, какому из них посылать пару ключ/значение, выработанную распределителем. По умолчанию редуктор определяется путем хеширования ключа. Hadoop поддерживает такую стратегию, предлагая класс `HashPartitioner`. Но иногда `HashPartitioner` только сбивает с истинного пути. Вернемся к классу `Edge`, который был представлен в разделе 3.2.1.

Допустим, что класс `Edge` используется для анализа данных о полетах с целью узнать, сколько пассажиров вылетало из каждого аэропорта. Данные могут быть представлены в таком виде:

(San Francisco, Los Angeles) Chuck Lam
(San Francisco, Dallas) James Warren
...

При использовании класса `HashPartitioner` эти две строки могут быть посланы разным редукторам. Количество отлетов будет обработано дважды и оба раза неверно. Как настроить разбиватель для конкретного приложения? В данном случае нам нужно, чтобы все ребра с одним и тем же пунктом вылета направлялись

одному редуктору. Этого легко добиться путем хеширования члена `departureNode` класса `Edge`:

```
public class EdgePartitioner implements Partitioner<Edge, Writable>
{
   @Override
   public int getPartition(Edge key, Writable value, int numPartitions)
   {
      return key.getDepartureNode().hashCode() % numPartitions;
   }

   @Override
   public void configure(JobConf conf) { }
}
```

Разбиватель должен реализовать всего два метода: `configure()` и `getPartition()`. Первый конфигурирует разбиватель, пользуясь конфигурационными файлами Hadoop, а второй возвращает целое число в диапазоне от нуля до количества заданий редукции, которое определяет, какому редактору отправлять пару ключ/значение.

Чтобы вам было проще понять механизм работы разбивателя, на рис. 3.2 приведена иллюстрация.

Между этапами распределения и редукции, MapReduce-программа должна получить результаты работы распределителей и раздать их редукторам. Обычно этот процесс называют *тасованием*, поскольку результат работы распределителя на одном узле может быть послан редукторам, работающим на нескольких узлах.

3.2.5. Комбинатор – локальная редукция

Часто в MapReduce-программах возникает желание произвести «локальную редукцию» перед тем, как раздавать результаты работы распределителя. Вернемся снова к программе `WordCount` из главы 1. Если задача обрабатывает документ, содержащий слово «the» 574 раза, то было бы гораздо эффективнее один раз сохранить и перетасовать пару («the», 574), а не многократно раздавать пару («the», 1). Этот шаг обработки называется комбинированием. Подробнее мы будем рассматривать комбинаторы в разделе 4.6.

3.2.6. Подсчет слов с помощью готовых классов распределителя и редуктора

Мы завершили предварительное рассмотрение всех основных компонентов каркаса MapReduce. Теперь, познакомившись с некоторыми

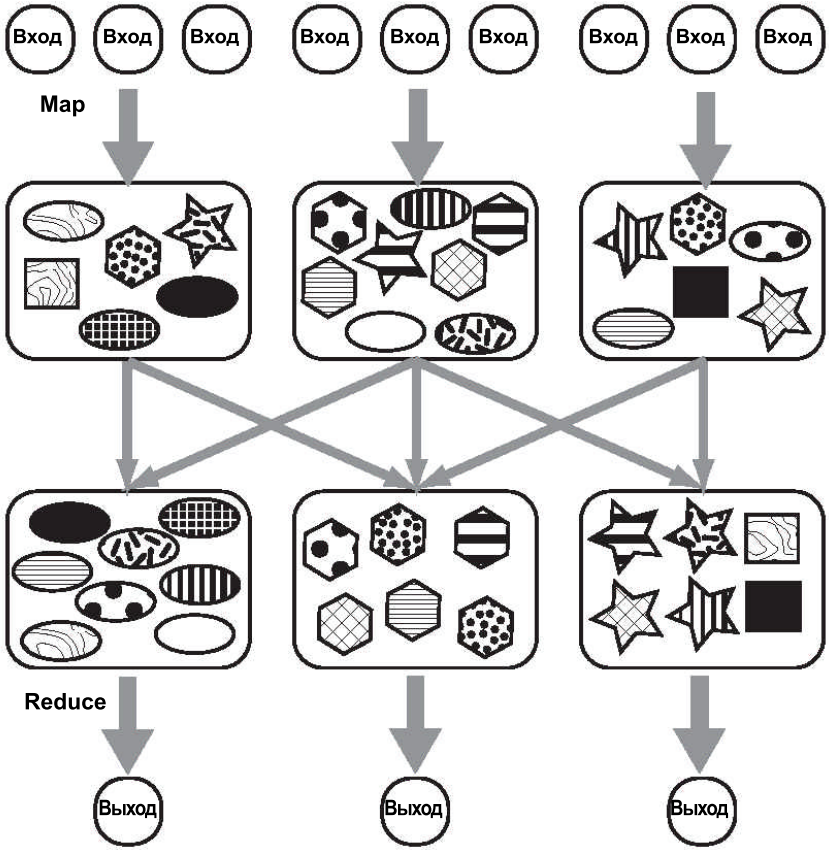


Рис. 3.2. Взгляд на поток данных в MapReduce-программе с точки зрения разбиения и тасования. Каждая фигурка представляет пару ключ/значение. Форма фигурок соответствуют ключам, а штриховка – значениям. В результате тасования все фигурки одной и той же формы (одинаковые ключи) попадают одному и тому же редуктору. Разные ключи также могут попасть одному редуктору (как видно на примере самого правого редуктора). Решение о том, кому какой ключ передать, принимает разбиватель. Отметим, что нагрузка на самый левый редуктор выше, поскольку данных с ключом в форме эллипса оказалось больше.

классами, входящими в состав Hadoop, будет полезно вернуться к программе `WordCount` и переписать ее на основе этих классов (см. листинг 3.3).

Листинг 3.3.
Новая версия программы WordCount

```
public class WordCount2 {
   public static void main(String[] args) {
      JobClient client = new JobClient();
      JobConf conf = new JobConf(WordCount2.class);

      FileInputFormat.addInputPath(conf, new Path(args[0]));
      FileOutputFormat.setOutputPath(conf, new Path(args[1]));

      conf.setOutputKeyClass(Text.class);
      conf.setOutputValueClass(LongWritable.class);
      conf.setMapperClass(TokenCountMapper.class);
      conf.setCombinerClass(LongSumReducer.class);
      conf.setReducerClass(LongSumReducer.class);

      client.setConf(conf);
      try {
         JobClient.runJob(conf);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Класс TokenCountMapper входит в состав Hadoop

Класс LongSumRducer входит в состав Hadoop

Мы должны написать только управляющую часть этой MapReduce-программы, поскольку воспользовались готовыми классами `TokenCountMapper` ❶ и `LongSumReducer` ❷, входящими в состав Hadoop. Просто, не правда ли? Hadoop предоставляет средства и для создания более сложных программ (это будет основной темой второй части книги), а сейчас мы хотим только подчеркнуть, что Hadoop позволяет быстро писать полезные программы, ограничиваясь минимальным объемом кода.

3.3. Чтение и запись

Посмотрим, как MapReduce считывает входные и записывает выходные данные, обращая особое внимание на форматы файлов. Для упрощения распределенной обработки MapReduce принимает некоторые допущения об обрабатываемых данных. Он также обеспечивает гибкость при работе с различными форматами данных.

Обычно входные данные находятся в больших файлах, размером десятки, сотни гигабайтов и даже больше. Один из фундаментальных принципов, лежащих в основе вычислительной мощи каркаса

MapReduce, – разбиение входных данных на *куски*, которые можно параллельно обработать на нескольких машинах. В терминологии Hadoop эти куски называются *входными порциями* (input splits). Для эффективного распараллеливания размер порции должен быть достаточно мал. (Если все входные данные поместить в одну порцию, то никакого распараллеливания вообще не будет.) С другой стороны, порция не должна быть слишком мала, иначе накладные расходы на запуск и останов обработки одной порции будут чрезмерно велики.

Идея разбиения входных данных (которые часто представляют собой один гигантский файл) на порции для параллельной обработки проясняет некоторые решения, принятые при проектировании обобщенного класса `FileSystem` и файловой системы HDFS в частности. Например, класс `FileSystem` предоставляет для чтения файлов класс `FSDataInputStream` взамен стандартного класса Java `java.io.DataInputStream`. Класс `FSDataInputStream` расширяет класс `DataInputStream`, добавляя возможность чтения с произвольной выборкой, необходимую MapReduce потому, что машине может быть поручено обработать порцию, находящуюся где-то в середине входного файла. Без умения читать с произвольного места пришлось бы читать файл с самого начала до нужной точки, что крайне неэффективно. Напомним также, что HDFS спроектирована для хранения данных, которые MapReduce разбивает на части и обрабатывает параллельно. HDFS хранит файлы в виде блоков, находящихся на разных машинах. Грубо говоря, каждый блок – это порция. Поскольку на разных машинах, скорее всего, окажутся разные блоки, то распараллеливание произойдет автоматически, если каждый блок (порция) будет обрабатываться той машиной, на которой он находится. И наконец, для повышения надежности HDFS реплицирует блоки на несколько узлов, поэтому MapReduce может выбирать любой узел из тех, на которых находится копия нужного блока.

Входные порции и границы записей

Отметим, что входные порции – это разбиение по границам логических записей, тогда как блоки HDFS – физическое разбиение входных данных. Если они совпадают, то эффективность резко возрастает, однако на практике такого идеального совмещения не бывает. Записи могут пересекать границы блоков. Hadoop гарантирует, что будут обработаны все записи. Машине, обрабатывающей некую порцию, может потребоваться фрагмент записи из блока, не совпадающего с «главным» и находящегося на другом узле. Впрочем, затратами на получение фрагмента записи можно пренебречь, так как это случается сравнительно редко.

Напомним, что MapReduce работает с парами ключ/значение. До сих пор мы видели, что Hadoop по умолчанию считает каждую строку входного файла записью, при этом ключом является смещение строки от начала файла в байтах, а значением – содержимое строки. Но данные необязательно должны быть представлены в виде таких записей. Hadoop поддерживает и другие форматы, а также позволяет определить собственный формат.

3.3.1. Интерфейс InputFormat

Способ разбиения входного файла на порции определяется реализацией интерфейса `InputFormat`. По умолчанию используется класс `TextInputFormat`, именно этот формат данных неявно подразумевался до сих пор. Часто бывает, что входные данные не имеют явно выраженного ключа, а читаются построчно. Класс `TextInputFormat` возвращает в качестве ключа смещение строки от начала файла в байтах, но нам еще не встречалась программа, в которой такой ключ использовался бы для обработки данных.

Популярные классы, реализующие InputFormat

В табл. 3.4 перечислены другие популярные реализации интерфейса `InputFormat` с описанием того, какую пару ключ/значение данный класс возвращает распределителю.

Таблица 3.4. Популярные реализации `InputFormat`. Класс `TextInputFormat` подразумевается по умолчанию, если явно не указан какой-то другой. Описаны также типы ключей и значений.

InputFormat	Описание
`TextInputFormat`	Каждая строка в текстовом файле считается записью. Ключом является смещение строки от начала файла в байтах, значением – содержимое строки. ключ: `LongWritable` значение: `Text`
`KeyValueTextInputFormat`	Каждая строка в текстовом файле считается записью. Первый встретившийся в строке символ-разделитель делит ее на две части. Все, что находится перед разделителем, является ключом, а все находящееся после него – значением. Что считать разделителем, определяется свойством key.value.separator.in.input.line property; по умолчанию подразумевается знак табуляции (\t). ключ: `Text` значение: `Text`

Таблица 3.4. (окончание).

InputFormat	Описание
`SequenceFileInputFormat<K,V>`	`InputFormat` для чтения файлов последовательностей. Ключи и значения определяются пользователем. Файл последовательности – это специфичный для Hadoop сжатый двоичный формат. Он оптимизирован для передачи данных с выхода одной задачи MapReduce на вход другой. ключ: `K` (определяется пользователем) значение: `V` (определяется пользователем)
`NLineInputFormat`	То же, что `TextInputFormat`, но каждая порция гарантированно состоит ровно из `N` строк. Значение `N` задается свойством mapred.line.input.format.linespermap, по умолчанию равным 1. ключ: `LongWritable` значение: `Text`

Формат `KeyValueTextInputFormat` применяется в структурированных входных файлах, где некий предопределенный символ, обычно знак табуляции (\t), разделяет ключи и значение в каждой строке (записи). Рассмотрим, к примеру, файл, содержащий временные метки и URL-адреса, разделенные знаком табуляции:

```
17:16:18 http://hadoop.apache.org/core/docs/r0.19.0/api/index.html
17:16:19 http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html
17:16:20 http://wiki.apache.org/hadoop/GettingStartedWithHadoop
17:16:20 http://www.maxim.com/hotties/2008/finalist_gallery.aspx
17:16:25 http://wiki.apache.org/hadoop/
```

Свой объект `JobConf` вы можете настроить так, чтобы для считывания этого файла использовался класс `KeyValueTextInputFormat`.

```
conf.setInputFormat(KeyValueTextInputFormat.class);
```

Тогда первая прочитанная распределителем запись будет иметь ключ «17:16:18» и значение «http://hadoop.apache.org/core/docs/r0.19.0/api/index.html», вторая запись – ключ «17:16:19» и значение «http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html». И так далее.

Напомним, что во всех ранее рассмотренных распределителях ключи имели тип `LongWritable`, а значения – тип `Text`. В случае формата `TextInputFormat` тип `LongWritable` представляется разумным, так как это числовое смещение от начала файла. Но при использовании формата `KeyValueTextInputFormat` и ключ, и значение будут

иметь тип `Text`, поэтому реализацию класса `Mapper` и метод `map()` придется соответственно изменить.

На вход задачи MapReduce необязательно подавать внешние данные. На самом деле, часто бывает, что входом одной задачи MapReduce является выход другой задачи. Как мы увидим, формат выходных данных также можно настраивать. По умолчанию выходной формат совпадает с форматом `KeyValueTextInputFormat` (то есть каждая строка является записью, в которой ключ и значение разделены знаком табуляции). Hadoop предлагает также гораздо более эффективный двоичный сжатый формат *файла последовательности* (sequence file). Он оптимизирован для эффективной обработки и является предпочтительным при создании цепочек задач MapReduce. Для чтения файлов последовательности служит класс `SequenceFileInputFormat`. Типы ключей и значений в таком файле определяются пользователем. Типы данных на входе и выходе должны совпадать, а реализация класса `Mapper` и метода `map()` должны принимать правильный тип.

Создание своего формата InputFormat – классы InputSplit и RecordReader

Иногда требуется читать входные данные, представленные в нестандартном формате, не поддерживаемом ни одной из готовых реализаций `InputFormat`. В этом случае нужно написать свой класс, реализующий интерфейс `InputFormat`. Посмотрим, во что это выливается. Интерфейс `InputFormat` содержит всего два метода.

```
public interface InputFormat<K, V> {

   InputSplit[] getSplits(JobConf job, int numSplits) throws IOException;

   RecordReader<K, V> getRecordReader(InputSplit split,
                                      JobConf job,
                                      Reporter reporter) throws
IOException;
}
```

Эти методы определяют функциональность входного формата:

- идентифицировать все файлы, используемые в качестве входных данных, и разбить их на порции. Каждому распределителю назначается одна порция;
- предоставить объект (`RecordReader`), который будет перебирать записи в данной порции и выделять из них ключи и значения определенных типов.

Кто должен позаботиться о том, как файлы разбиваются на порции? При создании собственной реализации `InputFormat` вы должны унаследовать классу `FileInputFormat`, который отвечает за разбиение файла. На самом деле, все реализации `InputFormat`, перечисленные в табл. 3.4, являются подклассами `FileInputFormat`. В этом классе реализован метод `getSplits()`, но метод `getRecordReader()` оставлен абстрактным и должен быть предоставлен подклассами. Реализация `getSplits()` в классе `FileInputFormat` пытается разбить входные данные на порции, количество которых примерно равно значению параметра `numSplits`, учитывая при этом ограничение, что число байтов в каждой порции должно быть не меньше `mapred.min.split.size`, но меньше, чем размер блока файловой системы. На практике размер порции обычно оказывается равен размеру блока, который в HDFS по умолчанию составляет 64 МБ.

В классе `FileInputFormat` имеется ряд защищенных методов, которые подкласс может переопределить. Один из них, `isSplitable(FileSystem fs, Path filename)`, проверяет, можно ли вообще разбить данный файл. Реализация по умолчанию возвращает `true`, так что все файлы, размер которых больше одного блока, будут разбиваться. Но иногда желательно, чтобы файл состоял всего из одной порции, и тогда следует переопределить метод `isSplitable()`, так чтобы он возвращал `false`. Например, некоторые алгоритмы сжатия не поддерживают разбиение файлов (невозможно начать чтение с середины сжатого файла). А некоторые операции обработки, например конвертирование файлов, рассматривают каждый файл как неделимую запись, поэтому разбивать файл на порции нельзя.

Наследуя классу `FileInputFormat`, вы должны сосредоточиться на написании своего класса, реализующего интерфейс `RecordReader`, который отвечает за разбиение на записи и последующее преобразование каждой записи в пару ключ/значение. Рассмотрим сигнатуру этого интерфейса:

```
public interface RecordReader<K, V> {
   boolean next(K key, V value) throws IOException;

   K createKey();
   V createValue();

   long getPos() throws IOException;
   public void close() throws IOException;
   float getProgress() throws IOException;
}
```

Вместо того чтобы реализовывать интерфейс `RecordReader` с нуля, мы снова можем прибегнуть к помощи классов, предоставляемых Hadoop. Например, класс `LineRecordReader` реализует интерфейс `RecordReader<LongWritable, Text>`. Он используется в классе `TextInputFormat`, так как считывает по одной строке, возвращая смещение от начала файла в качестве ключа и содержимое строки в качестве значения. Класс `KeyValueLineRecordReader` используется форматом `KeyValueTextInputFormat`. Как правило, ваша реализация `RecordReader` будет являться оберткой вокруг одной из существующих реализаций, а основная работа будет производиться в методе `next()`.

В качестве примера создания собственного класса `InputFormat` мы рассмотрим задачу чтения записей специального типа, а не обобщенного типа `Text`. Ранее мы применяли класс `KeyValueTextInputFormat` для чтения файла, в котором временные метки и URL-адреса были разделены знаками табуляции. При этом и временная метка, и URL рассматривались как данные типа `Text`. А теперь напишем класс `TimeUrlTextInputFormat`, который будет работать практически так же, но с одним отличием – URL будет представлен типом `URLWritable`[4]. Как было предложено выше, создадим свой класс, расширив `FileInputFormat` и реализовав фабричный метод, который будет возвращать экземпляр нашей реализации `RecordReader`.

```
public class TimeUrlTextInputFormat extends
      ➥FileInputFormat<Text, URLWritable> {

   public RecordReader<Text, URLWritable> getRecordReader(
         ➥InputSplit input, JobConf job, Reporter reporter)
         ➥throws IOException {
      return new TimeUrlLineRecordReader(job, (FileSplit)input);
   }
}
```

Наш класс `URLWritable` не представляет собой ничего сложного:

```
public class URLWritable implements Writable {

   protected URL url;

   public URLWritable() { }

   public URLWritable(URL url) {
```

[4] Ключ (время) также можно было бы представить типом, отличным от `Text`, например `CalendarWritableComparable`. Оставляем это в качестве упражнения для читателя, а сами ограничимся более простой постановкой задачи.

```
      this.url = url;
   }

   public void write(DataOutput out) throws IOException {
      out.writeUTF(url.toString());
   }

   public void readFields(DataInput in) throws IOException {
      url = new URL(in.readUTF());
   }

   public void set(String s) throws MalformedURLException {
      url = new URL(s);
   }
}
```

Класс `TimeUrlLineRecordReader` должен реализовать все шесть методов интерфейса `RecordReader` плюс конструктор. По существу, это обертка вокруг класса `KeyValueTextInputFormat`, только значение записи преобразуется из типа `Text` в тип `URLWritable`.

```
class TimeUrlLineRecordReader implements RecordReader<Text, URLWritable> {

   private KeyValueLineRecordReader lineReader;
   private Text lineKey, lineValue;

   public TimeUrlLineRecordReader(JobConf job, FileSplit split) throws
    ➥IOException {
      lineReader = new KeyValueLineRecordReader(job, split);
      lineKey = lineReader.createKey();
      lineValue = lineReader.createValue();
   }

   public boolean next(Text key, URLWritable value) throws IOException {
      if (!lineReader.next(lineKey, lineValue)) {
         return false;
      }
      key.set(lineKey);
      value.set(lineValue.toString());
      return true;
   }

   public Text createKey() {
      return new Text("");
   }

   public URLWritable createValue() {
      return new URLWritable();
   }
```

```
    public long getPos() throws IOException {
        return lineReader.getPos();
    }

    public float getProgress() throws IOException {
        return lineReader.getProgress();
    }

    public void close() throws IOException {
        lineReader.close();
    }
}
```

Наш класс `TimeUrlLineRecordReader` создает объект типа `KeyValueLineRecordReader` и делегирует ему вызовы методов `getPos()`, `getProgress()` и `close()`. Метод `next ()` преобразует объект `lineValue` из типа `Text` в тип `URLWritable`.

3.3.2. Интерфейс OutputFormat

Каркас MapReduce выводит данные в виде файлов, пользуясь классом `OutputFormat`, аналогичным классу `InputFormat`. При выводе никаких порций нет, поскольку каждый редуктор записывает результаты в свой собственный файл. Выходные файлы размещаются в общем каталоге и обычно именуются по схеме part-*nnnnn*, где *nnnnn* – идентификатор редуктора. Объекты `RecordWriter` форматируют результаты, тогда как объекты `RecordReader` разбирают входные данные.

Hadoop предлагает несколько стандартных реализаций интерфейса `OutputFormat`, они приведены в табл. 3.5. Неудивительно, что почти все они наследуют абстрактному классу `FileOutputFormat` – точно так же, как реализации `InputFormat` наследуют `FileInputFormat`.

Чтобы указать, какой объект `OutputFormat` использовать, нужно вызвать метод `setOutputFormat()` объекта `JobConf`, в котором хранится конфигурация задачи MapReduce.

Примечание. Возможно, вам непонятно, зачем различать интерфейс `OutputFormat` (`InputFormat`) и класс `FileOutputFormat` (`FileInputFormat`), коль скоро все классы, реализующие `OutputFormat` (`InputFormat`) вроде бы расширяют `FileOutputFormat` (`FileInputFormat`). Существуют ли реализации `OutputFormat` (`InputFormat`), которые не работают с файлами? Да, есть класс `NullOutputFormat`, реализующий `OutputFormat` тривиальным образом и потому не нуждающийся в наследовании `FileOutputFormat`. Но гораздо важнее, что имеются реализации `OutputFormat` (`InputFormat`), которые работают с базами данных, а не с файлами, и эти классы выделены в отдельную ветвь иерархии наследования, независимую от `FileOutput-`

Format (FileInputFormat). У них есть специализированные применения, интересующийся читатель может поискать в онлайновой документации описание классов DBInputFormat и DBOutputFormat.

Таблица 3.5. Основные классы, реализующие OutputFormat. По умолчанию подразумевается TextOutputFormat

OutputFormat	Описание
TextInputFormat<K,V>	Выводит каждую запись в виде строки текста. Ключи и значения записываются в виде строк и разделяются знаком табуляции (\t). Разделитель можно изменить с помощью свойства mapred.textoutputformat.separator.
SequenceFileOutputFormat<K,V>	Выводит пары ключ/значение в формате файла последовательности, разработанном специально для Hadoop. Работает в сочетании с классом SequenceFileInputFormat.
NullOutputFormat<K,V>	Не выводит ничего.

По умолчанию применяется класс TextOutputFormat, который выводит каждую запись в виде строки текста. Ключи и значения преобразуются в строки методом toString() и между ними вставляется знак табуляции (\t). Изменить символ-разделитель позволяет свойство mapred.textoutputformat.separator.

Класс TextOutputFormat выводит данные в формате, который можно прочитать с помощью класса KeyValueTextInputFormat. Можно также вывести данные в формате, совместимом с TextInputFormat, если в качестве типа ключа взять NullWritable. В таком случае ключ пары ключ/значение вообще не будет выводиться, и разделителя тоже не будет. Чтобы подавить всякий вывод, воспользуйтесь форматом NullOutputFormat. Такое подавление может быть полезно, если редуктор выводит результаты по-своему и не нуждается в том, чтобы Hadoop записывал какие-нибудь дополнительные файлы.

Наконец, класс SequenceFileOutputFormat выводит файл последовательности, который можно прочитать с помощью класса SequenceFileInputFormat. Это полезно для вывода промежуточных данных при связывании задач MapReduce в цепочку.

3.4. Резюме

Hadoop – это программный каркас, предлагающий новый взгляд на обработку данных. В нем есть собственная файловая система, HDFS, разработанная для хранения данных в виде, оптимизированном для обработки больших объемов. Для работы с HDFS необходим специальный набор инструментов. Они входят в состав Hadoop и, к счастью, очень похожи на стандартные утилиты Unix и Java.

Часть Hadoop, относящаяся к обработке данных, известна под названием MapReduce. Хотя основная работа MapReduce-программы сводится к операциям Map и Reduce (что и неудивительно), каркас производит и другие операции, в частности, разбиение данных на порции и тасование, важность которых не следует недооценивать. Имеется возможность настроить под свои потребности такие операции, как разбиение и комбинирование. Hadoop предоставляет средства для чтения данных и вывода их в разных форматах.

Теперь, когда вы лучше понимаете принципы функционирования Hadoop, можно перейти ко второй части книги, в которой рассматриваются различные подходы к написанию практически полезных программ с помощью Hadoop.

Часть 2

Hadoop в действии

В части 2 мы приобретем практические навыки, необходимые для создания и запуска программ обработки данных в Hadoop. Мы изучим различные примеры использования Hadoop для анализа набора данных о патентах, в том числе некоторые достаточно сложные алгоритмы, например фильтр Блума. Мы также рассмотрим приемы программирования и администрирования, полезные при работе с Hadoop в производственной среде.

ГЛАВА 4.
Создание простых MapReduce-программ

В этой главе:

- Данные о патентах как пример набора данных для обработки с помощью Hadoop.
- Шаблон MapReduce-программы.
- Простые MapReduce-программы для вычисления статистических показателей.
- Потоковый API Hadoop для написания MapReduce-программ на скриптовых языках.
- Комбинатор, повышающий производительность.

Модель программирования в каркасе MapReduce отличается от большинства знакомых вам моделей. Для ее освоения нужны время и практика. Для наработки опыта мы в следующих двух главах рассмотрим целый ряд примеров, иллюстрирующих различные приемы программирования с помощью MapReduce. По-разному применяя MapReduce, вы постепенно разовьете интуицию и приобретете привычку думать «в духе MapReduce». Примеры различаются по степени сложности. В одном из наиболее развитых приложений мы познакомимся с фильтром Блума, структурой данных, которую обычно не изучают в стандартных курсах информатики. Вы увидите, что при обработке больших наборов данных, неважно с помощью Hadoop или как-то иначе, часто приходится переосмысливать привычные алгоритмы.

Мы предполагаем, что вы уже поняли, что представляет собой Hadoop. Вы научились настраивать Hadoop, а также откомпилирова-

ли и запустили пример программы, например подсчета слов. Теперь поработаем с реальными наборами данных.

4.1. Получение набора данных о патентах

Для того чтобы сделать в Hadoop что-то осмысленное, необходимы данные. Во многих примерах ниже будут использоваться наборы данных о патентах, которые можно загрузить с сайта Национального бюро экономических исследований (National Bureau of Economic – NBER) по адресу http://www.nber.org/patents/. Первоначально эти наборы были представлены в работе «The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools»[1]. Мы возьмем данные о цитировании `cite75_99.txt` и описания патентов `apat63_99.txt`.

Примечание. Размер каждого из этих наборов данных составляет примерно 250 МБ. Это не слишком много, поэтому примеры можно будет запускать в автономном или псевдораспределенном режиме Hadoop. Так что есть шанс попрактиковаться в написании MapReduce-программ, не имея доступа к реальному кластеру. Что в Hadoop хорошо, так это уверенность, что написанная MapReduce-программа будет работать в кластере из нескольких машин, обрабатывающем наборы данных, в 100 или 1000 раз большие, практически без внесения каких-либо изменений в код. Широко распространена практика создания сравнительно небольшого подмножества набора производственных данных, называемого *набором данных для разработки*. Размер такого набора может составлять порядка нескольких сотен мегабайтов. Он подается на вход программы, работающей в автономном или псевдораспределенном режиме. В результате вы получаете короткий цикл разработки, все удобства от запуска программы на своей собственной машине и изолированную среду для отладки.

Эти два набора мы выбрали в качестве примера, потому что они являются типичными представителями данных, с которыми вам, вероятно, придется столкнуться в реальной жизни. Прежде всего, данные о цитировании описываются графом, примерно таким же, какой можно встретить при работе с гиперссылками в веб или с социальными сетями. Далее, патенты публикуются в хронологическом порядке, поэтому некоторые их свойства напоминают временные ряды. С каждым патентом связано физическое лицо (изобретатель) и местоположение (страна проживания изобретателя), то есть нечто, что можно рассматривать как персональные и географические данные соответственно. Наконец, на эти данные можно взглянуть как на отношения в

[1] NBER Working Paper 8498, by Hall, B. H., A. B. Jaffe, and M. Tratjenberg (2001).

базе данных с четко определенной схемой, представленные в формате CSV (значения через запятую)[2].

4.1.1. Данные о цитировании патентов

Набор данных о цитировании патентов содержит библиографические ссылки на патенты, выданные в США в период с 1975 по 1999 год. В нем свыше 16 миллионов строк, и первые несколько строк выглядят так:

```
"CITING","CITED"
3858241,956203
3858241,1324234
3858241,3398406
3858241,3557384
3858241,3634889
3858242,1515701
3858242,3319261
3858242,3668705
3858242,3707004
...
```

Данные представлены в стандартном формате CSV (значения через запятую), причем в первой строке находится описание столбцов. Каждая из последующих строк представляет одну библиографическую ссылку. Например, вторая строка говорит, что патент 3858241 ссылается на патент 956203. Файл отсортирован по номеру ссылающегося патента. Как видите, патент 3858241 ссылается на пять разных патентов. Пристальный количественный анализ данных может выявить интересные закономерности.

Если просто читать файл данных, то данные о цитировании выглядят как нагромождение цифр. Однако можно предложить и более интересные способы «рассуждения» о данных. Один из способов – визуализировать их в виде графа. На рис. 4.1 изображена часть графа цитирования. Видно, что некоторые патенты цитируются часто, тогда как другие не цитируются вовсе[3]. Патенты 5936972 и 6009552 ссыла-

[2] Общеупотребительные типы данных не исчерпываются тем, что присутствует в этих двух наборах. Так, важным типом данных является текст; в этих примерах он не встречается, но мы уже имели с ним дело в программе подсчета слов. Кроме того, следует упомянуть о таких типах, как XML, изображения и географические координаты (заданные широтой и долготой). Математические матрицы в примерах также не представлены в общем виде, хотя граф цитирования можно интерпретировать как разреженную матрицу из нулей и единиц.

[3] Как и при любом способе анализа данных, следует осторожно подходить к интерпретации ограниченного набора. Если некоторый патент не ссылается ни на какие другие, то это может означать, что патент старый, и информация о цитировании для него отсутствует. С другой стороны, на недавно выданные патенты ссылок меньше, потому что об их существовании могут знать только патенты, выданные позже.

ются на множества патентов, имеющие большое общее подмножество (4354269, 4486882, 5598422), хотя и не ссылаются друг на друга. Мы воспользуемся Hadoop для вывода дескриптивных статистических характеристик этого набора, а также обнаружим интересные закономерности, не очевидные с первого взгляда.

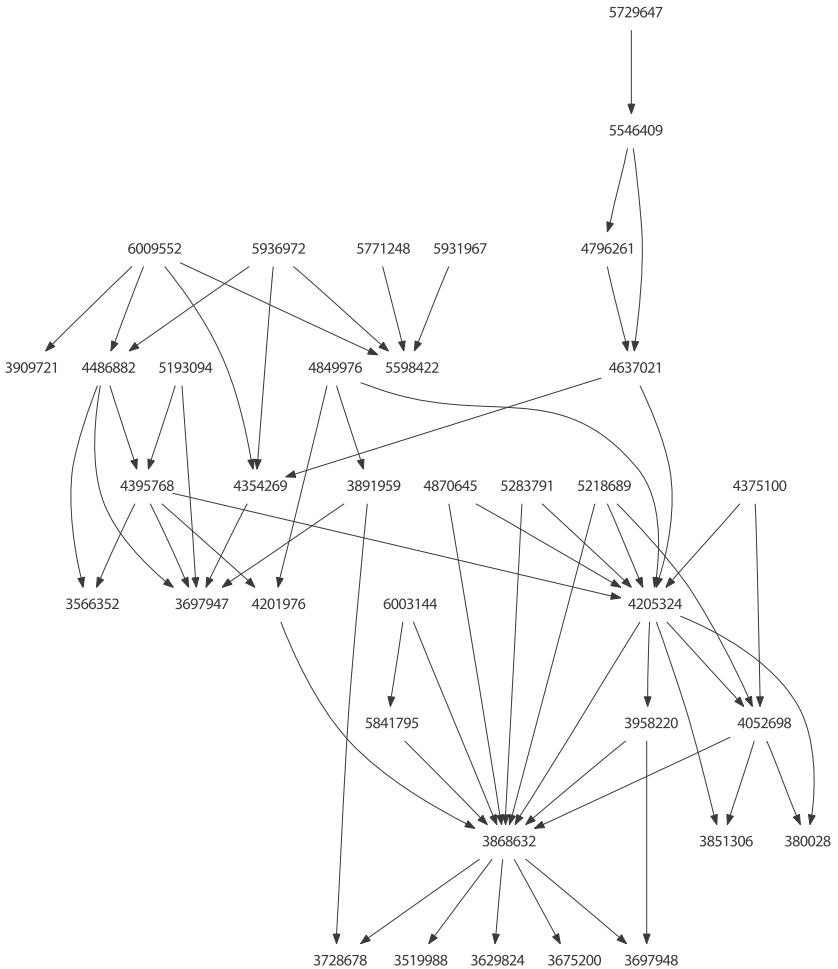


Рис. 4.1. Часть набора данных о цитировании патентов, представленного в виде графа. Каждому патенту соответствует вершина графа, а каждой ссылке – ориентированное ребро (стрелка).

4.1.2. Данные об описаниях патентов

В качестве второго набора данных мы возьмем описания патентов. Описание включает номер патента, год подачи заявки, год выдачи патента, количество пунктов в формуле изобретения и прочее. Взгляните на первые несколько строк этого набора. Он напоминает таблицу в реляционной базе данных, только представленную в формате CSV. В этом наборе более 2,9 миллионов строк. Как и в большинстве реальных наборов, многие данные отсутствуют.

```
"PATENT","GYEAR","GDATE","APPYEAR","COUNTRY","POSTATE","ASSIGNEE",
➥ "ASSCODE","CLAIMS","NCLASS","CAT","SUBCAT","CMADE","CRECEIVE",
➥ "RATIOCIT","GENERAL","ORIGINAL","FWDAPLAG","BCKGTLAG","SELFCTUB",
➥ "SELFCTLB","SECDUPBD","SECDLWBD"
3070801,1963,1096,,"BE","",,1,,269,6,69,,1,,0,,,,,,,
3070802,1963,1096,,"US","TX",,1,,2,6,63,,0,,,,,,,,
3070803,1963,1096,,"US","IL",,1,,2,6,63,,9,,0.3704,,,,,,
3070804,1963,1096,,"US","OH",,1,,2,6,63,,3,,0.6667,,,,,,,
3070805,1963,1096,,"US","CA",,1,,2,6,63,,1,,0,,,,,,,
3070806,1963,1096,,"US","PA",,1,,2,6,63,,0,,,,,,,,,
3070807,1963,1096,,"US","OH",,1,,623,3,39,,3,,0.4444,,,,,,,
3070808,1963,1096,,"US","IA",,1,,623,3,39,,4,,0.375,,,,,,,
3070809,1963,1096,,"US","AZ",,1,,4,6,65,,0,,,,,,,,,
```

Первая строка содержит имена 23 атрибутов, понятных только патентным поверенным. Но даже не вникая в детали, полезно понимать, что означают некоторые из них. В табл. 4.1 описаны первые десять атрибутов.

Таблица 4.1. Определение первых 10 атрибутов набора данных с описаниями патентов.

Имя атрибута	Назначение
PATENT	Номер патента
GYEAR	Год выдачи патента
GDATE	Дата выдачи патента, представленная как количество дней с 1 января 1960 года
APPYEAR	Год подачи заявки (присутствует только для патентов, выданных не ранее 1967 года)
COUNTRY	Страна проживания первого автора
POSTATE	Штат проживания первого автора (если проживает в США)
ASSIGNEE	Числовой идентификатор патентообладателя

Таблица 4.1. (окончание).

Имя атрибута	Назначение
ASSCODE	Однозначный (число от 1 до 9) тип патентообладателя (частное лицо, проживающее в США; правительство США; организация, зарегистрированная в США; частное лицо, проживающее вне США, и т. д.)
CLAIMS	Количество пунктов в формуле изобретения (присутствует только для патентов, выданных не ранее 1975 года)
NCLASS	Трехзначный цифровой класс основного патента

4.2. Определение шаблона MapReduce-программы

По большей части MapReduce-программы составляются по единому шаблону. Приступая к созданию новой программы, вы обычно берете какую-нибудь существующую и модифицируете ее, добиваясь желаемого. В этом разделе мы напишем первую MapReduce-программу и объясним, из каких частей она состоит. В дальнейшем она послужит нам шаблоном для составления других программ.

Наша первая программа будет *инвертировать* данные о цитировании патентов. Для каждого патента мы хотим найти и сгруппировать все патенты, ссылающиеся на него. Результат должен получиться примерно такой:

```
1       3964859,4647229
10000   4539112
100000  5031388
1000006 4714284
1000007 4766693
1000011 5033339
1000017 3908629
1000026 4043055
1000033 4190903,4975983
1000043 4091523
1000044 4082383,4055371
1000045 4290571
1000046 5918892,5525001
1000049 5996916
1000051 4541310
1000054 4946631
1000065 4748968
1000067 5312208,4944640,5071294
1000070 4928425,5009029
```

Мы уже видели, что патенты 5 312 208, 4 944 640 и 5 071 294 ссылаются на патент 1 000 067. В этом разделе мы не станем уделять много внимания потоку данных в каркасе MapReduce, об этом уже шла речь в главе 3. Основной интерес для нас будет представлять сама структура MapReduce-программы. Вся программа будет состоять из одного файла, приведенного в листинге 4.1.

Листинг 4.1.
Шаблон типичной программы для Hadoop

```
public class MyJob extends Configured implements Tool {

   public static class MapClass extends MapReduceBase
      implements Mapper<Text, Text, Text, Text> {

      public void map(Text key, Text value,
                      OutputCollector<Text, Text> output,
                      Reporter reporter) throws IOException {
            output.collect(value, key);
      }
   }

   public static class Reduce extends MapReduceBase
      implements Reducer<Text, Text, Text, Text> {

      public void reduce(Text key, Iterator<Text> values,
                         OutputCollector<Text, Text> output,
                         Reporter reporter) throws IOException {

         String csv = "";
         while (values.hasNext()) {
            if (csv.length() > 0) csv += ",";
               csv += values.next().toString();
         }
         output.collect(key, new Text(csv));
      }
   }

   public int run(String[] args) throws Exception {
      Configuration conf = getConf();

      JobConf job = new JobConf(conf, MyJob.class);

      Path in = new Path(args[0]);
      Path out = new Path(args[1]);
      FileInputFormat.setInputPaths(job, in);
      FileOutputFormat.setOutputPath(job, out);

      job.setJobName("MyJob");
      job.setMapperClass(MapClass.class);
```

```
        job.setReducerClass(Reduce.class);

        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.set("key.value.separator.in.input.line", ",");

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new MyJob(), args);

        System.exit(res);
    }
}
```

Мы будем придерживаться соглашения о том, что всякая задача MapReduce полностью определяется единственным классом, в данном случае `MyJob`. Hadoop требует, чтобы распределитель `Mapper` и редуктор `Reducer` были реализованы в виде отдельных статических классов. Эти классы невелики, и в нашем шаблоне мы сделали их внутренними классами, вложенными в `MyJob`. Преимущество такого решения в том, что все находится в одном файле, поэтому управлять кодом становится проще. Однако имейте в виду, что внутренние классы совершенно независимы и не имеют почти никаких специальных связей с классом `MyJob`. Различные узлы, на которых установлены разные виртуальные машины Java, клонируют и запускают классы `Mapper` и `Reducer` по ходу исполнения задачи, но все остальные части класса задачи исполняются *только* на клиентской машине.

Классы `Mapper` и `Reducer` мы рассмотрим чуть ниже. А без них скелет класса `MyJob` выглядит так:

```
public class MyJob extends Configured implements Tool {

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();

        JobConf job = new JobConf(conf, MyJob.class);
        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setJobName("MyJob");
```

```
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.set("key.value.separator.in.input.line", ",");

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
       int res = ToolRunner.run(new Configuration(), new MyJob(), args);
        System.exit(res);
    }
}
```

Основу программы составляет метод `run()`, который иногда называют *драйвером*. Драйвер создает и конфигурирует объект именованной задачи `JobConf`, а затем передает его методу `JobClient.runJob()`, который запускает задачу MapReduce. (Класс `JobClient`, в свою очередь, обращается к демону JobTracker для задачи в кластере.) В объекте `JobConf` хранятся все конфигурационные параметры, необходимые для исполнения задачи. Драйвер должен указать в `job` пути к входным и выходным файлам, а также классы `Mapper` и `Reducer` – основные параметры любой задачи. Помимо этого, любая задача может изменить свойства, установленные по умолчанию: `InputFormat`, `OutputFormat` и прочие. Для изменения любого конфигурационного параметра следует вызвать метод `set()` объекта `JobConf`. После того как объект `JobConf` передан методу `JobClient.runJob()`, он трактуется как генеральный план исполнения задачи.

У объекта `JobConf` есть много параметров, но мы не хотим, чтобы драйвер задавал все. Неплохой отправной точкой могут служить конфигурационные файлы Hadoop. При запуске задачи из командной строки пользователь также может передать дополнительные параметры, изменяющие конфигурацию задачи. Кроме того, драйвер может определить собственный набор команд и обработать переданные в командной строке аргументы, позволяя тем самым пользователю модифицировать некоторые конфигурационные параметры. Поскольку необходимость в этом возникает достаточно часто, Hadoop предоставляет вспомогательные классы `ToolRunner`, `Tool` и `Configured`. При совместном использовании в шаблоне `MyJob` эти классы позво-

ляют задаче распознать заданные пользователем аргументы, которые поддерживаются классом `GenericOptionsParser`. Например, выше вы запускали класс `MyJob` такой командой:

```
bin/hadoop jar playground/MyJob.jar MyJob input/cite75_99.txt output
```

А если бы мы хотели, чтобы задача видела только результаты работы распределителя (это бывает полезно для отладки), то могли бы указать, что число редукторов должно быть равно нулю, добавив флаг `-D mapred.reduce.tasks=0`.

```
bin/hadoop jar playground/MyJob.jar MyJob -D mapred.reduce.tasks=0
➥ input/cite75_99.txt output
```

Это работает, хотя нигде в программе обработка флага `-D` явно не присутствует. Благодаря классу `ToolRunner` `MyJob` автоматически поддерживает параметры, перечисленные в табл. 4.2.

Таблица 4.2. Параметры, поддерживаемые классом GenericOptionsParser

Параметр	Описание
`-conf <конфигурационный файл>`	Задать имя конфигурационного файла.
`-D <свойство=значение>`	Установить значение свойства.
`-fs <local|namenode:port>`	Задать узел NameNode, может быть «local».
`-jt <local|jobtracker:port>`	Задать узел JobTracker.
`-files <список файлов>`	Задать список файлов (через запятую), необходимых задаче MapReduce. Эти файлы автоматически пересылаются на все узлы, где исполняются задания, чтобы к ним можно было обращаться локально.
`-libjars <список jar-файлов>`	Задать список JAR-файлов (через запятую), которые нужно включить в путь к классам для всех виртуальных машин Java, исполняющих задания.
`-archives <список архивов>`	Задать список архивов (через запятую), которые следует распаковать на всех узлах, где исполняются задания.

В нашем шаблоне принято соглашение называть класс распределителя, реализующий интерфейс `Mapper`, `MapClass`, а класс редуктора, реализующий интерфейс `Reducer`, – `Reduce`. Было бы симметричнее

назвать класс распределителя `Map`, но в Java уже есть класс (точнее, интерфейс) с таким именем. Классы распределителя и редуктора расширяют `MapReduceBase` – небольшой класс, предоставляющий пустые реализации методов `configure()` и `close()`, объявленных в обоих интерфейсах. Эти методы используются для того, чтобы настроить и очистить задания map (или reduce). Переопределять их придется только в более сложных задачах.

Ниже представлены сигнатуры классов `MapClass` и `Reduce`:

```
public static class MapClass extends MapReduceBase
   implements Mapper<K1, V1, K2, V2> {

   public void map(K1 key, V1 value,
                   OutputCollector<K2, V2> output,
                   Reporter reporter) throws IOException { }
}

public static class Reduce extends MapReduceBase
   implements Reducer<K2, V2, K3, V3> {

   public void reduce(K2 key, Iterator<V2> values,
                      OutputCollector<K3, V3> output,
                      Reporter reporter) throws IOException { }
}
```

Центральным для класса `MapClass` является метод `map()`, а для класса `Reduce` – метод `reduce()`. Методу `map()` при вызове передается пара, состоящая из ключа типа `K1` и значения типа `V1`. Пары ключ/значение, порожденные распределителем, выводятся с помощью метода `collect()` объекта `OutputCollector`. Где-то в методе `map()` должно быть обращение

```
output.collect((K2) k, (V2) v);
```

Методу `reduce()` при вызове передается пара, состоящая из ключа типа `K2` и списка значений типа `V2`. Отметим, что типы `K2` и `V2` должны совпадать с одноименными типами, которые использовались в распределителе. Скорее всего, в методе `reduce()` будет цикл для обхода всех значений типа `V2`.

```
while (values.hasNext()) {
   V2? v = values.next();
   ...
}
```

Методу `reduce()` также передается объект `OutputCollector` для сбора результирующих пар ключ/значение типа `K3/V3`. Где-то в методе `reduce()`, наверное, будет обращение

```
output.collect((K3) k, (V3) v);
```

Помимо одинаковости типов `K2` и `V2` в классах `MapClass` и `Reduce`, необходимо еще, чтобы используемые в них типы ключей и значений были согласованы с форматом ввода, классом выходного ключа и классом выходного значения, заданными в драйвере. Использование `KeyValueTextInputFormat` подразумевает, что типы `K1` и `V1` должны совпадать с `Text`. Драйвер должен вызвать методы `setOutputKeyClass()` и `setOutputValueClass()`, указав типы `K2` и `V2` соответственно.

Наконец, типы всех ключей и значений должны быть подтипами `Writable`, что позволит Hadoop сериализовать данные при передаче между узлами распределенного кластера. На самом деле, тип ключа должен реализовывать интерфейс `WritableComparable`, производный от `Writable`. Типы ключей должны дополнительно поддерживать метод `compareTo()`, поскольку в разных местах каркаса MapReduce производится сортировка по ключам.

4.3. Подсчет всякой всячины

Неспециалист представляет себе статистику, как науку о подсчетах, и именно этим занимаются многие задачи для Hadoop. В главе 1 мы уже встречались с подсчетом слов. В случае данных о цитировании патентов было бы интересно знать, сколько имеется ссылок на каждый патент. Это тоже разновидность подсчета. Результат хорошо бы получить в таком виде:

```
1       2
10000   1
100000  1
1000006 1
1000007 1
1000011 1
1000017 1
1000026 1
1000033 2
1000043 1
1000044 2
1000045 1
1000046 2
1000049 1
1000051 1
1000054 1
1000065 1
1000067 3
```

В каждой записи с номером патента ассоциируется количество ссылок на него. Нетрудно написать MapReduce-программу для решения этой задачи. Выше мы уже отмечали, что не стоит всякий раз писать MapReduce-программу с нуля. У нас ведь уже есть программа, обрабатывающая данные похожим образом. Так скопируйте ее и модифицируйте под новые требования.

Мы уже написали программу для получения инвертированного индекса цитирования. Ее можно модифицировать так, чтобы вместо списка ссылающихся патентов она выводила их количество. Изменить придется только класс `Reduce`. Если мы решим выводить счетчик в виде `IntWritable`, то нужно будет указать тип `IntWritable` в трех местах кода класса `Reducer`. В приведенных выше обозначениях это тип `V3`.

```
public static class Reduce extends MapReduceBase
   implements Reducer<Text, Text, Text, IntWritable> {

   public void reduce(Text key, Iterator<Text> values,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
      int count = 0;
      while (values.hasNext()) {
         values.next();
         count++;
      }
      output.collect(key, new IntWritable(count));
   }
}
```

Изменив всего несколько строк и типы классов, мы получили новую MapReduce-программу. В данном случае модификация незначительна. Далее мы рассмотрим пример, где изменений будет больше, но базовая структура MapReduce-программ останется неизменной.

Выполнив этот пример, мы будем иметь набор данных о количестве ссылок на каждый патент. Было бы любопытно подсчитать счетчики. Давайте построим гистограмму распределения счетчиков ссылок. Естественно ожидать, что будет много патентов, цитированных только один раз, и мало – цитированных сотни раз. Интересно посмотреть, какое распределение получится в действительности.

Примечание. Поскольку данные о цитировании патентов охватывают лишь период с 1975 по 1999 год, оценка счетчика ссылок по необходимости будет заниженной. (Ссылки, оказавшиеся вне этого периода, не учитываются.) Мы также не рассматриваем патенты, на которые предположительно вообще нет ссылок. Несмотря на такие ограничения, анализ будет полезным.

Первым делом при написании любой MapReduce-программы нужно определиться с потоком данных. В данном случае распределитель, читая запись, игнорирует номер патента и выводит промежуточную пару ключ/значение <citation_count, 1>. Редуктор просуммирует количество пар с единицей в качестве счетчика ссылок и выведет итог.

Разобравшись с потоком данных, подумаем о типах ключей и значений — K1, V1, K2, V2, K3 и V3 во входных, промежуточных и выходных парах. Мы будем использовать класс KeyValueTextInputFormat, который автоматически выделяет из входных записей ключи и значения исходя из символа-разделителя. Для входного формата типы K1 и V1 будут совпадать с Text. В качестве типов K2, V2, K3 и V3 мы возьмем IntWritable, так как эти данные целочисленные и представлять их типом IntWritable более эффективно.

Приняв решение о потоке и типах данных, мы можем написать окончательную программу. Она показана в листинге 4.2. Как видите, структурно она аналогична всем предшествующим MapReduce-программам. Комментарии, касающиеся ее работы, приведены после листинга.

Листинг 4.2.
CitationHistogram.java: подсчет патентов, цитированных один раз, два раза и т. д.

```
public class CitationHistogram extends Configured implements Tool {

   public static class MapClass extends MapReduceBase
      implements Mapper<Text, Text, IntWritable, IntWritable> {

      private final static IntWritable uno = new IntWritable(1);
      private IntWritable citationCount = new IntWritable();

      public void map(Text key, Text value,
                      OutputCollector<IntWritable, IntWritable> output,
                      Reporter reporter) throws IOException {
                     citationCount.set(Integer.parseInt(value.toString()));
         output.collect(citationCount, uno);
      }
   }

   public static class Reduce extends MapReduceBase
      implements Reducer<IntWritable,IntWritable,IntWritable,IntWritable>
   {
      public void reduce(IntWritable key, Iterator<IntWritable> values,
                         OutputCollector<IntWritable,
                         IntWritable>output,
```

```
                        Reporter reporter) throws IOException {
        int count = 0;
        while (values.hasNext()) {
           count += values.next().get();
        }
        output.collect(key, new IntWritable(count));
     }
  }

  public int run(String[] args) throws Exception {
     Configuration conf = getConf();

     JobConf job = new JobConf(conf, CitationHistogram.class);

     Path in = new Path(args[0]);
     Path out = new Path(args[1]);
     FileInputFormat.setInputPaths(job, in);
     FileOutputFormat.setOutputPath(job, out);

     job.setJobName("CitationHistogram");
     job.setMapperClass(MapClass.class);
     job.setReducerClass(Reduce.class);
     job.setInputFormat(KeyValueTextInputFormat.class);
     job.setOutputFormat(TextOutputFormat.class);
     job.setOutputKeyClass(IntWritable.class);
     job.setOutputValueClass(IntWritable.class);

     JobClient.runJob(job);

     return 0;
  }

  public static void main(String[] args) throws Exception {
     int res = ToolRunner.run(new Configuration(),
                              new CitationHistogram(),
                              args);
     System.exit(res);
  }
}
```

Новый класс называется `CitationHistogram`; это имя всюду используется вместо `MyJob`. Метод `main()` почти всегда одинаковый. Драйвер также изменился мало. Входной и выходной формат по-прежнему представлены классами `KeyValueTextInputFormat` и `TextOutputFormat` соответственно. Основное изменение состоит в том, что выходной ключ и значение теперь принадлежат классу `IntWritable`, поскольку именно так выбраны типы `K2` и `V2`.

Мы также удалили строку

```
job.set("key.value.separator.in.input.line", ",");
```

В ней задается символ-разделитель для формата `KeyValueTextInputFormat`, с помощью которого из каждой входной строки выделяется ключ и значение. Раньше, при обработке исходных данных о цитировании патентов, в этом качестве выступала запятая. Если же не устанавливать это свойство, то по умолчанию подразумевается знак табуляции, что для данных о счетчиках ссылок нас вполне устраивает.

Поток данных в этом распределителе мало чем отличается от предыдущих, мы только определили в классе и используем две переменные: `citationCount` и `uno`.

```
public static class MapClass extends MapReduceBase
   implements Mapper<Text, Text, IntWritable, IntWritable> {

   private final static IntWritable uno = new IntWritable(1);
   private IntWritable citationCount = new IntWritable();

   public void map(Text key, Text value,
                   OutputCollector<IntWritable, IntWritable> output,
                   Reporter reporter) throws IOException {
      citationCount.set(Integer.parseInt(value.toString()));
      output.collect(citationCount, uno);
   }
}
```

В метод `map()` добавлена строка для установки `citationCount`, в которой осуществляется преобразование типа. Переменные `citationCount` и `uno` мы определили на уровне класса, а не внутри метода, исключительно из соображений эффективности. Метод `map()` будет вызываться столько раз, сколько есть записей (в одной порции, на каждой JVM). Уменьшив количество объектов, создаваемых внутри `map()`, мы повысим производительность и сэкономим на сборке мусора. Передавая `citationCount` и `uno` методу `output.collect()`, мы полагаемся на его обязательство не модифицировать эти объекты[4].

Редуктор суммирует все значения для каждого ключа. Вроде бы неэффективно, поскольку мы заведомо знаем, что все значения равны 1 (точнее, `uno`). Так зачем же суммировать? Мы пошли таким путем, потому что впоследствии будет проще добавить *комбинатор* для повышения производительности. В отличие от `MapClass`, в классе `Reduce` при обращении к методу `output.collect()` создается новый объект `IntWritable`, а не повторно используется уже существующий.

```
output.collect(key, new IntWritable(count));
```

[4] В разделе 5.1.3 мы увидим, что это обязательство запрещает классу `ChainMapper` использовать передачу по ссылке.

Мы могли бы улучшить производительность, заведя в классе переменную `IntWritable`. Но обращений к методу `reduce()` в этой конкретной программе гораздо меньше, наверное, всего несколько тысяч (и это для всех редукторов вместе). Что-то оптимизировать в этом месте нет необходимости.

Запуск задачи MapReduce для данных о счетчиках ссылок дает показанный ниже результат. Как мы и ожидали, количество патентов, на которые есть всего одна ссылка, очень велико (больше 900 тысяч). А патенты, на которые ссылались несколько сотен раз, исчисляются единицами. Самый популярный патент цитировался 779 раз.

```
1       921128
2       552246
3       380319
4       278438
5       210814
6       163149
7       127941
8       102155
9       82126
10      66634
...
411     1
605     1
613     1
631     1
633     1
654     1
658     1
678     1
716     1
779     1
```

Поскольку гистограмма насчитывает всего несколько сот строк, можно загрузить ее в электронную таблицу и построить график. На рис. 4.2 показано распределение патентов по частоте цитирования. График построен в двойном логарифмическом масштабе. Если в таком масштабе распределение выглядит как прямая линия, то говорят, что имеет место *степенная зависимость*. Гистограмма счетчиков ссылок близка к этому, хотя ее приблизительное сходство с параболой наводит также на мысль о *логнормальном* распределении.

Вы уже, вероятно, убедились, что MapReduce-программы зачастую совсем невелики, и для упрощения разработки имеет смысл придерживаться определенной структуры. Основная работа при написании таких программ – придумать, как должен выглядеть поток данных.

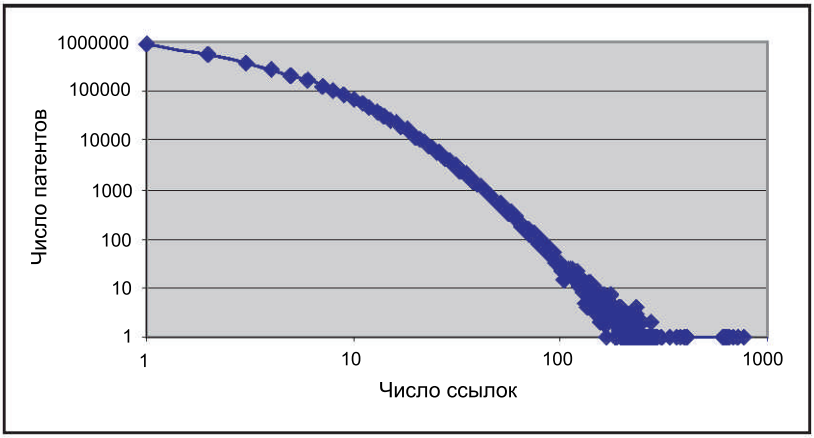


Рис. 4.2. График распределения патентов по частоте цитирования. На многие патенты есть всего одна ссылка (или ни одной, хотя на графике это не показано). На некоторые патенты имеется несколько сотен ссылок. В двойном логарифмическом масштабе график близок к прямой линии, что свидетельствует о степенном распределении.

4.4. Адаптация к изменениям в API Hadoop

Одна из основных движущих сил при проектировании версии Hadoop 1.0 – предложить стабильный и расширяемый API для работы с каркасом MapReduce. На момент написания этой книги последней была версия 0.20, она считается мостом между старым API (который используется в книге) и планируемым. Версия 0.20 поддерживает будущий API, но при этом сохраняет обратную совместимость с прежним, лишь помечая его как нерекомендуемый. В версиях после 0.20 поддержка старого API будет прекращена. Но пока мы не рекомендуем переходить на новый API по двум причинам:

1. Многие библиотечные классы в самом Hadoop еще не переписаны под новый API. Вы не сможете воспользоваться этими классами, если будете использовать в своих MapReduce-программах новый API, появившийся в версии 0.20.
2. Многие все еще считают, что самая стабильная и пригодная для промышленной эксплуатации версия Hadoop 0.18.3. Некоторые пользователи с энтузиазмом бросились переходить на

версию 0.20, но мы советуем еще немного погодить с внедрением ее в производственные системы[5].

К моменту выхода книги из печати ситуация может измениться. В этом разделе мы рассмотрим изменения, которые несет с собой новый API. К счастью, почти все изменения влияют только на основной шаблон MapReduce-программ. Мы перепишем этот шаблон, чтобы вы могли использовать его в будущем.

Первое, что бросается в глаза в новом API, – перенос многих классов из пакета `org.apache.hadoop.mapred` в другое место. Часть классов теперь находится в пакете `org.apache.hadoop.mapreduce`, а библиотечные классы – в различных подпакетах `org.apache.hadoop.mapreduce.lib`. После перехода на новый API не должно быть предложений `import` (или полных ссылок) для классов из пакета `org.apache.hadoop.mapred`, все они объявлены нерекомендуемыми.

Самое существенное изменение в новом API – появление *контекстных* объектов. Нагляднее всего это проявляется в замене объектов `OutputCollector` и `Reporter` в методах `map()` и `reduce()`. Теперь для вывода пар ключ/значение следует обращаться к методу `Context.write()`, а не `OutputCollector.collect()`. В долгосрочной перспективе это означает унификацию взаимодействия между вашим кодом и каркасом MapReduce и стабилизацию интерфейсов `Mapper` и `Reducer` таким образом, что сигнатуры объявленных в них методов не будут изменяться при добавлении новой функциональности. Расширение функциональности приведет только к появлению дополнительных методов у контекстных объектов. Программы, написанные до появления этой функциональности, ничего не будут знать о новых методах, но будут без ошибок компилироваться и исполняться в новых версиях.

Новые методы `map()` и `reduce()` объявлены в новых *абстрактных классах* `Mapper` и `Reducer` соответственно. Они заменяют *интерфейсы* `Mapper` и `Reducer`, присутствовавшие в первоначальном API (`org.apache.hadoop.mapred.Mapper` и `org.apache.hadoop.mapred.Reducer`). Новые абстрактные классы также делают излишним класс `MapReduceBase`, который объявлен нерекомендуемым.

В новые методы `map()` и `reduce()` внесено несколько мелких изменений. Они могут возбуждать исключение `InterruptedExcep-`

[5] Возможно, вас интересует вопрос о версии 0.19. Существует общее мнение, что первый выпуск этой версии изобиловал ошибками. В дополнительных выпусках были предприняты усилия по исправлению ошибок, но сообщество склонно переходить сразу к версии 0.20.

tion вместо IOException. Кроме того, метод reduce() принимает список значений в виде объекта Iterable, а не Iterator, что позволяет использовать для итерирования предложение foreach в Java. Резюмируем все рассмотренные выше изменения, приведя сигнатуры методов классов MapClass и Reduce. Сначала напомним, как эти сигнатуры выглядели в старом API:

```
public static class MapClass extends MapReduceBase
   implements Mapper<K1, V1, K2, V2> {

   public void map(K1 key, V1 value,
                   OutputCollector<K2, V2> output,
                   Reporter reporter) throws IOException { }
}

public static class Reduce extends MapReduceBase
   implements Reducer<K2, V2, K3, V3> {

   public void reduce(K2 key, Iterator<V2> values,
                      OutputCollector<K3, V3> output,
                      Reporter reporter) throws IOException { }
}
```

В новом API они стали немного проще:

```
public static class MapClass extends Mapper<K1, V1, K2, V2> {

   public void map(K1 key, V1 value, Context context)
                   throws IOException, InterruptedException { }
}

public static class Reduce extends Reducer<K2, V2, K3, V3> {

   public void reduce(K2 key, Iterable<V2> values, Context context)
                      throws IOException, InterruptedException { }
}
```

Чтобы поддержать новый API, понадобится также внести некоторые изменения в драйвер. Классы JobConf и JobClient теперь заменены. Их функциональность перенесена в класс Configuration (который первоначально был родительским классом JobConf) и в новый класс Job. Класс Configuration занимается только конфигурированием задачи, тогда как класс Job определяет план исполнения задачи и управляет ее исполнением. Методы setOutputKeyClass() и setOutputValueClass() перенесены из JobConf в Job. За конструирование задачи и передачу ее на исполнение теперь отвечает класс Job. Раньше задача конструировалась с помощью JobConf:

```
JobConf job = new JobConf(conf, MyJob.class);
job.setJobName("MyJob");
```

Теперь для этого применяется `Job`:

```
Job job = new Job(conf, "MyJob");
job.setJarByClass(MyJob.class);
```

Раньше класс `JobClient` передавал задачу на исполнение:

```
JobClient.runJob(job);
```

Теперь это тоже делает `Job`:

```
System.exit(job.waitForCompletion(true)?0:1);
```

В листинге 4.3 приведен шаблон программы на основе API, появившегося в версии Hadoop 0.20. Здесь учтены все упомянутые в этом разделе изменения.

Листинг 4.3.
Базовый шаблон программы для Hadoop (ср. с листингом 4.1), переписанный с учетом нового API версии 0.20

```
public class MyJob extends Configured implements Tool {

   public static class MapClass
                       extends Mapper<LongWritable, Text, Text, Text> {

      public void map(LongWritable key, Text value, Context context)
                     throws IOException, InterruptedException {

         String[] citation = value.toString().split(",");
         context.write(new Text(citation[1]), new Text(citation[0]));
      }
   }

   public static class Reduce extends Reducer<Text, Text, Text, Text> {

      public void reduce(Text key, Iterable<Text> values,
                         Context context)
                        throws IOException, InterruptedException {

         String csv = "";
         for (Text val:values) {
            if (csv.length() > 0) csv += ",";
               csv += val.toString();
         }

         context.write(key, new Text(csv));
      }
   }

   public int run(String[] args) throws Exception {
```

Iterable позволяет использовать цикл foreach

```
        Configuration conf = getConf();

        Job job = new Job(conf, "MyJob");
        job.setJarByClass(MyJob.class);

        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        System.exit(job.waitForCompletion(true)?0:1);

        return 0;
    }

    public static void main(String[] args) throws Exception {
       int res = ToolRunner.run(new Configuration(), new MyJob(), args);

        System.exit(res);
    }
}
```

1 Совместим с классом InputFormat

Этот код реализует ту же функцию построения инвертированного индекса, что и программа в листинге 4.1, но с использованием API версии 0.20. К сожалению, класс `KeyValueTextInputFormat`, которым мы пользовались в листинге 4.1, еще не перенесен на новый API. Нам пришлось переделать шаблон под класс `TextInputFormat` 1. Мы ожидаем, что в версии 0.21 все классы Hadoop будут поддерживать новый API. Чтобы сохранить единообразие, мы в дальнейшем будем продолжать использовать API, существовавший до выхода версии 0.20.

4.5. Интерфейс Hadoop Streaming

Все наши программы для Hadoop написаны на языке Java. Но Hadoop поддерживает и другие языки с помощью обобщенного API, получившего название Streaming. На практике интерфейс Streaming особенно полезен для создания простых коротких MapReduce-программ, которые быстрее всего написать на каком-нибудь скриптовом языке, способном воспользоваться преимуществами библиотек на языках, отличных от Java.

В API Hadoop Streaming взаимодействие с программами строится на основе парадигмы потоков, заимствованной из Unix. Входные данные подаются через `STDIN`, а выходные выводятся в `STDOUT`. Данные должны быть текстовыми, а каждая строка считается записью. Именно так работают многие команды Unix, и Hadoop Streaming позволяет использовать их в качестве распределителей и редукторов. Если вы знакомы с такими командами Unix, как `wc`, `cut` или `uniq`, то можете применить к обработке больших наборов данных с помощью Hadoop Streaming.

Общий поток данных в Hadoop Streaming напоминает конвейер: входные данные сначала поступают распределителю, результат его работы сортируется и подается на вход редуктора. На псевдокоде с использованием обозначений, принятых в Unix, это выглядит так:

```
cat [input_file] | [mapper] | sort | [reducer] >[output_file]
```

В примерах ниже демонстрируется использование Streaming совместно с командами Unix.

4.5.1. Интерфейс Streaming и команды Unix

В первом примере мы получим список патентов из файла `cite75_99.txt`, на которые имеются ссылки.

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
          ➥ -input input/cite75_99.txt
          ➥ -output output
          ➥ -mapper 'cut -f 2 -d ,'
          ➥ -reducer 'uniq'
```

Вот и все! Команда состоит из единственной строки. Посмотрим, что делает каждая ее часть.

Streaming API находится в дополнительном пакете `contrib/streaming/hadoop-*-streaming.jar`. Первая часть команды, а также аргументы `-input` и `-output` говорят о том, что мы запускаем Streaming-программу с указанными входным и выходным файлом или каталогом. Распределитель и редуктор заданы в отдельных аргументах и заключены в кавычки. Мы видим, что в качестве распределителя взята команда Unix `cut`, которая извлекает данные из второго столбца в предположении, что столбцы разделены запятыми. В наборе данных о цитировании второй столбец содержит номер цитируемого патента. Затем номера патентов сортируются и передаются редуктору. В качестве редуктора задана команда `uniq`, которая удаляет дубликаты из отсортированных данных. На выходе получаем:

```
"CITED"
1
10000
100000
1000006
...
999973
999974
999977
999978
999983
```

В первой строке оказался описатель «CITED» из исходного файла. Отметим, что строки отсортированы лексикографически, потому что Streaming обрабатывает все данные как текстовые и о других типах ничего не знает.

Получив список цитируемых патентов, мы можем узнать, сколько таких патентов существует. И снова на помощь приходит интерфейс Streaming – для быстрого подсчета строка мы воспользуемся командой Unix `wc -l`.

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
          ➥ -input output
          ➥ -output output_a
          ➥ -mapper 'wc -l'
          ➥ -D mapred.reduce.tasks=0
```

Здесь команда `wc -l` играет роль распределителя для подсчета количества записей в каждой порции. Интерфейс Hadoop Streaming (начиная с версии 0.19.0) поддерживает класс `GenericOptionsParser`. Указанный в предыдущей команде аргумент `-D` служит для задания конфигурационных свойств. Мы хотели, чтобы распределитель выводил данные напрямую, без редуктора, поэтому присвоили свойству `mapred.reduce.tasks` значение 0, а аргумент `-reducer` вообще опустили[6]. Окончательно получилось 3258984. Согласно нашим данным, ссылки имеются более чем на 3 миллиона патентов.

4.5.2. Streaming и скрипты

Совместно с Hadoop Streaming можно использовать любой исполняемый скрипт, который обрабатывает разбитые на строки данные, посту-

[6] Возможно, вы обратили внимание, что при таком подходе подсчитывается количество записей в каждой порции, а не во всем файле. Если файл велик или файлов несколько, то пользователю придется самостоятельно просуммировать частичные результаты для получения общего итога. Чтобы полностью автоматизировать подсчет, необходимо написать скрипт-редуктор, который будет складывать частичные счетчики.

пающие из STDIN, и выводит их на STDOUT. Например, в листинге 4.4 приведен скрипт на языке Python, который производит случайную выборку данных из STDIN. Для тех, кто незнаком с Python, поясним, что в цикле for выполняется построчное чтение из STDIN. Прочитав очередную строку, мы выбираем случайное число от 1 до 100 и сравниваем его с аргументом, заданным в командной строке (sys.argv[1]). В зависимости от результата сравнения мы либо передаем строку на выход, либо игнорируем ее. С помощью этого скрипта можно сделать случайную выборку из строчного файла данных, например:

```
cat input.txt | RandomSample.py 10 >sampled_output.txt
```

Эта команда выполняет скрипт с аргументом 10; в файле sampled_output.txt окажется приблизительно 10 процентов записей из файла input.txt. Вообще, задав в качестве аргумента целое число от 1 до 100, мы получим на выходе соответствующий процент входных записей.

Листинг 4.4.
RandomSample.py: скрипт на языке Python для вывода случайно выбранных из STDIN строк

```
#!/usr/bin/env python
import sys, random

for line in sys.stdin:
    if (random.randint(1,100) <= int(sys.argv[1])):
       print line.strip()
```

Этим скриптом можно воспользоваться в Hadoop для получения небольшой выборки из всего набора данных. Часто такая выборка оказывается полезна в процессе разработки, поскольку запустить Hadoop-программу для меньшего набора данных можно в автономном или псевдораспределенном режиме, в котором проще и быстрее отлаживаться. Кроме того, когда необходимо получить лишь качественное представление о данных, скорость и удобство обработки небольшого набора оказываются важнее, чем некоторая потеря точности. Один из примеров такой качественной информации – выявление имеющихся в данных кластеров. В пакетах R, MATLAB и других имеются оптимизированные реализации многих алгоритмов кластеризации. Поэтому гораздо разумнее сделать небольшую выборку и применить к ней стандартный пакет, чем пытаться обработать в Hadoop все данные с помощью какого-то распределенного алгоритма.

Предупреждение. Приемлема ли потеря точности вследствие обработки выборочных данных, зависит от того, что именно вы пытаетесь вычислить, а также от распределения данных в исходном наборе. Например, обычно среднее допустимо вычислять по выборочным данным, но если распределение данных далеко от равномерного и среднее определяется главным образом несколькими выделяющимися значениями, то вычисления по выборке могут дать неверный результат. Аналогично кластеризация по выборочным данным приемлема, если требуется всего лишь получить о них общее представление. Но если вы ищете небольшие аномальные кластеры, то такой подход не годится – в результате выборки они могут пропасть. Вычислять минимум и максимум по выборочным данным вообще никогда не стоит.

Запуск скрипта `RandomSample.py` с помощью интерфейса Streaming аналогичен запуску команд Unix; единственное различие состоит в том, что команды Unix уже имеются на всех узлах кластера, тогда как о скрипте `RandomSample.py` этого не скажешь. Hadoop Streaming поддерживает флаг `-file`, позволяющий включать исполняемый файл в один пакет с задачей[7]. Команда для исполнения нашего скрипта `RandomSample.py` выглядит так:

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
        ➥ -input input/cite75_99.txt
        ➥ -output output
        ➥ -mapper 'RandomSample.py 10'
        ➥ -file RandomSample.py
        ➥ -D mapred.reduce.tasks=1
```

Задав в качестве распределителя команду `'RandomSample.py 10'`, мы включаем в выборку 10 процентов исходных данных. Отметим, что число редукторов (`mapred.reduce.tasks`) задано равным 1. Поскольку явно редуктор не указан, по умолчанию используется `IdentityReducer`, который копирует вход на выход. В данном случае мы можем задать любое ненулевое число редукторов и получим столько же выходных файлов. С другой стороны, можно задать 0 редукторов, тогда выходных файлов будет столько, сколько имеется распределителей. Для задания выборки это, пожалуй, не идеальное решение, потому что на выходе каждого распределителя оказывается лишь небольшая часть входных данных, так что может образоваться много маленьких файлов. Впрочем, впоследствии это легко исправить, воспользовавшись командой HDFS `getmerge` или другими командами для работы с файлами, чтобы получить столько файлов, сколько нужно. Какой подход выбрать, дело вкуса.

Скрипт для случайной выборки мы написали на языке Python, но подошел бы и любой другой язык, умеющий работать с `STDIN` и

[7] Кроме того, неявно предполагается, что во всех узлах кластера уже установлен Python.

STDOUT. Для иллюстрации реализуем тот же скрипт на PHP[8] (листинг 4.5). Запустите этот скрипт командой

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
        ➥ -input input/cite75_99.txt
        ➥ -output output
        ➥ -mapper 'php RandomSample.php 10'
        ➥ -file RandomSample.php
        ➥ -D mapred.reduce.tasks=1
```

Листинг 4.5.
RandomSample.php: PHP-скрипт для вывода случайно выбранных из STDIN строк

```
<?php

while (!feof(STDIN)) {
   $line = fgets(STDIN);
   if (mt_rand(1,100) <= $argv[1]) {
      echo $line;
   }
}
```

Скрипты для случайной выборки не нуждаются в каком-то специальном редукторе, но так бывает не во всех Streaming-программах. Поскольку на практике вы часто будете использовать интерфейс Streaming, рассмотрим еще один пример, на этот раз со специальным редуктором. Предположим, что нас интересует максимальное число пунктов в формуле изобретения, изложенной в одном патенте. В наборе данных о патентах количество пунктов указано в девятом столбце. Наша цель – найти максимум по всем значениям в девятом столбце описаний патентов.

При работе со Streaming каждый распределитель видит весь поток данных, и ответственность за разбиение потока на записи возлагается

[8] Вы, вероятно, заметили, что в листинге 4.5 отсутствует закрывающая скобка ?>, соответствующая открывающей скобке <?php. Напомним, что первоначально язык PHP предназначался для встраивания кода в статическое HTML-содержимое. Все, что находится вне программных скобок <?p...?>, считается статическим содержимым и выводится буквально. При использовании PHP в качестве чистого скриптового языка надо внимательно следить за тем, чтобы вне скобок не было пробелов. В противном случае они будут скопированы в выходной поток, что может привести к нежелательному поведению, с трудом поддающемуся отладке. (Создается впечатление, что пробелы в выходных данных появились из ниоткуда.)

Гарантировать отсутствие пробелов перед открывающей скобкой <?php просто – достаточно поместить ее в самое начало скрипта. Но пробелы после закрывающей скобки ?> легко можно не заметить. Поэтому при использовании PHP-скрипта безопаснее вообще опускать закрывающую скобку. Тогда интерпретатор PHP будет считать, что все символы вплоть до конца файла составляют часть программы, а не статическое содержимое.

на распределитель. В стандартной модели Java сам каркас разбивает входные данные на записи и передает методу `map()` по одной записи при каждом вызове. Модель Streaming упрощает запоминание информации о состоянии между обработкой различных записей порции, и мы можем воспользоваться этим фактом при вычислении максимума. Стандартная модель Java также способна запоминать состояние между обработкой записей порции, но делается это сложнее. Как именно, мы рассмотрим в следующей главе. При создании Hadoop-программы вычисления максимума мы воспользуемся свойством *дистрибутивности* максимума. Если имеется набор данных, разбитый на несколько порций, то максимум по всему набору равен максимуму из максимумов по отдельным порциям. При наличии четырех записей X1, X2, X3, X4, разбитых на порции (X1, X2) и (X3, X4), максимум по всем четырем записям можно найти, зная максимумы по каждой порции:

```
max(X1,X2,X3,X4) = max(max(X1,X2), max(X3,X4))
```

Наша стратегия будет заключаться в том, чтобы вычислять максимумы по каждой порции. В конце работы распределитель выведет единственное значение. Затем нам понадобится один редуктор, который вычислит максимум по этим значениям и выведет результат. В листинге 4.6 приведен скрипт на Python, реализующий распределитель для вычисления максимума по одной порции.

Листинг 4.6.
AttributeMax.py: скрипт на Python для вычисления максимального значения атрибута

```
#!/usr/bin/env python

import sys

index = int(sys.argv[1])
max = 0
for line in sys.stdin:
    fields = line.strip().split(",")
    if fields[index].isdigit():
       val = int(fields[index])
       if (val > max):
          max = val
else:
    print max
```

Скрипт совсем простой. В цикле `for` читаются записи – по одной за раз. Каждая запись разбивается на поля; если значение заданного

пользователем поля больше текущего максимума, то максимум обновляется. Отметим, что распределитель выводит результат – максимальное значение в порции – только по завершении цикла. Раньше мы поступали по-другому – при обработке каждой записи выводилась одна или несколько промежуточных записей для последующей обработки редукторами.

При столь лаконичном распределителе мы можем воспользоваться стандартным редуктором `IdentityReducer` для обработки отсортированного результата работы распределителей:

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
      ➥ -input input/apat63_99.txt
      ➥ -output output
      ➥ -mapper 'AttributeMax.py 8'
      ➥ -file playground/AttributeMax.py
      ➥ -D mapred.reduce.tasks=1
```

В качестве распределителя используется команда `'AttributeMax.py 8'`. Она выводит максимальное значение в девятом столбце по всем записям в одной порции. Результаты работы всех распределителей собирает один редуктор. Если имеется семь распределителей, то конечный результат работы этой команды выглядит так:

```
0
260
306
348
394
706
868
```

В каждой строке находится максимум, вычисленный по одной порции. Как видим, имеется одна порция, для которой во всех записях число пунктов формулы изобретения равно нулю. На первый взгляд, подозрительно, но не надо забывать, что число пунктов не фиксировалось в патентах, выданных до 1975 года.

Видно, что наш распределитель работает правильно. Теперь можно воспользоваться редуктором, который выведет максимум по всем значениям, вычисленным распределителями. Здесь наблюдается любопытная ситуация: благодаря свойству дистрибутивности максимума мы можем в качестве редуктора взять тот же самый скрипт `AttributeMax.py`. Только теперь он будет искать максимум по первому столбцу.

```
bin/hadoop jar contrib/streaming/hadoop-0.18.1-streaming.jar
```

```
    ➥ -input input/apat63_99.txt
    ➥ -output output
    ➥ -mapper 'AttributeMax.py 8'
    ➥ -reducer 'AttributeMax.py 0'
    ➥ -file AttributeMax.py
    ➥ -D mapred.reduce.tasks=1
```

Эта команда выводит файл из одной строки и оказывается, что максимальное число пунктов формулы изобретения в одном патенте равно 868.

Классификация агрегатных функций

Для вычисления статистических характеристик мы пользуемся агрегатными функциями. Обычно выделяют три класса таких функций: дистрибутивные, алгебраические и холистические. Максимум – пример дистрибутивной функции. Другими дистрибутивными функциями являются минимум, сумма и счетчик. Как следует из названия, дистрибутивная функция обладает свойством дистрибутивности. Любую дистрибутивную функцию можно вычислять итеративно, применяя ее к подмножеству данных, – как мы видели на примере максимума.

К алгебраическим функциям относятся, например, среднее и дисперсия. Они не обладают свойством дистрибутивности и подразумевают некоторые «вычисления» с более простыми функциями. Примеры мы рассмотрим в следующем разделе.

Наконец, функции типа медианы и нахождения *K* наименьших или наибольших значений принадлежат классу холистических агрегатных функций. Любители сложных задач могут попытаться придумать эффективную реализацию функции вычисления медианы с помощью Hadoop.

4.5.3. Интерфейс Streaming и пары ключ/значение

Сейчас вы, наверное, задаетесь вопросом, что случилось со способом кодирования записей в виде пар ключ/значение. До сих пор при обсуждении интерфейса Streaming мы рассматривали каждую запись как атомарную единицу, а не композицию ключа и значения. Но на самом деле Streaming работает с парами ключ/значение точно так же, как стандартная модель Java MapReduce. По умолчанию в Streaming для разделения ключа и значения в записи используется знак табуляции. Если знака табуляции нет, что вся запись считается ключом, а значением будет пустая строка. В нашем наборе данных знаков табуляции не было, поэтому создалось впечатление, будто мы обрабатываем запись как неделимое целое. Но даже если в записях имеются знаки табуляции, то Streaming API просто будет

тасовать и сортировать записи в другом порядке. При условии, что распределитель и редуктор работают с записями, можно поддерживать иллюзию «записе-ориентированности».

Работа с парами ключ/значение позволяет воспользоваться преимуществами сортировки и тасования по ключу для выполнения интересных видов анализа данных. Чтобы продемонстрировать обработку ключей и значений с помощью Streaming, напишем программу вычисления максимального числа пунктов в формуле изобретения *для каждой страны*. В отличие от скрипта `AttributeMax.py` эта программа будет искать максимум для каждого ключа, а не по всем записям. Чтобы сделать упражнение еще более интересным, давайте вычислим не максимум, а среднее. (Как мы увидим, в Hadoop уже включен пакет Aggregate, содержащий классы, упрощающие поиск максимума для каждого ключа.)

Но сначала изучим, как пары ключ/значение работают со Streaming API на каждом шаге потока данных в каркасе MapReduce.

1. Как мы видели, распределитель в Streaming читает порцию из `STDIN`, извлекая каждую строку в виде записи. Ваш распределитель может по своему усмотрению интерпретировать входную запись либо как строку текста, либо как пару ключ/значение.
2. Streaming API интерпретирует каждую строку, порождаемую распределителем, как пару ключ/значение, разделенные знаком табуляции. Как и в стандартной модели MapReduce, мы применяем разбиватель к ключу, чтобы найти редуктор, которому следует передать запись. Все пары с одним и тем же ключом попадают одному и тому же редуктору.
3. До передачи редуктору Streaming API сортирует пары ключ/значение по ключу. Напомним, что в модели Java все пары с одинаковым ключом группируются в одну пару с этим ключом и списком значений в качестве значения. Затем эта группа передается методу `reduce()`. В Streaming API *за выполнение такой группировки отвечает ваш редуктор*. Это не так уж плохо, потому что пары ключ/значение уже отсортированы по ключу. Все записи с одинаковым ключом находятся в одной непрерывной последовательности. Ваш редуктор должен будет читать по одной строке из `STDIN` и следить за тем, когда ключ сменится.
4. На практике выходом (`STDOUT`) вашего редуктора обычно является файл. Технически перед записью в файл произво-

дится еще один шаг, на котором Streaming API разбивает каждую строку данных, порожденных редуктором, по знаку табуляции и подает получившуюся пару ключ/значение на вход подразумеваемого по умолчанию форматера вывода `TextOutputFormat`, который заново вставляет знак табуляции перед тем, как записать результат в файл. Если в порождаемых редуктором данных нет знака табуляции, то этот шаг не дает никакого полезного результата. Такое поведение по умолчанию можно изменить и делать что-то иное, но лучше оставить его, как есть, а содержательную обработку перенести в сам редуктор.

Чтобы лучше разобраться в потоке данных, напишем Streaming-программу, которая будет вычислять среднее количество пунктов в формуле изобретения для каждой страны. Распределитель будет извлекать страну и количество пунктов из каждого патента и представлять их в виде пары ключ/значение. В соответствии с принятым в Streaming соглашением распределитель при выводе вставляет между ключом и значением знак табуляции. Streaming API выделит ключ, а процедура тасования гарантирует, что все счетчики пунктов для одной страны попадут одному и тому же редуктору. Код на Python представлен в листинге 4.7. Для каждой записи распределитель извлекает страну (`fields[4][1:-1]`), делая ее ключом, и число пунктов в формуле изобретения (`fields[8]`), которое становится значением. Надо еще учесть, что в нашем наборе данных встречаются отсутствующие данные. Поэтому мы добавили условие для пропуска записей без счетчика пунктов.

Листинг 4.7.
AverageByAttributeMapper.py: вывод страны и числа пунктов в формуле изобретения

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    fields = line.split(",")
        if (fields[8] and fields[8].isdigit()):
           print fields[4][1:-1] + "\t" + fields[8]
```

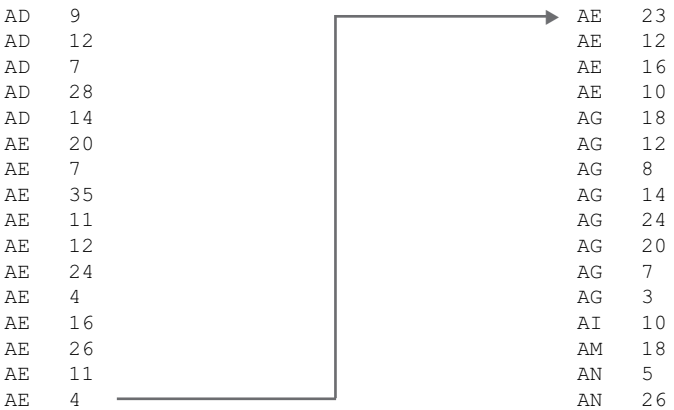
Прежде чем переходить к редуктору, запустим этот распределитель в двух ситуациях: без редуктора и со стандартным редуктором `IdentityReducer`. Это полезно, так как мы сможем точно узнать, что

именно выводит распределитель (если редуктора нет) и что подается на вход редуктора (если используется `IdentityReducer`). Вы оцените этот прием по достоинству, когда начнете отлаживать собственные MapReduce-программы. По крайней мере, будет понятно, выводит ли распределитель ожидаемые данные и попадает ли на вход редуктора то, что вы предполагаете. Сначала запустим распределитель без редуктора.

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
       ➥ -input input/apat63_99.txt
       ➥ -output output
       ➥ -file playground/AverageByAttributeMapper.py
       ➥ -mapper 'AverageByAttributeMapper.py'
       ➥ -D mapred.reduce.tasks=0
```

На выходе должны получаться строки, содержащие код страны, за которым следует знак табуляции и числовое значение счетчика. Выходные записи не отсортированы по (новому) ключу. На самом деле, они следуют в том же порядке, что входные записи, хотя при взгляде на результат это не очевидно.

Интереснее, однако, задать ненулевое число редукторов `IdentityReducer`. Тогда мы увидим, какие отсортированные и перетасованные записи поступают на вход редуктора. Чтобы не усложнять себе жизнь, зададим один редуктор (`-D mapred.reduce.tasks=1`) и посмотрим на первые 32 записи.



При работе со Streaming API редуктор увидит на `STDIN` эти текстовые данные. Мы должны написать редуктор так, чтобы он восстанавливал пары ключ/значение, разбивая строки по знакам табуляции. Процедура сортировки «сгруппировала» записи с одинаковыми клю-

чами. При чтении из STDIN мы должны обнаруживать точки смены ключа. Отметим, что записи отсортированы только по ключам, порядок значений не определен. Наконец, редуктор должен производить заявленное вычисление, то есть подсчитывать среднее значение для каждого ключа. В листинге 4.8 представлен полный код редуктора на языке Python.

Листинг 4.8.
AverageByAttributeReducer.py

```
#!/usr/bin/env python

import sys

(last_key, sum, count) = (None, 0.0, 0)

for line in sys.stdin:
    (key, val) = line.split("\t")

    if last_key and last_key != key:
       print last_key + "\t" + str(sum / count)
      (sum, count) = (0.0, 0)

    last_key = key
    sum += float(val)
    count += 1

print last_key + "\t" + str(sum / count)
```

Эта программа подсчитывает сумму и количество значений для каждого ключа. Обнаружив во входном потоке новый ключ или конец файла, она вычисляет среднее для предыдущего ключа и записывает результат в STDOUT. По завершении всей задачи MapReduce можно легко проверить правильность нескольких первых результатов.

```
AD 14.0
AE 15.4
AG 13.25
AI 10.0
AM 18.0
AN 9.625
```

Примечание. Для интересующихся сообщим, что на сайте NBER, откуда были взяты данные о патентах, имеется также файл list_of_countries.txt, где приведены полные названия стран, соответствующие их кодам. Сравнив результат нашей задачи с расшифровкой кодов стран, мы увидим, что в патентах, выданных в Андорре (AD), количество пунктов в формуле изобретения в среднем равно 14, а Арабских Эмиратах (AE) – 15,4, в Антигуа и Барбуда – 13,25 и т. д.

4.5.4. Интерфейс Streaming и пакет Aggregate

В состав Hadoop включен библиотечный пакет Aggregate, упрощающий получение агрегированных статистик набора данных. С его помощью легко писать на Java сборщики статистики, особенно в сочетании со Streaming. Именно этим мы и займемся в настоящем разделе[9].

При совместной работе со Streaming пакет Aggregate играет роль редуктора, вычисляющего агрегированные статистики. Вы должны лишь предоставить распределитель, который будет обрабатывать записи и выводить их в специальном формате. Каждая строка, выводимая распределителем, должна иметь вид:

```
function:key\tvalue
```

В начале выходной строки находится имя функции агрегатора значений (из числа предопределенных функций, входящих в состав пакета Aggregate). Затем идет двоеточие, а за ним ключ и значение, разделенные знаком табуляции. Редуктор из пакета Aggregate применяет указанную функцию к множеству значений с одинаковым ключом. Например, если задана функция `LongValueSum`, то результатом будут суммы значений для каждого ключа. (При этом считается, что каждое значение имеет стандартный тип Java `long`.) А в случае функции `LongValueMax` на выходе получатся максимальные значения для каждого ключа. Список агрегатных функций, включенных в пакет Aggregate, приведен в табл. 4.3.

Таблица 4.3. Агрегатные функции, поддерживаемые пакетом Aggregate

Агрегатор значений	Описание
`DoubleValueSum`	Вычисляет сумму последовательности значений типа double.
`LongValueMax`	Находит максимум в последовательности значений типа long.
`LongValueMin`	Находит минимум в последовательности значений типа long.
`LongValueSum`	Вычисляет сумму последовательности значений типа long.
`StringValueMax`	Находит лексикографический максимум в последовательности значений типа string.

[9] О работе с пакетом Aggregate рассказывается на странице http://hadoop.apache.org/core/docs/current/api/ org/apache/hadoop/mapred/lib/aggregate/package-summary.html.

Таблица 4.3. (окончание).

Агрегатор значений	Описание
`StringValueMin`	Находит лексикографический минимум в последовательности значений типа string.
`UniqValueCount`	Подсчитывает количество уникальных значений (для каждого ключа).
`ValueHistogram`	Находит количество, минимум, максимум, медиану, среднее и стандартное отклонение значений для каждого ключа. (Дополнительные пояснения см. в тексте.)

Рассмотрим на примере, как легко пользоваться пакетом Aggregate. Мы хотим узнать, сколько патентов выдавалось каждый год. Здесь можно применить тот же подход, что и в задаче о подсчете слов из главы 1. Для каждой записи распределитель будет выводить год выдачи патента в качестве ключа и «1» в качестве значения. Редуктор просуммирует все значения (единицы) и тем самым получит искомый счетчик. Только теперь мы воспользуемся интерфейсом Streaming в сочетании с пакетом Aggregate. Код распределителя показан в листинге 4.9.

Листинг 4.9.
AttributeCount.py

```
#!/usr/bin/env python

import sys

index = int(sys.argv[1])
for line in sys.stdin:
    fields = line.split(",")
    print "LongValueSum:" + fields[index] + "\t" + "1"
```

Скрипт `AttributeCount.py` будет работать для любого входного файла в формате CSV. Пользователю остается только указать индекс столбца атрибута, для которого вычисляется счетчик. Основное действие этой коротенькой программы – предложение `print`. Оно говорит пакету Aggregate, что требуется просуммировать все значения (единицы) для каждого ключа, который выбирается из указанного пользователем столбца (с номером `index`). Чтобы подсчитать количество патентов, выданных в каждом году, мы запустим Streaming-программу с пакетом Aggregate и сообщим распределителю, что интересующий атрибут находится во втором столбце (`index` = 1) входного файла.

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
       ➥ -input input/apat63_99.txt
       ➥ -output output
       ➥ -file AttributeCount.py
       ➥ -mapper 'AttributeCount.py 1'
       ➥ -reducer aggregate
```

Почти все аргументы этой Streaming-программы нам уже знакомы. Главное, на что следует обратить внимание, – название редуктора: `'aggregate'`. Оно является для Streaming API указанием на необходимость использовать пакет Aggregate. На выходе задачи MapReduce (после сортировки) получается такой результат:

```
"GYEAR" 1
1963    45679
1964    47375
1965    62857
...
1996    109645
1997    111983
1998    147519
1999    153486
```

Первая строка выбивается из общего ряда, потому что в первой строке входного файла находятся описания столбцов. Но во всех остальных отношениях эта MapReduce-задача отлично справилась с подсчетом выданных патентов по годам. На рис. 4.3 показан график, позволяющий наглядно представить данные. Как видите, наблюдается отчетливая тенденция к повышению количества патентов.

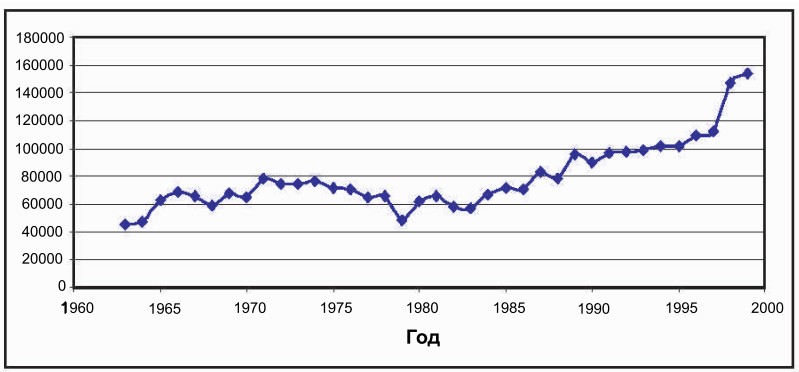


Рис. 4.3. Для подсчета количества выданных патентов по годам, использовался Hadoop, а для построения графика – Excel. Анализ показывает, что за 40 лет количество выдаваемых патентов возросло почти в четыре раза.

Легко видеть, что большинство функций в пакете Aggregate (см. табл. 4.3) представляют собой вычисление максимума, минимума и суммы для разных типов данных. (По какой-то причине функции `DoubleValueMax` и `DoubleValueMin` отсутствуют, хотя их можно добавить путем тривиальной модификации `LongValueMax` и `LongValueMin`.) Но функции `UniqValueCount` и `ValueHistogram` выделяются из общего ряда, и ниже мы приведем несколько примеров их использования.

`UniqValueCount` вычисляет количество уникальных значений для каждого ключа. Например, пусть мы хотим узнать, увеличилось ли со временем количество стран, пользующихся услугами патентной системы США. Для этого достаточно проанализировать по годам, в каких странах проживают авторы изобретений. В листинге 4.10 показан скрипт, являющийся бесхитростной оберткой вокруг функции `UniqValueCount`. Этот скрипт применен к столбцам файла `apat63_99.txt`, содержащим год и код страны (их индексы равны соответственно 1 и 4).

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
    ➥ -input input/apat63_99.txt
    ➥ -output output
    ➥ -file UniqueCount.py
    ➥ -mapper 'UniqueCount.py 1 4'
    ➥ -reducer aggregate
```

На выходе мы получаем по одной записи на каждый год. Построенный по результатам график показан на рис. 4.4. Легко видеть, что увеличение количества патентов, выданных с 1960 по 1990 год (см. рис 4.3) не связано с увеличением количества стран-участниц (рис. 4.4). Количество стран почти не изменилось, но они стали подавать больше заявок.

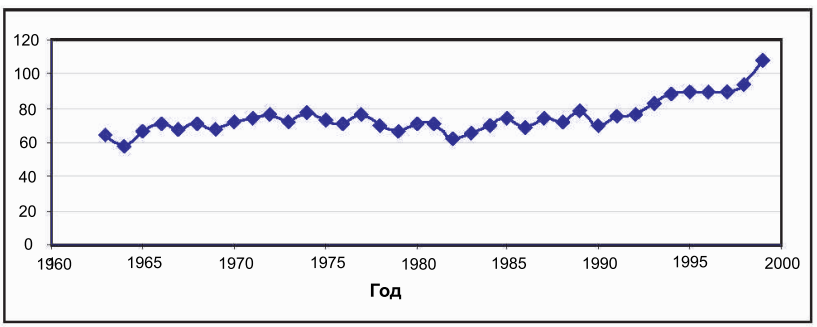


Рис. 4.4. Количество стран, в которых выдавались патенты США в разные годы. Вычисления выполнены с помощью Hadoop, а график построен в Excel.

Листинг 4.10.
UniqueCount.py: обертка вокруг функции UniqValueCount

```
#!/usr/bin/env python

import sys

index1 = int(sys.argv[1])
index2 = int(sys.argv[2])
for line in sys.stdin:
    fields = line.split(",")
    print "UniqValueCount:" + fields[index1] + "\t" + fields[index2]
```

Агрегатная функция `ValueHistogram` – самая развитая из имеющихся в пакете Aggregate. Для каждого ключа она выводит следующие данные:

1. Количество уникальных значений.
2. Минимальное значение.
3. Медиану.
4. Максимальное значение.
5. Среднее значение.
6. Стандартное отклонение.

В самом общем виде эта функция ожидает, что распределитель будет выводить строки в формате

```
ValueHistogram:key\tvalue\tcount
```

Мы указываем имя функции `ValueHistogram`, затем двоеточие и тройку, состоящую из ключа, значения и счетчика, разделенных знаками табуляции. Редуктор Aggregate выводит шесть статистических показателей для каждого ключа. Отметим, что для всех показателей, кроме первого (количество уникальных значений), производится суммирование счетчиков по всем парам ключ/значение. Формирование распределителем таких двух записей:

```
ValueHistogram:key_a\tvalue_a\t10
ValueHistogram:key_a\tvalue_a\t20
```

в точности эквивалентно выводу одной записи, содержащей сумму:

```
ValueHistogram:key_a\tvalue_a\t30
```

Существует полезный частный случай, когда распределитель выводит только ключ и значение, без счетчика и предшествующего ему знака табуляции. Тогда `ValueHistogram` автоматически предполагает, что счетчик равен 1. В листинге 4.11 приведена тривиальная обертка вокруг функции `ValueHistogram`.

Листинг 4.11.
ValueHistogram.py: обертка вокруг функции ValueHistogram из пакета Aggregate

```
#!/usr/bin/env python

import sys

index1 = int(sys.argv[1])
index2 = int(sys.argv[2])
for line in sys.stdin:
    fields = line.split(",")
    print "ValueHistogram:" + fields[index1] + "\t" + fields[index2]
```

Запустим эту программу, чтобы найти распределение стран, в которых выдавались патенты, по годам.

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar
       ➥ -input input/apat63_99.txt
       ➥ -output output
       ➥ -file ValueHist.py
       ➥ -mapper 'ValueHist.py 1 4'
       ➥ -reducer aggregate
```

На выходе получается файл, каждая строка которого состоит из семи столбцов, разделенных знаками табуляции. Первый столбец, год выдачи патента, – это ключ. Остальные шесть столбцов – статистики, вычисляемые функцией `ValueHistogram`. Ниже приведен фрагмент результирующего файла (первые две строки мы опустили):

```
1964    58    1    7    38410    816.8103448275862    4997.413601595352
1965    67    1    5    50331    938.1641791044776    6104.779230296307
1966    71    1    5    54634    963.4507042253521    6443.625995189338
1967    68    1    8    51274    965.4705882352941    6177.445623039149
1968    71    1    7    45781    832.4507042253521    5401.229955880634
1969    68    1    8    50394    993.5147058823529    6080.713518728092
1970    72    1    7    47073    894.8472222222222    5527.883233761672
1971    74    1    9    55976    1058.337837837838    6492.837390992137
```

Первый после года столбец содержит количество уникальных значений. Оно в точности совпадает с тем, что выводит функция `UniqValueCount`. Второй, третий и четвертый столбцы – соответственно минимум, медиана и максимум. Видно, что в каждом году наименьшее (отличное от 0) количество патентов, выданных в одной стране, равно 1. В 1964 году была страна, получившая 38410 патентов, тогда как в половине стран было выдано менее 7 патентов. Среднее количество патентов, выданных в одной стране в 1964 году, составило 816,8 со стандартным отклонением 4997,4. Следовательно, распреде-

ление патентов по странам было очень неравномерным, что и неудивительно, если принять во внимание разницу между медианой (7) и средним (816,8).

Мы показали, что при использовании пакета Aggregate совместно с интерфейсом Streaming можно без труда получить некоторые широко распространенные метрики. Это демонстрация того, как мощь Hadoop позволяет упростить анализ больших наборов данных.

4.6. Повышение производительности с помощью комбинаторов

В скриптах AverageByAttributeMapper.py и AverageByAttributeReducer.py (листинги 4.7 и 4.8) мы видели, как вычисляется среднее для каждого атрибута. Распределитель считывает записи и выводит пары ключ/значение, содержащие имя атрибута и счетчик. Затем эти пары перетасовываются, и редуктор вычисляет среднее для каждого ключа. В нашем примере вычисления среднего числа пунктов в формуле изобретения для патентов, выданных в каждой отдельной стране, наблюдается по меньшей мере два узких места:

1. При наличии 1 миллиарда входных записей распределители породят 1 миллиард пар ключ/значение, которые нужно будет перетасовать и разослать по сети. Если вычисляется, например, функция максимума, то ясно, что распределителю следует выводить только максимум для каждого встретившегося ему ключа. Это уменьшило бы сетевой трафик и повысило производительность. Для функции среднего ситуация несколько сложнее, но все равно алгоритм можно переформулировать так, что каждый распределитель будет выводить для каждого ключа только одну запись, принимающую участие в тасовании.
2. Использование страны в качестве ключа иллюстрирует проблему неравномерности распределения данных. В этом случае неравномерность обусловлена тем, что в подавляющем большинстве записей в качестве ключа фигурирует США. Мало того что каждая пара ключ/значение во входных данных порождает пару в промежуточных данных, так еще и промежуточные пары попадают одному редуктору, перегружая его.

Hadoop решает обе проблемы, дополняя каркас MapReduce шагом *комбинирования*, расположенным между распределителем и редуктором. Можно считать, что комбинатор помогает редуктору. Его назначение – сократить объем данных, порождаемых распределителем, и тем самым уменьшить нагрузку на сеть и на редуктор. Если указан комбинатор, то каркас MapReduce может применить его к промежуточным данным нуль, один или более раз. Чтобы от комбинатора была польза, он должен осуществлять эквивалентное относительно редуктора преобразование данных. Если убрать комбинатор, то выход редуктора не должен измениться. Более того, свойство эквивалентности преобразования должно сохраняться и в случае, когда комбинатор применяется к произвольному подмножеству промежуточных данных.

Если редуктор реализует дистрибутивную функцию, например максимум, минимум или суммирование (подсчет), то в качестве комбинатора можно использовать сам редуктор. Однако многие полезные функции не являются дистрибутивными. Но некоторые из них, к примеру усреднение, можно переписать с использованием комбинаторов.

Скрипт `AverageByAttributeMapper.py`, являющий распределителем для вычисления среднего, выводит все пары ключ/значение. Затем редуктор `AverageByAttributeReducer.py` подсчитывает количество полученных им пар, вычисляет сумму значений и в конце делит одно на другое, получая среднее. Основное препятствие на пути использования комбинатора – операция подсчета, потому что редуктор предполагает, что полученное им число пар ключ/значение равно числу пар во входных данных. Однако мы можем переработать MapReduce-программу, так чтобы она подсчитывала счетчик явно. Тогда комбинатор становится простой функцией суммирования, обладающей свойством дистрибутивности.

Перед тем как писать комбинатор, переделаем распределитель и редуктор, поскольку задача MapReduce должна корректно работать даже без комбинатора. Новую программу вычисления среднего мы напишем на Java, так как комбинатор должен быть классом Java.

Примечание. Streaming API позволяет задать комбинатор с помощью флага `-combiner`. В версиях вплоть до 0.20 комбинатор должен быть классом Java. Поэтому лучше всего писать распределитель и редуктор тоже на Java. К счастью, в плане развития Hadoop предусмотрена поддержка комбинаторов, реализованных в виде Streaming-скриптов. На практике получить эквивалент комбинатора можно, взяв в качестве распределителя конвейер Unix 'mapper.py | sort | combiner.py'. Кроме того, если вы используете пакет Aggregate, то

в нем для каждого агрегатора значений уже имеется встроенный комбинатор (написанный на Java). Причем Aggregate применяет эти комбинаторы автоматически.

Напишем на Java распределитель (листинг 4.12), эквивалентный скрипту `AverageByAttributeMapper.py` из листинга 4.7.

Листинг 4.12.
Эквивалент AverageByAttributeMapper.py на языке Java

```
public static class MapClass extends MapReduceBase
   implements Mapper<LongWritable, Text, Text, Text> {

   public void map(LongWritable key, Text value,
                   OutputCollector<Text, Text> output,
                   Reporter reporter) throws IOException {

      String fields[] = value.toString().split(",", -20);
      String country = fields[4];
      String numClaims = fields[8];
      if (numClaims.length() > 0 && !numClaims.startsWith("\""))
{
         output.collect(new Text(country),
                        new Text(numClaims + ",1")); q
      }
   }
}
```

1 **Включить в выходную информацию счетчик 1**

Важнейшее отличие нового Java-распределителя в том, что в выходную информацию теперь добавляется счетчик, равный 1 (1). Можно было бы определить новый тип данных `Writable` для хранения значения и счетчика вместе, но здесь ситуация настолько проста, что вполне достаточно представлять то и другое в виде текстовой строки с запятой в качестве разделителя.

На стороне редуктора разбирается список значений для каждого ключа. Общая сумма и количество вычисляются далее путем суммирования, и в конце для получения среднего одно делится на другое.

```
public static class Reduce extends MapReduceBase
   implements Reducer<Text, Text, Text, DoubleWritable> {

   public void reduce(Text key, Iterator<Text> values,
                      OutputCollector<Text, DoubleWritable> output,
                       Reporter reporter) throws IOException {

      double sum = 0;
      int count = 0;
      while (values.hasNext()) {
```

```
            String fields[] = values.next().toString().split(",");
            sum += Double.parseDouble(fields[0]);
            count += Integer.parseInt(fields[1]);
        }
        output.collect(key, new DoubleWritable(sum/count));
    }
}
```

Логика переработанной задачи MapReduce легко прослеживается, не так ли? Мы добавили к каждой паре ключ/значение явный счетчик и в результате получили возможность комбинировать промежуточные данные в узле каждого распределителя перед тем, как передавать их по сети.

Программно комбинатор должен реализовывать интерфейс `Reducer`. Операцию комбинирования выполняет метод `reduce()` комбинатора. На первый взгляд, такая схема именования представляется неудачной, но напомним, что для важного класса дистрибутивных функций комбинатор и редуктор выполняют одни и те же действия. Поэтому комбинатор принял на вооружение сигнатуру редуктора, дабы упростить повторное использование. Вам не придется переименовывать свой класс редуктора, чтобы использовать его как комбинатор. Кроме того, поскольку комбинатор производит эквивалентное преобразование, типы ключа и значения на его выходе должны совпадать с соответствующими типами на входе. В итоге мы разработали класс `Combine`, похожий на класс `Reduce`, но отличающийся от него тем, что его работа завершается выводом (частичной) суммы и счетчика, тогда как редуктор вычисляет окончательное среднее.

```
public static class Combine extends MapReduceBase
    implements Reducer<Text, Text, Text, Text> {

    public void reduce(Text key, Iterator<Text> values,
                       OutputCollector<Text, Text> output,
                       Reporter reporter) throws IOException {

        double sum = 0;
        int count = 0;
        while (values.hasNext()) {
            String fields[] = values.next().toString().split(",");
            sum += Double.parseDouble(fields[0]);
            count += Integer.parseInt(fields[1]);
        }
        output.collect(key, new Text(sum + "," + count));
    }
}
```

Чтобы подключить этот комбинатор, драйвер должен указать его класс в объекте `JobConf`. Для этого служит метод `setCombinerClass()`. Драйвер задает распределитель, редуктор и комбинатор:

```
job.setMapperClass(MapClass.class);
job.setCombinerClass(Combine.class);
job.setReducerClass(Reduce.class);
```

Комбинатор необязательно способствует повышению производительности. Вы должны понаблюдать за поведением задачи и убедиться, что количество записей, выведенных комбинатором, заметно меньше, чем было получено им на входе. Выигрыш на этапе редукции должен оправдывать дополнительное время, затраченное на работу комбинатора. Проверить все это легко с помощью веб-интерфейса демона JobTracker, который мы будем рассматривать в главе 6.

На рис. 4.5 видно, что на этапе распределения комбинатор получает на входе 1 984 625 записей, а выводит только 1063 записи. Очевидно, что комбинатор уменьшил объем промежуточных данных. Отметим, что комбинатор исполняется и на стороне редуктора, хотя в данном случае выгода пренебрежимо мала.

	Счетчик	Map	Reduce	Total
Job Counters	Data-local map tasks	0	0	4
	Launched reduce tasks	0	0	2
	Launched map tasks	0	0	4
Map-Reduce Framework	Reduce input records	0	151	151
	Map output records	1 984 055	0	1 984 055
	Map output bytes	18 862 764	0	18 862 764
	Combine output records	1063	151	1214
	Map input records	2 923 923	0	2 923 923
	Reduce input groups	0	151	151
	Combine input records	1 984 625	493	1 985 118
	Map input bytes	236 903 179	0	236 903 179
	Reduce output records	0	151	151
File Systems	HDFS bytes written	0	2658	2658
	Local bytes written	20 554	2510	23 064
	HDFS bytes read	236 915 470	0	236 915 470
	Local bytes read	21 112	2510	23 622

Рис. 4.5. Мониторинг эффективности комбинатора в задаче AveragingWithCombiner

4.7. Упражнения

Практика – путь к мастерству. Попробуйте выполнить следующие упражнения, чтобы проверить, как вы научились мыслить в парадигме MapReduce.

1. *Первые K-записей.* Измените скрипт `AttributeMax.py` (или `AttributeMax.php`), так чтобы он выводил запись целиком, а не только максимальное значение. Перепишите его так, чтобы задача MapReduce выводила записи, содержащие *K* наибольших значений, а не только максимум.
2. *Измерение веб-трафика.* Возьмите какой-нибудь файл журнала веб-сервера и напишите Streaming-программу, пользующуюся пакетом Aggregate, которая вычисляет трафик этого сайта с разбивкой по часам.
3. *Скалярное произведение двух разреженных векторов.* Вектор – это список значений. Если даны два вектора X = [x1, x2, ...] и Y = [y1, y2, ...], то их скалярным произведением называется выражение Z = x1 * y1 + x2 * y2 + ... Длинные векторы X и Y, содержащие много нулевых элементов, называются разреженными; обычно они задаются в таком виде:

   1,0.46
   9,0.21
   17,0.92
   ...

   где ключ (первый столбец) – это индекс элемента вектора. Предполагается, что элементы, которые не заданы явно, равны нулю. Отметим, что ключи необязательно следуют в порядке возрастания. Более того, ключи могут быть даже нечисловыми. (В случае обработке текстов на естественном языке ключами могут быть встречающиеся в документе слова, тогда скалярное произведение – это мера схожести документов.) Напишите Streaming-программу для вычисления скалярного произведения двух разреженных векторов. Можете добавить шаг постобработки, выполняемый после задачи MapReduce для завершения вычисления.
4. *Обработка временных рядов.* Рассмотрим временной ряд, в котором каждая запись снабжена временной меткой, интерпретируемой как ключ, и содержит результат измерения – значение. Мы хотим получить на выходе результат применения к

временному ряду линейной функции вида:

$$y(t) = a0 * x(t) + a1 * x(t-1) + a2 * x(t-2) + ... + aN * x(t-N),$$

где t – обозначает время, а a0, ..., aN – известные константы. В теории обработки сигналов это называется фильтром с конечной импульсной характеристикой (КИХ-фильтр). Особенный интерес представляет частный случай *скользящего среднего*, когда $a0 = a1 = ... = aN = 1/N$. Каждая точка y – это среднее N предыдущих точек x. Это простой способ сглаживания временных рядов.

Реализуйте этот линейный фильтр в виде MapReduce-программы. Используйте комбинатор. Если временные ряды хронологически упорядочены (а так обычно и бывает) и N относительно невелико, то какого сокращения сетевого трафика при тасовании можно добиться за счет использования комбинатора? В качестве дополнительного упражнения напишите свой разбиватель, гарантирующий, что выходные данные также будут отсортированы в хронологическом порядке.

Опытные программисты заметят в этом примере иллюстрацию различия между масштабируемостью и производительностью. Реализация КИХ-фильтра в Hadoop позволяет добиться масштабируемости, то есть возможности обрабатывать терабайты данных и даже больше. Студенты, специализирующиеся в области обработки сигналов, знают, что *высокопроизводительные* реализации КИХ-фильтра часто основаны на технике быстрого преобразования Фурье (БПФ). Решение, одновременно масштабируемое *и* высокопроизводительное должно было бы основываться на MapReduce-реализации БПФ, но эта тема уже выходит за рамки данной книги.

5 *Свойство коммутативности*. Из курса элементарной математики известно, что под коммутативностью понимается независимость от порядка выполнения операций. Например, сложение коммутативно, то есть a + b = b + a и a + b + c = b + a + c = b + c + a = c + a + b = c + b + a. Верно ли, что каркас MapReduce принципиально предназначен для реализации коммутативных функций? Объясните свой ответ.

6 *Умножение (вычисление произведения)*. Во многих алгоритмах машинного обучения и статистической классификации требуется многократно перемножать значения вероятности.

Обычно мы сравниваем произведения двух наборов вероятностей и принимаем решение о классификации в зависимости от того, какое произведение больше. Выше мы видели, что максимум – дистрибутивная функция. Верно ли, что произведение также обладает свойством дистрибутивности? Напишите MapReduce-программу для перемножения всех значений в наборе данных. Верно ли что, реализация такой программы с помощью MapReduce решает все вопросы масштабируемости? Что нужно сделать, чтобы исправить положение дел?

(Часто отвечают, что нужно написать собственную библиотеку для выполнения арифметических действий с плавающей точкой, но это плохой ответ.)

7 *Трансляция на выдуманный язык*. На первых занятиях по курсу информатики часто предлагают написать программу для перевода с английского на «пиратский». Есть и другие варианты выдуманных языков, например, «язык Snoop Dogg»[10] и «E- 40»[11]. Обычно решение основано на поиске в словаре точных соответствий («for» преобразуется в «fo», «sure» – в «sho», «the» – в «da» и т. д), применении простых правил текстовой подстановки (слова, оканчивающиеся на «ing», получают окончание «in'», последняя гласная в слове и все буквы после нее заменяются на «izzle» и т. д.) и случайных вставках («kno' wha' im sayin'?»). Напишите такую программу и с помощью Hadoop примените ее к большому корпусу текстов, например, к википедии.

4.8. Резюме

MapReduce-программы строятся по определенному шаблону. Часто вся программа определяется внутри одного класса Java. В этом классе есть драйвер, который настраивает конфигурационный объект задачи MapReduce, описывающий, как следует ее запускать и исполнять. Функции map и reduce представляются с помощью классов, реализующих интерфейсы `Mapper` и `Reducer` соответственно. Они часто занимают всего пару десятков строчек, поэтому для удобства обычно кодируются в виде внутренних классов.

[10] Настоящее имя Cordozar Calvin Broadus. Афроамериканскйи рэпер, продюсер и актер. *Прим. перев.*

[11] Американский рэпер, Чарли Хасл, внесший большой вклад в культуру хип-хопа. В частности, придумал массу жаргонных словечек, которые до сих пор в ходу у рэперов. *Прим. перев.*

Hadoop предоставляет интерфейс Streaming API для создания MapReduce-программ на языках, отличных от Java. Многие MapReduce-программы гораздо проще писать на скриптовых языках с помощью Streaming API, особенно программы для ситуативного анализа данных. Существует пакет Aggregate, который при использовании совместно с интерфейсом Streaming позволяет быстро разрабатывать программы для подсчета и вычисления простых статистических функций.

Каркас MapReduce определяет прежде всего функции распределения (map) и редукции (reduce), но Hadoop позволяет также задать функцию комбинирования для повышения производительности за счет введения шага «предредукции» промежуточных данных на стороне распределителя.

В стандартных программах (без применения парадигмы MapReduce) подсчет, суммирование, вычисление среднего и другие подобные операции обычно производятся с помощью одного простого прохода по данным. Переработка таких программ под MapReduce, то есть то, чем мы занимались в этой главе, концептуально сравнительно проста. Но для более сложных алгоритмов анализа данных требуется более глубокая модификация. Это и станет темой следующей главы.

4.9. Дополнительные ресурсы

В этой главе мы использовали набор данных о патентах, но существуют и другие находящиеся в открытом доступе большие наборы данных, которые можно загрузить для экспериментирования. Ниже приведено несколько примеров.

http://www.netflixprize.com/index. Netflix – это сайт для аренды кинофильмов. Важнейшей его особенностью является подсистема рекомендования, которая предлагает пользователю новые фильмы, опираясь на проставленные пользователями оценки предыдущих фильмов. Компания опубликовала набор данных, содержащий пользовательские оценки, предлагая всем желающим разработать более качественный алгоритм рекомендования. Без сжатия размер набора составляет более 2 ГБ. Он содержит свыше 100 миллионов оценок 17 тысяч фильмов, проставленных 480 тысячами пользователей.

http://aws.amazon.com/publicdatasets/. На серверах компании Amazon размещено несколько бесплатных больших наборов данных для пользователей предлагаемой ей службы EC2.

В настоящее время есть наборы данных, относящиеся к трем категориям: биология, химия и экономика. Например, один из биологических наборов – это аннотированные данные о геноме человека объемом примерно 550 ГБ. В разделе экономики имеются данные о переписи США 2000 года (порядка 200 ГБ).

http://boston.lti.cs.cmu.edu/Data/clueweb09/. Институт языковых технологий при университете Карнеги-Меллон опубликовал набор данных ClueWeb09 для поддержки исследований в области крупномасштабных веб-проектов. Это миллиард собранных роботом веб-страниц на 10 языках. В несжатом виде набор занимает 25 ТБ. С учетом размера для получения этого набора лучше всего заказать доставку на жестких дисках в сжатом виде (около 5 ТБ). (При определенном масштабе доставка сжатых дисков почтовой службой FedEx характеризуется высокой «полосой пропускания»). В настоящее время университет запрашивает 790 долларов США за отгрузку четырех дисков емкостью 1,5 ТБ, содержащих сжатые данные.

ГЛАВА 5.
Углубленное изучение MapReduce

В этой главе:

- Сцепление нескольких задач MapReduce.
- Соединение нескольких наборов данных.
- Создание фильтров Блума.

По мере усложнения обработки данных возникает потребность в использовании дополнительных возможностей Hadoop. В этой главе мы рассмотрим некоторые из более сложных технических приемов.

Часто в процессе нетривиальной обработки данных обнаруживается, что программу нельзя представить в виде одной задачи MapReduce. Hadoop поддерживает *сцепление* нескольких MapReduce-программ в одну более крупную задачу. Далее, нередко выясняется, что необходимо обрабатывать несколько наборов данных. Мы изучим различные имеющиеся в Hadoop методы *соединения*, применяемые для этой цели сразу нескольких наборов данных. Некоторые операции обработки данных можно запрограммировать более эффективно, если обрабатывать сразу группы записей. Мы видели, что интерфейс Streaming поддерживает обработку целой порции, и в реализации функции вычисления максимума с использованием Streaming эта возможность нашла применение. Мы покажем, что то же самое справедливо и для программ на Java. Мы объясним, что такое фильтр Блума, и реализуем его с помощью распределителя, сохраняющего информацию о состоянии при переходе от одной записи к другой.

5.1. Сцепление задач MapReduce

До сих пор мы сталкивались с такими операциями обработки данных, для которых было достаточно одной задачи MapReduce. Но накапливая опыт работы с MapReduce, вы обнаружите, что многие сложные операции необходимо разбивать на более простые подоперации, каждая из которых реализуется отдельной задачей. Например, пусть требуется найти десять самых цитируемых патентов. Это можно сделать с помощью двух задач MapReduce. Первая создает «инвертированный» набор данных о цитировании и подсчитывает количество ссылок на каждый патент, а вторая отбирает первые десять записей «инвертированных» данных.

5.1.1. Последовательное сцепление задач MapReduce

Две задачи можно запустить одну за другой вручную, но удобнее автоматизировать последовательность выполнения. Hadoop позволяет сцеплять последовательного выполняемые задачи MapReduce, так что выход одной становится входом другой. Сцепление задач MapReduce аналогично конвейерам Unix.

```
mapreduce-1 | mapreduce-2 | mapreduce-3 | ...
```

Сцепление задач MapReduce реализуется очень просто. Напомним, что драйвер настраивает объект `JobConf`, задавая в нем конфигурационные параметры задачи, и передает этот объект методу `JobClient.runJob()` для запуска задачи. Поскольку `JobClient.runJob()` блокирует программу до завершения задачи, то сцепление сводится к последовательному вызову драйверов нескольких задач. Драйвер каждой задачи должен будет создать новый объект `JobConf`, задав в качестве входного пути выходной путь предыдущей задачи. В конце можно удалить промежуточные данные, сгенерированные на каждом шаге цепочки.

5.1.2. Сцепление задач MapReduce со сложными зависимостями

Иногда подоперации составной операции обработки данных должны запускаться не последовательно, поэтому соответствующие задачи MapReduce сцепляются нелинейно. Например, mapreduce1 может

обрабатывать один набор данных, а mapreduce2 одновременно и независимо обрабатывать другой набор данных. Третья задача, mapreduce3, выполняет внутреннее соединение результатов первых двух (соединение данных мы будем обсуждать в следующем разделе). Эта операция зависит от двух предыдущих и потому может начаться только после того, как mapreduce1 и mapreduce2 завершатся. Но друг от друга mapreduce1 и mapreduce2 не зависят.

В Hadoop имеется механизм, позволяющий упростить управление такими нелинейными зависимостями между задачами. В нем используются классы `Job` и `JobControl`. Объект `Job` – это представление задачи MapReduce. Конструктору объекта `Job` передается объект `JobConf`. Помимо конфигурационных параметров задачи, объект `Job` хранит еще информацию о зависимостях, задаваемую с помощью метода `addDependingJob()`. Для объектов `x` и `y` типа `Job` вызов

```
x.addDependingJob(y)
```

означает, что `x` не должен запускаться, пока `y` не завершится. Если в объектах `Job` хранится информация о конфигурации и зависимостях, то объекты `JobControl` служат для управления и мониторинга исполнения задач. Метод `addJob()` позволяет добавить задачу в объект `JobControl`. После того как все задачи и зависимости определены, мы вызываем метод `run()` объекта `JobControl`, который запускает поток, подающий задачи на исполнение и следящий за ходом исполнения. В классе `JobControl` есть, например, методы `allFinished()` и `getFailedJobs()`, позволяющие отслеживать исполнение различных задач, входящих в состав пакета.

5.1.3. Включение в цепочку шагов пред- и постобработки

Во многих операциях обработки данных встречается пред- и постобработка записей. Например, при обработке документов в информационно-поисковой системе одним из шагов может быть удаление *стоп-слов* (слов типа *a*, *the* или *is*, которые встречаются часто, но не несут особого смысла), а другим – *выделение основы* (преобразование различных словоформ, например *finishing* и *finished* к основе *finish*). Для каждого такого шага пред- или постобработки можно написать отдельные задачи MapReduce и связать их в цепочку, используя в качестве редуктора `IdentityReducer` (или вообще обойдясь без редуктора). Но такой подход неэффективен, поскольку каждая операция в цепоч-

ке потребляет ресурсы ввода/вывода и место в системе хранения для обработки промежуточных результатов. Альтернативное решение – написать собственный распределитель, который в начале вызовет все шаги предобработки, и собственный редуктор, который вызовет все шаги постобработки в самом конце. Правда, при этом необходимо проектировать шаги пред- и постобработки так, чтобы они были модульными и допускали композицию. В версии Hadoop 0.19.0 были добавлены классы `ChainMapper` и `ChainReducer`, упрощающие композицию шагов пред- и постобработки.

Способ сцепления задач MapReduce, описанный в разделе 5.1.1, можно символически представить псевдорегулярным выражением:

```
[MAP | REDUCE]+
```

которое означает, что редуктор `REDUCE` исполняется после распределителя `MAP`, и эта последовательность [MAP | REDUCE] может повторяться один или более раз. Аналогичное выражение для задачи, в которой используются классы `ChainMapper` и `ChainReducer`, имеет вид:

```
MAP+ | REDUCE | MAP*
```

Эта задача последовательно запускает несколько распределителей для предобработки данных, и, возможно, еще несколько распределителей после фазы редукции для постобработки. Изящество механизма в том, что шаги пред- и постобработки кодируются как стандартные распределители. Каждый из них можно при желании запустить отдельно (это полезно, например, для отладки). Для композиции шагов пред- и постобработки следует вызвать метод `addMapper()` классах `ChainMapper` или `ChainReducer` соответственно. При запуске всех шагов пред- и постобработки в контексте одной задачи не остается никаких промежуточных файлов и наблюдается значительно меньшее потребление ресурсов ввода/вывода.

Рассмотрим пример, в котором имеется четыре распределителя (`Map1`, `Map2`, `Map3` и `Map4`) и один редуктор (`Reduce`), сцепленные в одну задачу MapReduce в такой последовательности:

```
Map1 | Map2 | Reduce | Map3 | Map4
```

На этом шаге `Map2` и `Reduce` следует рассматривать как основную часть задачи MapReduce, в которой между распределителем и редуктором выполняется стандартное разбиение и тасование. `Map1` при этом становится шагом предобработки, а `Map3` и `Map4` – шагами пос-

тобработки. Количество шагов обработки может варьироваться. Но это только один из возможных примеров.

Композицию распределителей и редуктора можно описать в драйвере (см. листинг 5.1). Необходимо гарантировать, что ключи и значения на выходе предыдущей задачи имеют те же типы (классы), что на входе следующей.

Листинг 5.1.
Драйвер для сцепления распределителей в одной задаче MapReduce

```
Configuration conf = getConf();
JobConf job = new JobConf(conf);

job.setJobName("ChainJob");
job.setInputFormat(TextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);

JobConf map1Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map1.class,
                      LongWritable.class,
                      Text.class,
                      Text.class,
                      Text.class,
                      true,
                      map1Conf);

JobConf map2Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map2.class,
                      Text.class,
                      Text.class,
                      LongWritable.class,
                      Text.class,
                      true,
                      map2Conf);

JobConf reduceConf = new JobConf(false);
ChainReducer.setReducer(job,
                        Reduce.class,
                        LongWritable.class,
                        Text.class,
                        Text.class,
                        Text.class,
                        true,
                        reduceConf);

JobConf map3Conf = new JobConf(false);
```

Добавить в задачу шаг Map1

Добавить в задачу шаг Map2

Добавить в задачу шаг Reduce

```
ChainReducer.addMapper(job,
                       Map3.class,
                       Text.class,
                       Text.class,
                       LongWritable.class,
                       Text.class,
                       true,
                       map3Conf);

JobConf map4Conf = new JobConf(false);
ChainReducer.addMapper(job,
                       Map4.class,
                       LongWritable.class,
                       Text.class,
                       LongWritable.class,
                       Text.class,
                       true,
                       map4Conf);

JobClient.runJob(job);
```

Добавить в задачу шаг Map3

Добавить в задачу шаг Map4

Драйвер сначала настраивает «глобальный» объект `JobConf`, задавая в нем имя задачи, входной путь, выходной путь и т. д. Затем он последовательно добавляет пять шагов сцепленной задачи в порядке их исполнения. Шаги, предшествующие `Reduce`, добавляются с помощью статического метода `ChainMapper.addMapper()`. Редуктор добавляется статическим методом `ChainReducer.setReducer()`. Для добавления последних шагов вызывается метод `ChainReducer.addMapper()`. Глобальный объект `JobConf (job)` передается при вызове каждого из пяти методов `add*`. Кроме того, у каждого распределителя и редуктора имеется локальный объект `JobConf (map1Conf, map2Conf, map3Conf, map4Conf` и `reduceConf)`, который имеет приоритет по сравнению с глобальным при конфигурировании отдельных распределителей и редукторов. Локальный объект `JobConf` рекомендуется инициировать так, чтобы умолчания не использовались – `new JobConf(false)`.

Рассмотрим сигнатуру метода `ChainMapper.addMapper()`, чтобы лучше понять, как добавляются шаги в сцепленную задачу. Сигнатуры методов `ChainReducer.setReducer()` и `ChainReducer.addMapper()` аналогичны, поэтому мы их опустим.

```
public static <K1,V1,K2,V2> void
              addMapper(JobConf job,
                        Class<? extends Mapper<K1,V1,K2,V2>> klass,
                        Class<? extends K1> inputKeyClass,
                        Class<? extends V1> inputValueClass,
                        Class<? extends K2> outputKeyClass,
                        Class<? extends V2> outputValueClass,
```

```
boolean byValue,
JobConf mapperConf)
```

У этого метода восемь аргументов. Первый и последний – глобальный и локальный объекты `JobConf` соответственно. Второй аргумент (`klass`) – производный от `Mapper` класс, отвечающий за обработку данных. Четыре аргумента `inputValueClass`, `inputKeyClass`, `outputKeyClass` и `outputValueClass` – типы классов входных и выходных ключей и значений для класса `Mapper`.

Аргумент `byValue` нуждается в пояснениях. В стандартной модели `Mapper` пары ключ/значение сериализуются и записываются на диск[1] для последующей передачи в ходе тасования редуктору, который может находиться на другом узле. Формально это расценивается как *передача по значению*, потому что пересылается копия пары ключ/значение. В данном случае, когда один `Mapper` сцепляется с другим, они могут исполняться в одном и том же потоке JVM. Поэтому открывается возможность передавать пары ключ/значение *по ссылке*, тогда выход предыдущего `Mapper`'а остается в памяти, а последующий `Mapper` ссылается него по адресу. Когда `Map1` вызывает метод `OutputCollector.collect(K k, V v)`, объекты `k` и `v` передаются напрямую методу `map()` объекта `Map2`. В результате производительность увеличивается, так как отпадает необходимость клонировать потенциально большой объем данных для передачи от одного распределителя другому. Однако при этом может быть нарушен один из тонких «контрактов» MapReduce API в Hadoop. Гарантируется, что вызов `OutputCollector.collect(K k, V v)` не изменяет содержимое `k` и `v`. Объект `Map1` может вызвать `OutputCollector.collect(K k, V v)`, а затем использовать объекты `k` и `v`, ожидая, что их значения не изменились. Но если эти объекты передаются `Map2` по ссылке, то никто не помешает `Map2` изменить их и тем самым нарушить гарантию API. Если вы уверены, что метод `map()` объекта `Map1` не использует содержимое `k` и `v` после обращения к `OutputCollector.collect(K k, V v)` или что `Map2` не изменяет переданные ему значения `k` и `v`, то можете добиться некоторого повышения производительности, присвоив параметру `byValue` значение `false`. Если же вы точно не знаете, как устроен код `Mapper`'а, то лучше не рисковать и оставить `byValue` равным `true`, сохранив верность модели передачи по значению. Тогда можно быть уверенным, что объекты `Mapper` будут работать в соответствии с ожиданиями.

[1] Возможность клонирования и сериализации ключей и значений обеспечивается тем, что они реализуют интерфейс `Writable`.

5.2. Соединение данных из разных источников

При решении задач анализа данных рано или поздно настает момент, когда необходимо брать данные из разных источников. Так, в примере набора данных о патентах может возникнуть вопрос, есть ли в патентах, выданных в одной стране, ссылки на патенты, выданные в другой стране. Тогда нужно принять в рассмотрение как данные о цитировании (файл `cite75_99.txt`), так и данные о странах (`apat63_99.txt`). В мире баз данных для этого было бы достаточно соединить две таблицы, и большинство СУБД произведут соединение автоматически. К сожалению, в Hadoop операция соединения данных более сложна. Существует несколько подходов, каждому из которых свойственны определенные компромиссы.

Для иллюстрации проблемы рассмотрим два «игрушечных» набора данных. В качестве одного файла возьмем представленный в формате CSV файл заказчиков Customers, в котором каждая запись состоит из трех полей: идентификатор заказчика (Customer ID), имя (Name) и телефон (Phone Number). Пусть в нем будет всего четыре записи:

```
1,Stephanie Leung,555-555-5555
2,Edward Kim,123-456-7890
3,Jose Madriz,281-330-8004
4,David Stork,408-555-0000
```

В другом файле, Orders, будем хранить информацию о заказах. Он тоже представлен в формате CSV, а запись состоит из четырех полей: идентификатор заказчика (Customer ID), идентификатор заказа (Order ID), цена (Price) и дата покупки (Purchase Date).

```
3,A,12.95,02-Jun-2008
1,B,88.25,20-May-2008
2,C,32.00,30-Nov-2007
3,D,25.02,22-Jan-2009
```

Результат соединения должен выглядеть, как показано в листинге 5.2.

Листинг 5.2.
Желаемый результат внутреннего соединения файлов Customers и Orders

```
1,Stephanie Leung,555-555-5555,B,88.25,20-May-2008
2,Edward Kim,123-456-7890,C,32.00,30-Nov-2007
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009
```

Hadoop умеет выполнять и другие виды соединений, но для простоты мы пока ограничимся внутренним.

5.2.1. Соединение на стороне редуктора

В состав Hadoop входит дополнительный пакет *datajoin*, играющий роль обобщенного механизма для соединения данных. Он находится в jar-файле contrib/datajoin/hadoop-*-datajoin.jar. Чтобы отличить этот механизм от других способов соединения, будем называть его *соединением на стороне редуктора*, поскольку большая часть обработки производится редуктором. Его также называют *соединением с переразбиением* (repartitioned join) или *соединением с переразбиением методом сортировки слиянием*, поскольку так называется аналогичная техника в базах данных. Не будучи самой эффективной, эта техника является наиболее общей и лежит в основе некоторых более сложных методов (например, полусоединения).

Техника соединения на стороне редуктора несет с собой новую терминологию и понятия: источник данных, тег и групповой ключ. *Источник данных* – это аналог таблицы в реляционных базах данных. В нашем примере источников данных два: Customers и Orders. Источником данных может быть как один, так и несколько файлов. Важно, что все записи в одном источнике данных имеют одинаковую структуру, которую можно уподобить схеме таблицы.

Парадигма MapReduce предполагает обработку записей по одной, без сохранения состояния. Если необходимо все-таки сохранять какую-то информацию о состоянии, необходимо *пометить* запись тегом, содержащим эту информацию. Так, в нашем примере, распределитель может видеть запись

```
3,Jose Madriz,281-330-8004
```

или

```
3,A,12.95,02-Jun-2008
```

где тип записи (Customers или Orders) отделен от самой записи. Пометка записи тегом гарантирует, что определенные метаданные всегда будут сопровождать эту запись. Для соединения данных нам нужно будет пометить каждую запись ее *источником данных*.

Групповой ключ работает по аналогии с ключом соединения в реляционной базе данных. В нашем примере групповым ключом является идентификатор заказчика (Customer ID). Пакет datajoin позволяет использовать в качестве группового ключа произвольную определенную пользователем функцию, так что это понятие оказывается более общим, чем ключ соединения в реляционной базе данных.

Прежде чем переходить к описанию работы с этим пакетом, давайте рассмотрим основные шаги соединения с переразбиением методом сортировки слиянием в применении к нашим игрушечным наборам данных. Разобравшись, как все сопрягается между собой, мы сможем понять, какие шаги выполняет пакет datajoin, а какие мы должны запрограммировать самостоятельно. Мы покажем, как наш код интегрируется с пакетом datajoin.

Поток данных при соединении на стороне редуктора

На рис. 5.1 показан поток данных при выполнении соединения с переразбиением для наборов Customers и Orders, вплоть до этапа редукции. Что происходит на этапе редукции, мы расскажем позже.

Прежде всего, видно, что распределители получают данные из двух файлов, Customers и Orders. Каждый распределитель знает имя файла того потока данных, который обрабатывает. Для каждой записи вызывается функция `map()`, основная задача которой – упаковать запись так, чтобы стало возможно произвести соединение на стороне редуктора.

Напомним, что в каркасе MapReduce функция `map()` выводит записи в виде пар ключ/значение, которые разбиваются по ключу, и далее все записи с одинаковыми ключами поступают одному редуктору и обрабатываются вместе. В случае соединения мы хотели бы, чтобы функция `map()` выводила упакованную запись, в которой ключом является групповой ключ для соединения – в нашем примере Customer ID. Значением же будет исходная запись, помеченная источником данных (то есть именем файла). Так, для записи

```
3,A,12.95,02-Jun-2008
```

из файла Orders функция `map()` выведет пару, в которой ключом будет «3» – идентификатор заказчика, по которому производится соединение с записями из файла Customers. А значением будет вся запись, помеченная тегом «`Orders`».

После того как `map()` упакует все входные записи, производятся стандартные операции MapReduce: разбиение, тасование и сортировка. Так как в качестве группового ключа был задан ключ соединения, то функция `reduce()` будет обрабатывать все записи с одним и тем же ключом соединения вместе. Эта функция извлечет из пакета исходную запись и ее источник данных, указанный в теге. Легко видеть, что для групповых ключей «1» и «2» `reduce()` получит два значения, одно из которых будет помечено тегом «`Customers`»,

а другое – тегом «Orders». Для порожденных распределителем данных с групповым ключом «4» reduce() увидит только одно значение, помеченное тегом «Customers». Это вполне ожидаемо, так как в файле Orders нет записи, в которой Customer ID равно «4». С другой стороны, для группового ключа «3» reduce() увидит три значения, одно из которых поступило из файла Customers, а два других – из файла Orders.

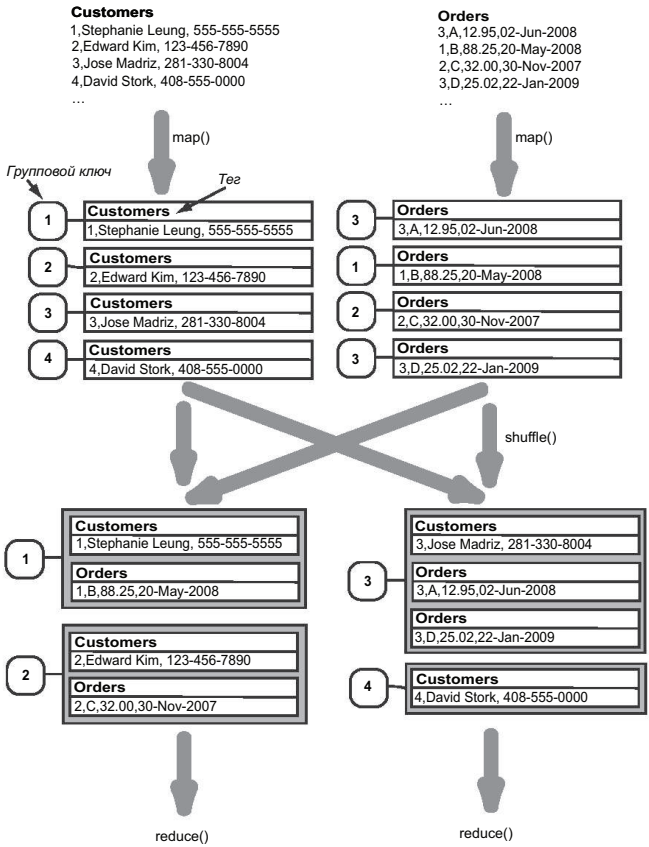


Рис. 5.1. При выполнении соединения с переразбиением распределитель сначала упаковывает каждую запись, добавляя групповой ключ и тег. В качестве группового ключа берется атрибут, по которому производится соединение, а в качестве тега – источник данных (в SQL он называется *таблицей*). На этапе разбиения и тасования все записи с одинаковыми групповыми ключами объединяются в одну группу. Эта группа передается одному редуктору.

Функция `reduce()` получает входные данные и вычисляет *перекрестное произведение* значений, то есть порождает все комбинации значений с условием, что никакая комбинация не должна содержать несколько экземпляров одного и того же тега. Если `reduce()` видит разные значения тегов, то в перекрестное произведение включаются наборы значений из исходных записей. В нашем примере так обстоит дело с групповыми ключами 1, 2 и 4. На рис. 5.2 показано, как строится перекрестное произведение для группового ключа 3. Здесь есть три значения: одно помечено тегом Customers и два – тегом Orders. В перекрестном произведении образуются две комбинации, так что в каждой есть по одному значению из Customers и из Orders.

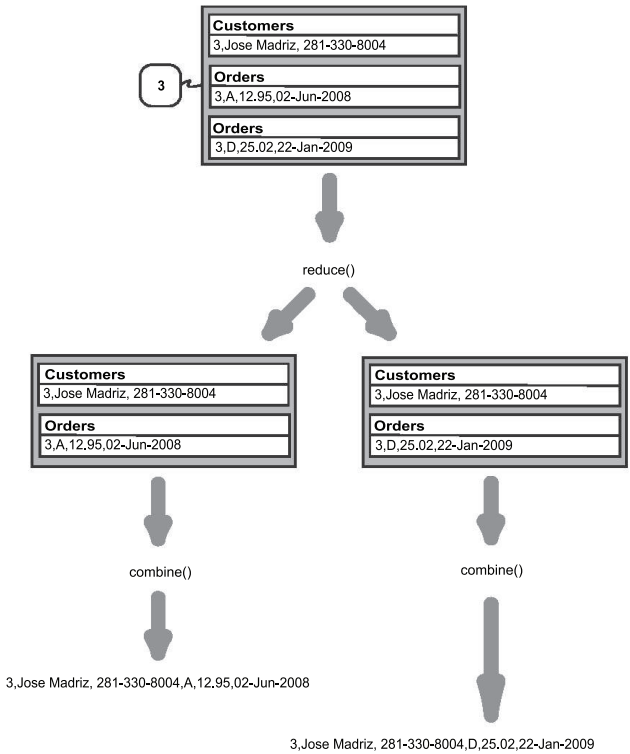


Рис. 5.2. Часть соединения с переразбиением, выполняемая на стороне редуктора. Для данного группового ключа редуктор вычисляет полное перекрестное произведение значений из разных источников. Каждая комбинация передается функции `combine()` для создания выходной записи. Эта функция вправе решить, что некую комбинацию выводить не следует.

Примечание. В нашем примере имеется неявно подразумеваемое допущение о том, что Customer ID – уникальный идентификатор в файле Customers, поэтому количество комбинаций с данным Customer ID в перекрестном произведении всегда равно количеству записей в файле Orders с этим Customer ID (кроме случая, когда в файле Orders нет таких записей, и тогда в перекрестном произведении оказывается сама запись из файла Customers). В более сложных случаях количество комбинаций в перекрестном произведении равно произведению числа записей, помеченных каждым тегом. Если `reduce()` видит две записи из Customers и три записи из Orders, то перекрестное произведение будет включать шесть (2 * 3) комбинаций. Если есть еще и третий источник данных (Accounts), содержащий две записи, то перекрестное произведение будет включать двенадцать (2 * 2 * 3) комбинаций.

Редуктор передает каждую комбинацию из перекрестного произведения функции `combine()` (не путайте ее с комбинаторами, о которых шла речь в разделе 4.5). Сама природа перекрестного произведения такова, что `combine()` гарантированно увидит не более одной записи из каждого источника данных (с соответствующим тегом), причем во всех увиденных им записях ключ соединения один и тот же. Именно функция `combine()` решает, какую операцию выполнять: внутреннее соединение, внешнее соединение или еще какой-то вид соединения. Если вычисляется внутреннее соединение, то `combine()` отбрасывает комбинации, в которых присутствуют не все теги; в нашем примере это комбинация с групповым ключом «4». В противном случае `combine()` объединяет записи из разных источников в одну выходную запись.

Теперь вы понимаете, почему эта процедура называется соединением с переразбиением методом сортировки слиянием. Записи в источниках данных могут следовать в произвольном порядке. Они *переразбиваются* на правильные группы до передачи редукторам. Затем редуктор *объединяет* записи с одинаковым ключом соединения, чтобы создать требуемые выходные записи. (*Сортировка* тоже имеет место, но для понимания процесса в целом она не очень существенна.)

Реализация соединения с помощью пакета datajoin

Входящий в состав Hadoop пакет datajoin реализует описанный выше поток данных. В нем предусмотрено несколько точек подключения для задания конкретной структуры данных и специальная точка для определения функции `combine()`.

В пакете datajoin имеются три абстрактных класса, которые конкретизируются в пользовательском коде: `DataJoinMapperBase`, `DataJoinReducerBase` и `TaggedMapOutput`. Наш класс `MapClass`

расширяет `DataJoinMapperBase`, а класс `Reduce` расширяет `DataJoinReducerBase`. В пакете datajoin в соответствующих базовых классах уже реализованы методы `map()` и `reduce()`, которые обеспечивают описанный выше поток данных. В своем подклассе мы должны лишь реализовать несколько новых методов для настройки деталей.

Перед тем как описывать работу с классами `DataJoinMapperBase` и `DataJoinReducerBase`, следует разобраться с назначением нового абстрактного типа данных `TaggedMapOutput`, который повсеместно используется в коде. Напомним, что в потоке данных распределитель выводит упакованную запись с групповым ключом и исходной помеченной записью в качестве значения. В пакете datajoin постулируется, что групповой ключ должен иметь тип `Text`, а значение (помеченная запись) – тип `TaggedMapOutput`. Тип данных `TaggedMapOutput` предназначен для пополнения записей тегом типа `Text`. В нем имеются тривиальные реализации методов `getTag()` и `setTag(Text tag)`, а также определен абстрактный метод `getData()`, который должен быть реализован в нашем подклассе для обработки записей данного типа. Явно не требуется, чтобы в подклассе был реализован метод `setData()`, но, если он присутствует, то ему следует передавать данные записи. Подкласс может для симметрии реализовать метод `setData()` или же принимать запись в качестве аргумента конструктора. Кроме того, будучи выходом распределителя, класс `TaggedMapOutput` должен реализовывать интерфейс `Writable`. Следовательно, в нашем подклассе должны быть методы `readFields()` и `write()`. Мы написали простой подкласс `TaggedWritable`, умеющий обрабатывать любую запись типа `Writable`.

```
public static class TaggedWritable extends TaggedMapOutput {

   private Writable data;

   public TaggedWritable(Writable data) {
      this.tag = new Text("");
      this.data = data;
   }

   public Writable getData() {
      return data;
   }

   ...
}
```

Еще раз напомним, что основная функция распределителя в потоке данных – упаковать запись таким образом, чтобы все записи

с одинаковым ключом соединения попадали одному и тому же редуктору. Класс `DataJoinMapperBase` осуществляет необходимую упаковку, но в нем определены три абстрактных метода, которые должны быть реализованы в подклассах:

```
protected abstract Text generateInputTag(String inputFile);
protected abstract TaggedMapOutput generateTaggedMapOutput(Object value);
protected abstract Text generateGroupKey(TaggedMapOutput aRecord);
```

Метод `generateInputTag()` вызывается в начале операции map с целью задать глобальный тег для всех записей, которые эта операция будет обрабатывать. Требуется, чтобы тег имел тип `Text`. Отметим, что методу `generateInputTag()` передается имя файла, содержащего входные записи. Если распределитель работает с файлом Customers, то этот метод получит в качестве аргумента строку `"Customers"`. Поскольку мы используем тег для обозначения источника данных, а источник данных определяется именем файла, то код метода `generateInputTag()` выглядит так:

```
protected Text generateInputTag(String inputFile) {
   return new Text(inputFile);
}
```

Если источник данных распределен по нескольким файлам (part-0000, part-0001 и т. д.), то в качестве тега следует выбирать не полное имя файла, а его начальную часть. Например, это может быть часть имени файла перед знаком минус:

```
protected Text generateInputTag(String inputFile) {
   String datasource = inputFile.split('-')[0];
   return new Text(datasource);
}
```

Результат, возвращенный методом `generateInputTag()`, сохраняется в переменной `inputTag` объекта `DataJoinMapperBase` для последующего использования. Если впоследствии потребуется также обращаться к имени файла, то его можно сохранить в переменной `inputFile` того же объекта.

После инициализации для каждой записи вызывается метод `map()` объекта `DataJoinMapperBase`. При этом он обращается к двум абстрактным методам, которые нам еще предстоит реализовать.

```
public void map(Object key, Object value,
                OutputCollector output, Reporter reporter) throws
IOException
{
   TaggedMapOutput aRecord = generateTaggedMapOutput(value);
```

```
    Text groupKey = generateGroupKey(aRecord);
    output.collect(groupKey, aRecord);
}
```

Метод `generateTaggedMapOutput()` обертывает исходную запись типом `TaggedMapOutput`. Напомним, что мы пользуемся конкретным подклассом `TaggedMapOutput`, который назвали `TaggedWritable`. Метод `generateTaggedMapOutput()` может возвращать объект `TaggedWritable` с любым тегом типа `Text`. В принципе можно даже возвращать разные теги для разных записей, взятых из одного файла. Но в стандартном случае мы хотим, чтобы тег представлял источник данных и совпадал со значением, которое ранее было вычислено методом `generateInputTag()` и сохранено в переменной `this.inputTag`.

```
protected TaggedMapOutput generateTaggedMapOutput(Object value) {
    TaggedWritable retv = new TaggedWritable((Text) value);
    retv.setTag(this.inputTag);
    return retv;
}
```

Метод `generateGroupKey()` принимает помеченную запись (типа `TaggedMapOutput`) и возвращает групповой ключ для соединения. Нам сейчас нужно просто распаковать помеченную запись и взять в качестве ключа соединения первое поле (запись представлена в формате CSV).

```
protected Text generateGroupKey(TaggedMapOutput aRecord) {
    String line = ((Text) aRecord.getData()).toString();
    String[] tokens = line.split(",");
    String groupKey = tokens[0];
    return new Text(groupKey);
}
```

В более общей реализации пользователь мог бы указать, какое поле должно быть ключом соединения и нужно ли использовать в качестве разделителя не запятую, а какой-то другой символ. `DataJoinMapperBase` – простой класс, и значительная часть кода распределителя находится в нашем подклассе. Рабочей же лошадкой пакета datajoin является класс `DataJoinReducerBase`, который и выполняет полное внешнее соединение. Наш подкласс должен только реализовать метод `combine()`, который отфильтровывает ненужные комбинации, оставляя лишь те, что должны присутствовать в соединении интересующего нас вида (внутреннем, левом внешнем и т. д.). Кроме того, в методе `combine()` производится форматирование комбинации для вывода.

Методу `combine()` передается одна комбинация из перекрестного произведения помеченных записей с одним и тем же групповым ключом. Если это не сразу понятно, взгляните еще раз на диаграммы потока данных на рис. 5.1 и 5.2 и вспомните, что в каноническом случае двух источников данных перекрестное произведение – вещь совсем простая. Каждая комбинация содержит либо две записи (это означает, что в каждом источнике есть хотя бы одна запись с данным ключом соединения), либо одну запись (тогда данный ключ присутствует только в одном источнике данных).

Рассмотрим сигнатуру метода `combine()`:

```
protected abstract TaggedMapOutput
                  ➥ combine(Object[] tags, Object[] values);
```

Комбинация представлена массивом тегов и массивом значений. Гарантируется, что размеры обоих массивов одинаковы и равны количеству помеченных записей в комбинации. Первая помеченная запись представлена элементами `tags[0]` и `values[0]`, вторая – элементами `tags[1]` и `values[1]` и так далее. При этом записи отсортированы по тегам.

Поскольку теги соответствуют источникам данных, то в каноническом случае соединения двух источников размер массива `tags`, переданного методу `combine()`, не превышает 2. На рис. 5.2 видно, что метод `combine()` вызывается дважды. Для левой части массивы `tags` и `values` таковы[2]:

```
tags = {"Customers", "Orders"};
values = {"3,Jose Madriz,281-330-8004", "A,12.95,02-Jun-2008"};
```

В случае внутреннего соединения метод `combine()` игнорирует комбинации, в которых присутствуют не все теги, и возвращает `null`. Если же комбинация допустима, то `combine()` должен конкатенировать все значения в одну выходную запись. Порядок конкатенации полностью определяется методом `combine()`. В случае внутреннего соединения длина массива `values[]` всегда будет равна количеству имеющихся источников данных (в каноническом случае – два), а теги отсортированы. Поэтому будет разумно в цикле обойти массив `values[]`, чтобы получить подразумеваемое по умолчанию алфавитное упорядочение, основанное на именах источников данных.

Класс `DataJoinReducerBase`, как и любой редуктор, выводит пары ключ/значение. Для каждой допустимой комбинации ключом всегда

[2] Массив `tags` имеет тип `Text[]`, а массив `values` – тип `TaggedWritable[]`. Но эти детали мы игнорируем, сосредоточившись на содержимом массивов.

является ключ соединения, а значением – результат, возвращенный методом `combine()`. Отметим, что ключ соединения по-прежнему присутствует в каждом элементе массива `values[]`. Метод `combine()` должен исключить из этих элементов ключ соединения перед тем, как конкатенировать их. В противном случае ключ соединения будет несколько раз входить в одну выходную запись.

Наконец, `DataJoinReducerBase` ожидает, что `combine()` вернет значение типа `TaggedMapOutput`. Правда, не вполне понятно, почему, ведь `DataJoinReducerBase` игнорирует тег в объекте `TaggedMapOutput`.

В листинге 5.3 приведен полный код, включая и наш подкласс редуктора.

Листинг 5.3.
Внутреннее соединение данных из двух файлов на стороне редуктора

```
public class DataJoin extends Configured implements Tool {

   public static class MapClass extends DataJoinMapperBase {

      protected Text generateInputTag(String inputFile) {
         String datasource = inputFile.split("-")[0];
         return new Text(datasource);
      }

      protected Text generateGroupKey(TaggedMapOutput aRecord) {
         String line = ((Text) aRecord.getData()).toString();
         String[] tokens = line.split(",");
         String groupKey = tokens[0];
         return new Text(groupKey);
      }

      protected TaggedMapOutput generateTaggedMapOutput(Object value) {
         TaggedWritable retv = new TaggedWritable((Text) value);
         retv.setTag(this.inputTag);
         return retv;
      }
   }

   public static class Reduce extends DataJoinReducerBase {

      protected TaggedMapOutput combine(Object[] tags, Object[] values) {
         if (tags.length < 2) return null;
         String joinedStr = "";
         for (int i=0; i<values.length; i++) {
            if (i > 0) joinedStr += ",";
```

```
            TaggedWritable tw = (TaggedWritable) values[i];
            String line = ((Text) tw.getData()).toString();
            String[] tokens = line.split(",", 2);
            joinedStr += tokens[1];
         }
        TaggedWritable retv = new TaggedWritable(new Text(joinedStr));
         retv.setTag((Text) tags[0]);
         return retv;
      }
   }

   public static class TaggedWritable extends TaggedMapOutput {

      private Writable data;

      public TaggedWritable(Writable data) {
         this.tag = new Text("");
         this.data = data;
      }

      public Writable getData() {
         return data;
      }

      public void write(DataOutput out) throws IOException {
         this.tag.write(out);
         this.data.write(out);
      }

      public void readFields(DataInput in) throws IOException {
         this.tag.readFields(in);
         this.data.readFields(in);
      }
   }

   public int run(String[] args) throws Exception {
      Configuration conf = getConf();

      JobConf job = new JobConf(conf, DataJoin.class);
      Path in = new Path(args[0]);
      Path out = new Path(args[1]);

      FileInputFormat.setInputPaths(job, in);
      FileOutputFormat.setOutputPath(job, out);

      job.setJobName("DataJoin");
      job.setMapperClass(MapClass.class);
      job.setReducerClass(Reduce.class);
      job.setInputFormat(TextInputFormat.class);
      job.setOutputFormat(TextOutputFormat.class);
```

```
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(TaggedWritable.class);
        job.set("mapred.textoutputformat.separator", ",");

        JobClient.runJob(job);
        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new DataJoin(),
                                 args);
        System.exit(res);
    }
}
```

Далее мы рассмотрим другой способ выполнения соединений, который в некоторых распространенных приложениях оказывается более эффективным.

5.2.2. Построение реплицированных соединений с помощью класса DistributedCache

Описанная в предыдущем разделе техника соединения на стороне редуктора обеспечивает гибкость, но может оказаться крайне неэффективной. Соединение откладывается до этапа редукции. Мы сначала тасуем и передаем по сети все данные, хотя во многих случаях отбрасываем значительную их часть при вычислении соединения. Было бы эффективнее устранить ненужные данные еще на этапе распределения. А еще лучше вычислять соединение целиком на этапе распределения.

Основным препятствием на этом пути является тот факт, что запись, обрабатываемую распределителем, возможно, потребуется соединить с записью, которую распределителю не так-то легко получить (или даже узнать, где она находится). Если можно гарантировать доступность всех необходимых для соединения записей данных, то соединение можно произвести на стороне распределителя. Например, если мы знаем, что оба источника данных разбиты на одинаковое число секций *и* все эти секции отсортированы по ключу *и* этот ключ как раз и является требуемым ключом соединения, то каждый распределитель (при надлежащем задании `InputFormat` и `RecordReader`) гарантированно может найти и получить все данные, необходимые для

вычисления соединения. На самом деле, входящий в состав Hadoop пакет org.apache.hadoop.mapred.join содержит вспомогательные классы, упрощающие соединение на стороне распределителя. К сожалению, ситуаций, когда эта техника естественно применима, не так уж много, а запуск дополнительных задач MapReduce для переразбиения источников данных в соответствии с требованиями этого пакета сводит на нет весь выигрыш. Поэтому больше мы обсуждать этот пакет не будем.

Однако не все потеряно. Существует еще один случай, который встречается довольно часто. При соединении больших массивов данных нередко бывает, что велик лишь один источник, а второй на много порядков меньше. Например, файл Customers с данными о клиентах телефонной компании может содержать всего несколько десятков миллионов записей (по одной записи с основными сведениями для каждого клиента), тогда как журнал операций – миллиарды записей с подробной информацией и каждом вызове. Если меньший источник целиком помещается в память распределителя, то мы можем получить гигантский выигрыш в производительности, скопировав меньший источник на все распределители и произведя соединение на этапе распределения. В литературе по базам данных это называется *реплицированным соединением*, потому что одна из таблиц данных реплицируется на все узлы кластера. (В следующем разделе мы рассмотрим случай, когда меньший источник не помещается в память.)

В Hadoop имеется механизм *распределенного кэша*, предназначенный для распределения файлов по всем узлам кластера. Обычно он применяется для распространения файлов, содержащих вспомогательную информацию, необходимую всем распределителям. Например, если Hadoop используется для классификации документов, то для каждого класса может быть определен список ключевых слов (или – еще лучше – вероятностная модель, но это мы отвлеклись...). Распределенный кэш может гарантировать, что к таким спискам ключевых слов будут иметь доступ все распределители. Для выполнения реплицированного соединения в роли вспомогательных данных будет выступать меньший источник данных.

Распределенный кэш реализован с помощью класса `DistributedCache`. Работа с ним сводится к двум шагам. Сначала, при конфигурировании задачи вызывается статический метод `DistributedCache.addCacheFile()` для задания файлов, которые нужно распространить на все узлы. Файлы описываются объектами URI и по умолчанию находятся в файловой системе HDFS, если

явно не указана иная. Демон JobTracker получает этот список URI-адресов и при запуске задачи создает локальные копии файлов на всех узлах TaskTracker. На втором шаге распределители на каждом узле TaskTracker вызывают статический метод `DistributedCache.getLocalCacheFiles()`, чтобы получить массив путей к локальным файлам, содержащим копии. В этот момент распределитель может пользоваться стандартными имеющимися в Java средствами ввода/вывода для чтения локальной копии.

Реплицированное соединение с помощью `DistributedCache` проще, чем соединение на стороне редуктора. Начнем со стандартного шаблона Hadoop-программы.

```
public class DataJoinDC extends Configured implements Tool {

   public static class MapClass extends MapReduceBase
      implements Mapper<Text, Text, Text, Text> {

      ...
   }

   public int run(String[] args) throws Exception {
      ...
   }

   public static void main(String[] args) throws Exception {
      int res = ToolRunner.run(new Configuration(),
                               new DataJoinDC(),
                               args);
      System.exit(res);
   }
}
```

Отметим, что класс `Reduce` мы исключили. Мы планируем выполнять соединение на этапе распределения и конфигурируем эту задачу вообще без редукторов. Драйвер для этой задачи вам тоже уже хорошо знаком.

```
public int run(String[] args) throws Exception {
   Configuration conf = getConf();
   JobConf job = new JobConf(conf, DataJoinDC.class);

   DistributedCache.addCacheFile(new Path(args[0]).toUri(), conf);

   Path in = new Path(args[1]);
   Path out = new Path(args[2]);
   FileInputFormat.setInputPaths(job, in);
   FileOutputFormat.setOutputPath(job, out);
```

```
    job.setJobName("DataJoin with DistributedCache");
    job.setMapperClass(MapClass.class);
    job.setNumReduceTasks(0);
    job.setInputFormat(KeyValueTextInputFormat.class);
    job.setOutputFormat(TextOutputFormat.class);
    job.set("key.value.separator.in.input.line", ",");

    JobClient.runJob(job);

    return 0;
}
```

Самое важное новшество здесь – передача первого аргумента классу `DistributedCache`. При запуске задачи каждый узел создает локальную копию этого файла, хранящегося в HDFS. Второй и третий аргументы – соответственно входной и выходной путь стандартной задачи Hadoop. Отметим, что мы ограничили количество источников данных двумя. Это не внутреннее ограничение описываемой техники, а допущение, принятое для того, чтобы код было легче понять.

До сих пор мы определяли в своем классе `MapClass` только метод `map()`. На самом деле, в интерфейсе `Mapper` (и `Reducer`) есть еще два абстрактных метода: `configure()` и `close()`. Метод `configure()` вызывается при создании объекта `MapClass`, а метод `close()` – когда распределитель заканчивает обработку своей порции. В классе `MapReduceBase` содержатся пустые реализации этих методов. Сейчас нам нужно переопределить метод `configure()`, так чтобы он загружал соединяемые данные в память в момент инициализации. Тогда эти данные будут доступны при каждом вызове метода `map()` для обработки новой записи.

```
public static class MapClass extends MapReduceBase
    implements Mapper<Text, Text, Text, Text> {

    private Hashtable<String, String> joinData =
                                    new Hashtable<String, String>();

    @Override
    public void configure(JobConf conf) {
        try {
          Path [] cacheFiles = DistributedCache.getLocalCacheFiles(conf);
           if (cacheFiles != null && cacheFiles.length > 0) {
              String line;
              String[] tokens;
              BufferedReader joinReader = new BufferedReader(
                 new FileReader(cacheFiles[0].toString()));
              try {
                 while ((line = joinReader.readLine()) != null) {
```

```
                tokens = line.split(",", 2);
                joinData.put(tokens[0], tokens[1]);
              }
            } finally {
              joinReader.close();
            }
          }
      } catch (IOException e) {
        System.err.println("Exception reading DistributedCache: " + e);
      }
    }

    public void map(Text key, Text value,
                    OutputCollector<Text, Text> output,
                    Reporter reporter) throws IOException {
      String joinValue = joinData.get(key);
      if (joinValue != null) {
        output.collect(key,
                       new Text(value.toString() + "," + joinValue));
      }
    }
}
```

После вызова метода `configure()` мы получаем массив путей к локальным копиям файлов, распространенных объектом `DistributedCache`. Поскольку наш метод-драйвер поместил в `DistributedCache` только один файл (переданный ему в первом аргументе), то массив будет иметь размер 1. Мы читаем файл стандартными средствами Java. В нашей программе предполагается, что каждая строка – это запись, что ключ и значение разделены запятой, что ключ уникален и именно по нему производится соединение. Программа считывает исходный файл в объект `joinData` типа `Hashtable`, и эта хеш-таблица будет доступна на протяжении всей жизни распределителя.

Вычисление соединения происходит в методе `map()` и теперь, когда один из источников данных находится в памяти в виде объекта `joinData`, не представляет никаких сложностей. Если ключ соединения отсутствует в `joinData`, то запись отбрасывается. В противном случае ключ (соединения) сопоставляется со значением, хранящимся в `joinData`, и значения конкатенируются. Результат выводится непосредственно в HDFS, поскольку редуктор для дальнейшей обработки не нужен.

Не так уж редко при использовании `DistributedCache` возникает ситуация, когда вспомогательные данные (в нашем случае меньший источник данных) находятся в локальной файловой системе клиента, а не хранятся в HDFS. Один из способов решения этой проблемы –

добавить код для загрузки локального файла клиента в HDFS перед обращением к `DistributedCache.addCacheFile()`. К счастью, эта процедура уже поддерживается в Hadoop с помощью одного из стандартных аргументов командной строки, распознаваемых классом `GenericOptionsParser`. Это флаг `-files`, который позволяет автоматически скопировать перечисленные через запятую файлы на все узлы, исполняющие задания. В нашем случае команду следует записать так:

```
bin/hadoop jar -files small_in.txt DataJoinDC.jar big_in.txt output
```

Теперь мы не вызываем метод `DistributedCache.addCacheFile()` самостоятельно, поэтому можем не включать имя файла с меньшим набором данных в состав аргументов. Следовательно, индексы аргументов изменились:

```
Path in = new Path(args[0]);
Path out = new Path(args[1]);
```

После этих мелких изменений программа соединения воспринимает локальный файл на клиентской машине как один из входных источников.

5.2.3. Полусоединение: соединение на стороне редуктора с фильтрацией на стороне распределителя

Одно из ограничений реплицированного соединения состоит в том, что одна из соединяемых таблиц должна быть достаточно мала, чтобы целиком поместиться в память. Даже при часто встречающейся асимметрии размеров входных источников меньший все же может оказаться достаточно объемным. Решить эту проблему можно путем изменения порядка шагов обработки с целью сделать всю процедуру более эффективной. Например, если нас интересует история заказов для всех заказчиков в регионе с телефонным кодом 415, то в принципе можно соединить таблицы Orders и Customers до того, как отфильтрованы все заказчики не из этого региона. Это правильно, но неэффективно. Обе таблицы могут быть слишком велики для реплицированного соединения, и тогда придется прибегнуть к неэффективному соединению на стороне редуктора. Гораздо лучше было бы оставить только заказчиков, проживающих в регионе с телефонным кодом 415, отбросив всех остальных. Этих

заказчиков можно сохранить во временном файле с именем Customers415. Результат соединения Orders с Customers415 будет точно таким же, как и раньше, но теперь размер таблицы Customers415 таков, что реплицированное соединение возможно. С созданием и распространением файла связаны некоторые накладные расходы, но зачастую они с лихвой компенсируются общим выигрышем в производительности.

Иногда объем подлежащих анализу данных очень велик и использовать реплицированное соединение не получается, как бы ни изменять порядок шагов обработки. Но не надо отчаиваться. Все равно остаются способы повысить эффективность соединения на стороне редуктора. Напомним, что основная проблема заключалась в том, что распределитель только помечает данные, но все они после тасования передаются по сети, хотя большую их часть редуктор игнорирует. Неэффективность можно устранить, снабдив распределитель дополнительной функцией предварительной фильтрации, которая отбросит большинство или даже все ненужные данные до передачи по сети. Вот такой механизм фильтрации нам и предстоит сконструировать.

Вернемся к примеру соединения таблиц Customers415 и Orders. Здесь ключом соединения является Customer ID, и мы хотели бы, чтобы распределители отфильтровывали всех заказчиков не из региона 415, а не посылали эти записи редукторам. Создадим набор данных CustomerID415, в котором будут храниться идентификаторы всех заказчиков из региона 415. Файл CustomerID415 меньше, чем Customers415, поскольку в нем всего одно поле данных. В предположении, что CustomerID415 помещается в память, мы можем улучшить соединение на стороне редуктора, воспользовавшись распределенным кэшем для распространения файла CustomerID415 по распределителям. При обработке наборов Customers и Orders распределитель будет отбрасывать записи, ключ которых отсутствует в CustomerID415. Иногда эту операцию называют полусоединением, заимствуя термин из теории баз данных.

И последнее – что если файл CustomerID415 все-таки не помещается в память? Или помещается, но настолько велик, что его репликация всем распределителям неэффективна? В такой ситуации на помощь приходит структура данных, которая называется *фильтр Блума*. Это компактное представление множества, поддерживающее только запросы о *вхождении* («содержит ли множество данный элемент?»). Кроме того, ответы на подобные запросы не обязательно

точны. Гарантируется лишь, что ложный ответ «нет» невозможен, а вероятность ложного ответа «да» мала. Небольшая неточность – это плата за компактность структуры данных. Используя представление CustomerID415 в виде фильтра Блума, распределители наверняка передадут редукторам всех заказчиков из региона 415, поэтому правильность работы алгоритма соединения гарантирована. Правда, фильтр Блума пропустит также сколько-то заказчиков из других регионов, но это не страшно, потому что редукторы их все равно проигнорируют. А общая производительность возрастет за счет резкого уменьшения сетевого трафика после тасования. Фильтры Блума – это стандартная техника соединения в распределенных базах данных, они используются и в коммерческих продуктах, например Oracle 11g. Подробнее фильтр Блума и его различные применения мы рассмотрим в следующем разделе.

5.3. Создание фильтра Блума

Если вы используете Hadoop для пакетной обработки больших наборов данных, то, наверное, вам нужна и транзакционность. Мы не будем рассматривать все технические приемы распределенной обработки данных в режиме реального времени (кэширование, секционирование и т. д.). Многие из них никак не связаны с Hadoop и далеко выходят за рамки этой книги. Но об одном из малоизвестных методов обработки данных в реальном времени, фильтре Блума, мы поговорим. Это дайджест набора данных, позволяющий повысить эффективность других способов обработки. Если набор данных велик, Hadoop часто просят сгенерировать его представление в виде фильтра Блума. Выше мы уже отмечали, что иногда фильтр Блума используется для соединения данных и внутри самого каркаса Hadoop. Любому специалисту по обработке данных полезно иметь фильтр Блума в своем активе. В этом разделе мы рассмотрим эту структуру данных более детально и применим ее к примеру онлайновой рекламной сети.

5.3.1. Что делает фильтр Блума?

По существу, в классе, реализующем фильтр Блума, есть всего два метода: `add()` и `contains()`. Работают они так же, как одноименные методы в интерфейсе Java `Set`. Метод `add()` добавляет объект в множество, а метод `contains()` возвращает булевское значение `true` или

`false` в зависимости от того, принадлежит объект множеству или нет. Но в случае фильтра Блума метод `contains()` не всегда дает правильный результат. Ложный ответ «нет, не принадлежит» он не дает никогда. Если `contains()` возвращает `false`, то можно быть уверенным, что множество не содержит запрошенный объект. Однако есть небольшая вероятность получить ложный ответ «да, содержит». Метод `contains()` может возвращать `true` даже, если объект не принадлежит множеству. Вероятность такой ошибки зависит от количества элементов в множестве и от некоторых конфигурационных параметров самого фильтра Блума.

Основное достоинство фильтра Блума состоит в том, что его размер, выраженный в количестве бит, постоянен и задается в момент инициализации. Добавление элементов не увеличивает размер, а лишь повышает вероятность ложноположительного ответа. У фильтра Блума есть и еще один конфигурационный параметр, задающий количество используемых хеш-функций. Зачем этот параметр нужен и как используются хэш-функции, мы расскажем позже, когда будем обсуждать реализацию фильтра Блума. А пока отметим лишь, что он влияет на вероятность ложноположительного ответа. Эта вероятность аппроксимируется величиной

$$(1 - \exp(-kn/m))k,$$

где k – количество используемых хеш-функций, m – количество битов, отведенных под фильтр Блума, а n – количество элементов, добавленных в фильтр. На практике m и n определяются требованиями к системе, а k выбирается так, чтобы минимизировать вероятность ложноположительного ответа при заданных m и n. Простое преобразование дает:

$$k = \ln(2) * (m/n) \approx 0.7 * (m/n).$$

Вероятность ложноположительного ответа при данном k равна $0{,}6185^{m/n}$, и k должно быть целым числом. Это лишь приближенное значение вероятности. Проектировщик должен принимать во внимание (m/n), то есть число бит на элемент, а не просто m. Например, пусть нужно представить множество из 10 миллионов URL-адресов (n = 10 000 000). Если отвести 8 бит на каждый URL (m/n = 8), то понадобится фильтр Блума размером 10 МБ) (m = 80 000 000 бит). Для такого фильтра вероятность ложноположительного ответа составит 0,6185 * 8, то есть около 2 процентов. Если же реализовывать класс `Set`, в котором будут храниться сами URL-адреса, то в предположении, что средняя длина URL равна 100 байтов, нам потребовалось бы

1 ГБ памяти. Фильтр Блума позволит уменьшить требования к памяти на два порядка ценой двухпроцентной вероятности ложноположительного ответа! Если немного увеличить отведенную под фильтр Блума память, то эту вероятность можно еще снизить. При 10 бит на URL фильтр будет занимать 12,5 МБ, а вероятность ложноположительного ответа составит 0,8 процента.

Итак, приведем сигнатуру класса фильтра Блума:

```
class BloomFilter<E> {
   public BloomFilter(int m, int k) { ... }
   public void add(E obj) { ... }
   public boolean contains(E obj) { ... }
}
```

Другие применения фильтра Блума

Первые успешные применения фильтра Блума относятся к временам, когда память была дефицитна. Один из ранних примеров – проверка правописания. Не имея возможности хранить в памяти весь словарь, программы проверки правописания использовали его представление в виде фильтра Блума, и это позволяло обнаружить большинство ошибок. По мере того как память становилась дешевле и объемнее, необходимость в такой экономии отпала. Второе рождение фильтр Блума пережил, когда на сцену вышли операции с большими объемами данных, поскольку очень скоро для обработки данных перестало хватать и памяти, и пропускной способности сети.

Мы уже упоминали, что в коммерческих продуктах, например Oracle 11g, фильтры Блума используются для соединения данных, хранящихся в распределенной базе. Среди сетевых продуктов также имеется пример успешного применения фильтра Блума – распределенный прокси-сервер с открытым исходным кодом Squid. Этот сервер кэширует веб-страницы, к которым часто производятся обращения, что позволяет сэкономить полосу пропускания сети и быстрее возвращать ответы пользователям. В кластере серверов Squid каждый может кэшировать разные подмножества содержимого. Входящий запрос будет направлен серверу Squid, который хранит копию запрошенного содержимого, а, если в кэше нужного материала не окажется, запрос будет передан исходному серверу. Механизм маршрутизации должен знать, что именно хранится на каждом сервере Squid. Поскольку пересылать списки URL-адресов для каждого сервера и хранить их в памяти дорого, используются фильтры Блума. Ложноположительный ответ означает, что запрос будет направлен не тому серверу Squid, в этом случае содержимого не окажется в кэше, и Squid переадресует запрос исходному веб-серверу, так что операция в целом будет выполнена корректно. Небольшие издержки, обусловленные ложноположительными ответами, с лихвой окупаются общим повышением производительности.

Похожий, но более сложный пример дают системы секционирования (sharding). В основе своей секционирование – это разнесение базы данных на несколько машин так, что каждая машина имеет дело только с подмножеством всех записей. У каждой записи имеется некоторый идентификатор,

определяющий, к какой машине она приписана. Самый простой подход – вычислять идентификатор в виде статической свертки, определяющей одну из фиксированного набора машин, обслуживающих базу. Но это негибкое решение, так как оно не допускает добавления новых секций или перебалансирования существующих. Более гибкий вариант – использовать динамический поиск по идентификатору записи, но, если такой поиск производится средствами СУБД (то есть на диске), то возрастает время обработки. Поэтому в более развитых системах секционирования тоже применяется фильтр Блума, хранящийся в памяти. Необходим, правда, какой-то механизм обработки ложноположительных ответов, но их частота столь мала, что не влияет на общую производительность.

В онлайновых рекламных сетях важно умение показывать рекламное объявление той категории посетителей, для которой оно предназначено. Если учесть объем трафика в типичной рекламной сети и требования к задержке, то получается, что оборудование, необходимое для поиска категории в режиме реального времени, будет очень и очень дорогим. Однако использование фильтров Блума позволяет резко снизить его стоимость. В автономном режиме пометим веб-страницы (или посетителей) тегами, описывающими ограниченное число категорий (спорт, семья, музыка и т. д.). Для каждой категории построим фильтр Блума и будем хранить его в памяти рекламного сервера. Получив запрос, рекламный сервер сможет быстро и без больших затрат, определить, из какой категории брать рекламное объявление. Количество ложноположительных ответов пренебрежимо мало.

5.3.2. Реализация фильтра Блума

Концептуально реализация фильтра Блума несложна. Сначала опишем, как он реализуется в одной системе, а потом перейдем к вопросу о распределенном использовании в Hadoop. Внутреннее представление фильтра Блума – битовый массив длиной m. Имеется k независимых хеш-функций, каждая из которых на входе принимает объект, а на выходе возвращает целое число от 0 до m–1. Это целое число интерпретируется как индекс битового массива. При добавлении элемента в фильтр Блума мы с помощью этих хеш-функций генерируем k индексов массива. Соответствующие k бит устанавливаются равными 1. На рис. 5.3 показано, что происходит при последовательном добавлении нескольких объектов (`x`, `y` и `z`) в фильтр Блума с тремя хеш-функциями. Отметим, что бит устанавливается равным 1 вне зависимости от его предыдущего состояния. Таким образом, количество единиц в битовом массиве может только расти.

Если требуется узнать, был ли некоторый объект ранее добавлен в фильтр Блума, то мы используем те же самые k хеш-функций, которые генерируют индексы массива. Далее мы проверяем, все ли k бит в этих позициях равны 1. Если это так, мы возвращаем `true`, утверждая

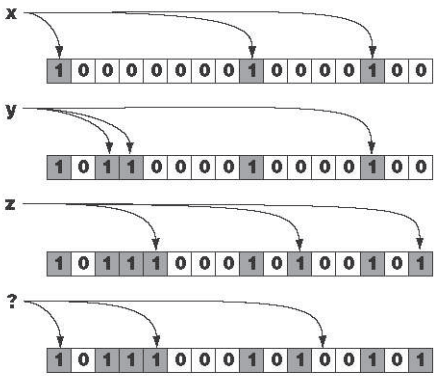


Рис. 5.3. Фильтр Блума – это битовый массив, который представляет множество с некоторой вероятностью ложноположительных ответов. Объектам (x, y, z) детерминированно сопоставляются позиции массива, и биты в этих позициях устанавливаются равными 1. Чтобы узнать, принадлежит ли объект множеству, нужно вычислить его хеши и проверить биты в соответствующих позициях.

тем самым, что объект присутствует в фильтре. В противном случае возвращаем `false`. Как видите, если объект действительно был добавлен в фильтр, то мы обязательно вернем `true`. Ложноотрицательных ответов (когда для объекта, принадлежащего множеству, возвращается `false`) быть не может. Однако все k бит, соответствующих запрошенному объекту, могут быть установлены в 1, хотя этот объект не добавлялся в фильтр. Такое может случиться, если добавление других объектов случайно привело к установке этих битов[3].

В реализации фильтра Блума на Java мы будем использовать для внутреннего представления класс `BitSet`. У нас будет функция `getHashIndexes(obj)`, которая принимает объект и возвращает массив целых чисел длиной k, содержащий индексы для `BitSet`. Основные функции фильтра Блума, `add()` и `contains()`, реализуются тривиально:

```
class BloomFilter<E> {

   private BitSet bf;

   public void add(E obj) {
      int[] indexes = getHashIndexes(obj);

      for (int index : indexes) {
```

[3] Элементарное введение в теорию фильтра Блума см. по адресу http://en.wikipedia.org/wiki/Bloom_filter.

```
            bf.set(index);
        }
    }

    public boolean contains(E obj) {
        int[] indexes = getHashIndexes(obj);

        for (int index : indexes) {
            if (bf.get(index) == false) {
                return false;
            }
        }
        return true;
    }

    protected int[] getHashIndexes(E obj) { ... }
}
```

Но реализация функции `getHashIndexes()` так, чтобы она вела себя как k истинно независимых хеш-функций, – дело непростое. Вместо этого в листинге 5.4 мы применили трюк – генерируем k приблизительно независимых индексов с равномерным распределением. Метод `getHashIndexes()` инициализирует генератор случайных чисел `Random` MD5-сверткой объекта, а затем получает от него k «случайных» чисел, интерпретируемых как индексы. Класс фильтра Блума много выиграл бы от более строгой реализации `getHashIndexes()`, но для демонстрационных целей достаточно и такого грубого приближения.

Чтобы создать фильтр Блума для объединения двух множеств, достаточно применить к их битовым массивам операцию поразрядного ИЛИ. Поскольку добавление объекта приводит к установке некоторых бит в 1, легко видеть, что таким образом мы действительно получим фильтр для объединения:

```
public void union(BloomFilter<E> other) {
    bf.or(other.bf);
}
```

Мы воспользуемся этим приемом для распределенного построения фильтров Блума. Каждый распределитель будет строить фильтр, исходя из собственной порции данных. Затем мы отправим эти фильтры одному редуктору, который вычислит их объединение и запишет конечный результат. Поскольку фильтры Блума, будучи выходными данными распределителя, подлежат тасованию, класс `BloomFilter` должен реализовывать интерфейс `Writable`, то есть содержать методы `write()` и `readFields()`. Нам требуется, чтобы эти методы выполняли преобразование из внутреннего представления (`Bit-`

Set) в массив байтов и обратно, чтобы можно было произвести сериализацию при выводе в DataOutput и десериализацию при вводе из DataInput. Полный код класса приведен в листинге 5.4.

Листинг 5.4.
Простая реализация фильтра Блума

```
class BloomFilter<E> implements Writable {

   private BitSet bf;
   private int bitArraySize = 100000000;
   private int numHashFunc = 6;

   public BloomFilter() {
      bf = new BitSet(bitArraySize);
   }

   public void add(E obj) {
      int[] indexes = getHashIndexes(obj);

      for (int index : indexes) {
         bf.set(index);
      }
   }

   public boolean contains(E obj) {
      int[] indexes = getHashIndexes(obj);

      for (int index : indexes) {
         if (bf.get(index) == false) {
            return false;
         }
      }
      return true;
   }

   public void union(BloomFilter<E> other) {
      bf.or(other.bf);
   }

   protected int[] getHashIndexes(E obj) {
      int[] indexes = new int[numHashFunc];

      long seed = 0;
      byte[] digest;
      try {
         MessageDigest md = MessageDigest.getInstance("MD5");
         md.update(obj.toString().getBytes());
         digest = md.digest();

         for (int i = 0; i < 6; i++) {
```

```
            seed = seed ^ (((long)digest[i] & 0xFF))<<(8*i);
         }
      } catch (NoSuchAlgorithmException e) {}

      Random gen = new Random(seed);

      for (int i = 0; i < numHashFunc; i++) {
         indexes[i] = gen.nextInt(bitArraySize);
      }
      return indexes;
   }

   public void write(DataOutput out) throws IOException {
      int byteArraySize = (int)(bitArraySize / 8);

      byte[] byteArray = new byte[byteArraySize];
      for (int i = 0; i < byteArraySize; i++) {
         byte nextElement = 0;
         for (int j = 0; j < 8; j++) {
            if (bf.get(8 * i + j)) {
               nextElement |= 1<<j;
            }
         }
         byteArray[i] = nextElement;
      }
      out.write(byteArray);
   }

   public void readFields(DataInput in) throws IOException {
      int byteArraySize = (int)(bitArraySize / 8);

      byte[] byteArray = new byte[byteArraySize];
      in.readFully(byteArray);
      for (int i = 0; i < byteArraySize; i++ ) {
         byte nextByte = byteArray[i];
         for (int j = 0; j < 8; j++) {
            if (((int)nextByte & (1<<j)) != 0) {
               bf.set(8 * i + j);
            }
         }
      }
   }
}
```

Далее мы напишем MapReduce-программу, которая создаст фильтр Блума с помощью Hadoop. Как уже было сказано, каждый распределитель будет создавать объект `BloomFilter` и добавлять в него ключ каждой записи из своей порции. (Мы используем именно ключ записи, поскольку имеем в виду применение к задаче о соединении данных.) Затем мы создадим объединение объектов `BloomFilter`, передав их одному-единственному редуктору.

Драйвер MapReduce-программы несложен. Наши распределители будут выводить пару ключ/значение, в которой значением является объект `BloomFilter`.

```
job.setOutputValueClass(BloomFilter.class);
```

Выходной ключ для разбиения не важен, так как редуктор у нас только один.

```
job.setNumReduceTasks(1);
```

Мы хотим, чтобы редуктор выводил результирующий объект `BloomFilter` в виде двоичного файла. Встроенные в Hadoop классы `OutputFormat` выводят либо текстовые файлы, либо пары ключ/значение. Поэтому мы в своем редукторе не станем пользоваться механизмами вывода, имеющимися в Hadoop, а будем записывать результат в файл самостоятельно.

```
job.setOutputFormat(NullOutputFormat.class);
```

Предупреждение. Вообще говоря, отклоняясь от средств ввода/вывода, предусмотренных в каркасе MapReduce, и начиная работать с файлами по-своему, вы ступаете на опасную дорожку. Теперь никто не дает гарантии, что ваши операции идемпотентны, и вы должны хорошо понимать, как они поведут себя в различных условиях. Например, если некоторые задачи перезапускаются, то может оказаться, что ваши файлы записаны лишь частично. Наш пример, пожалуй, безопасен, так как все файловые операции сосредоточены в одном месте – методе `close()` – и выполняются единственным редуктором. Но при более тщательной (параноидальной?) реализации каждую файловую операцию надо было бы проверять.

Напомним, что мы выбрали стратегию, при которой распределитель строит один фильтр Блума для всей порции и *по завершении порции* передает его редуктору. Но поскольку метод `map()` класса `MapClass` ничего не знает о том, какую именно запись порции он обрабатывает, приходится выводить объект `BloomFilter` в методе `close()`, так как только в этот момент мы знаем, что все записи порции прочитаны. Однако методу `map()` объект `OutputCollector` для сбора результатов распределителя передается, а методу `close()` – нет. Стандартное решение этой проблемы в Hadoop заключается в том, что объект `MapClass` сохраняет внутри себя ссылку на объект `OutputCollector`, переданный методу `map()`. Гарантируется, что `OutputCollector` может работать даже в методе `close()`. Код класса `MapClass` выглядит следующим образом:

```
public static class MapClass extends MapReduceBase
       implements Mapper<K1, V1, K2, V2> {

   OutputCollector<K2, V2> oc = null;
```

```
   public void map(K1 key, V1 value,
                   OutputCollector<K2,V2> output,
                   Reporter reporter) throws IOException {
      if (oc == null) oc = output;
      ...
   }

   public void close() throws IOException {
      oc.collect(k, v);
   }
}
```

Объекты `BloomFilter`, созданные всеми распределителями, посылаются одному редуктору. Метод `reduce()` класса `Reduce` объединяет все фильтры Блума.

```
while (values.hasNext()) {
   bf.union((BloomFilter<String>)values.next());
}
```

Как уже было сказано, мы хотим, чтобы результирующий объект `BloomFilter` выводился в виде файла специального формата, а не в одном из выходных форматов `OutputFormat`, встроенных в Hadoop. Чтобы отключить механизм вывода, мы задали в драйвере `OutputFormat` равным `NullOutputFormat`. Теперь же реализуем вывод файла в методе `close()`. Для этого необходимо знать выходной путь, заданный пользователем; его можно получить из свойства `mapred.output.dir` объекта `JobConf`. Но методу `close()` конфигурационный объект задачи не передается. Мы исправим это упущение точно так же, как поступили с объектом `OutputCollector` в распределителе, – сохраним ссылку на объект `JobConf` в классе `Reduce`, чтобы впоследствии ей можно было воспользоваться в методе `close()`. И затем воспользуемся имеющимися в Hadoop средствами файлового ввода/вывода для записи объекта `BloomFilter` в двоичный файл в системе HDFS. Полный код представлен в листинге 5.5.

Листинг 5.5.
MapReduce-программа для создания фильтра Блума

```
public class BloomFilterMR extends Configured implements Tool {

   public static class MapClass extends MapReduceBase
      implements Mapper<Text, Text, Text, BloomFilter<String>> {

   BloomFilter<String> bf = new BloomFilter<String>();
   OutputCollector<Text, BloomFilter<String>> oc = null;
```

```
    public void map(Text key, Text value,
                    OutputCollector<Text, BloomFilter<String>> output,
                    Reporter reporter) throws IOException {
       if (oc == null) oc = output;
       bf.add(key.toString());
    }

    public void close() throws IOException {
       oc.collect(new Text("testkey"), bf);
    }
}

public static class Reduce extends MapReduceBase
    implements Reducer<Text, BloomFilter<String>, Text, Text> {

    JobConf job = null;
    BloomFilter<String> bf = new BloomFilter<String>();

    public void configure(JobConf job) {
       this.job = job;
    }

    public void reduce(Text key, Iterator<BloomFilter<String>> values,
                       OutputCollector<Text, Text> output,
                       Reporter reporter) throws IOException {
       while (values.hasNext()) {
          bf.union((BloomFilter<String>)values.next());
       }
    }

    public void close() throws IOException {
       Path file = new Path(job.get("mapred.output.dir") +
                            "/bloomfilter");
       FSDataOutputStream out = file.getFileSystem(job).create(file);
       bf.write(out);
       out.close();
    }
}

    public int run(String[] args) throws Exception {
       Configuration conf = getConf();
       JobConf job = new JobConf(conf, BloomFilterMR.class);

       Path in = new Path(args[0]);
       Path out = new Path(args[1]);
       FileInputFormat.setInputPaths(job, in);
       FileOutputFormat.setOutputPath(job, out);

       job.setJobName("Bloom Filter");
       job.setMapperClass(MapClass.class);
```

```
        job.setReducerClass(Reduce.class);
        job.setNumReduceTasks(1);
        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(NullOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BloomFilter.class);
        job.set("key.value.separator.in.input.line", ",");

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new BloomFilterMR(),
                                 args);
        System.exit(res);
    }
}
```

5.3.3. Фильтр Блума в Hadoop версии 0.20+

В Hadoop версии 0.20 уже имеется класс фильтра Блума. Он служит для поддержки некоторых новых классов, появившихся в версии 0.20, и, вероятно, останется и в последующих версиях. Работает он очень похоже на класс `BloomFilter` из листинга 5.4, хотя реализация хеш-функций гораздо более тщательная. Будучи встроенным классом, он отлично подходит для реализации полусоединения в Hadoop. Но отделить его от каркаса и использовать как автономный класс нелегко. Если вы будете создавать фильтр Блума для приложений вне Hadoop, то встроенный в Hadoop класс `BloomFilter` может не подойти.

5.4. Упражнения

Следующие упражнения помогут проверить, насколько хорошо вы усвоили продвинутые приемы работы с каркасом MapReduce:

1. *Обнаружение аномалий.* Возьмите журнал веб-сервера. Напишите MapReduce-программу, которая суммирует количество посещений для каждого IP-адреса. Напишите еще одну MapReduce-программу, которая находит первые *K* IP-адресов,

для которых зарегистрировано максимальное количество посещений. Эти IP-адреса вполне могут принадлежать легальным прокси-серверам, организованным Интернет-провайдерами (они сообща используются многими пользователями), а могут быть адресами сайтов, сделанных специально для контекстной рекламы (MFA-сайтов), или мошеннических сайтов (если вы анализируете журнал веб-сервера рекламной сети). Сцепите эти две задачи MapReduce вместе, так чтобы их можно было запускать ежедневно в ходе рутинной процедуры.

2 *Фильтрация входных записей.* В обоих наборах данных о патентах (`cite75_99.txt` и `apat63_99.txt`) первая строка содержит метаданные (названия столбцов). До сих пор мы были вынуждены явно или неявно отфильтровывать эти строки в распределителях либо интерпретировать результаты, зная о том, какой эффект вносит строка с метаданными. Более правильное решение – удалить эту строку из входных данных и вспомнить о ней где-то в другом месте. Другой вариант – написать распределитель в виде препроцессора, который отфильтровывает записи, похожие на метаданные (например, записи, в начале которых не находится числовой номер патента). Напишите такой распределитель и с помощью классов `ChainMapper` и `ChainReducer` включите его в свои MapReduce-программы.

3 *Вычисление разности.* Взяв в качестве примера те же файлы Customers и Orders, которые мы использовали при рассмотрении пакета datajoin, измените код, так чтобы выводились заказчики, *не* присутствующие в источнике данных Orders. Быть может, файл Orders содержит только заказы, сделанные в последние *N* месяцев, тогда в результате этой операции мы узнаем, какие заказчики ничего не покупали в этот период. Компания могла бы вновь привлечь этих заказчиков, предложив им скидки или иные стимулы.

4 *Вычисление коэффициентов.* Коэффициенты нередко являются более подходящим аналитическим показателем, чем исходные числа. Пусть, например, имеются два набора данных: о сегодняшних и о вчерашних ценах на акции. Вас больше интересует не абсолютная цена, а ее рост. С помощью пакета datajoin напишите программу, которая принимает два источника данных и выводит отношения соответственных элементов.

5 *Умножение вектора на матрицу*. Посмотрите в своем любимом учебнике по линейной алгебре, как определяется умножение матриц. Напишите MapReduce-программу, которая вычисляет произведение вектора и матрицы. Для хранения вектора используйте распределенный кэш `DistributedCache`. Можете предполагать, что матрица записана в разреженном представлении.

6 *Пространственное соединение*. Это более сложная задача. Рассмотрим двумерное пространство, и пусть координаты x и y изменяются в диапазоне от -1 000 000 000 до +1 000 000 000. Имеется один файл с координатами *бук* и другой файл с координатами *бяк*. Каждая запись в том и другом файле содержит координаты x и y, разделенные запятой. Вот два примера строк:

145999.32455,888888880.001

834478899.2,5656.87660922

Напишите задачу MapReduce, которая находит все *буки* такие, что на расстоянии, меньшем 1, от них имеется хотя бы одна *бяка*. Подразумевается обычная евклидова метрика $\text{sqrt}[(x1-x2)^2 + (y1-y2)^2]$. Хотя *буки* и *бяки* разрежены в двумерном пространстве, соответствующие им файлы слишком велики и в память не помещаются. Использовать для операции пространственного соединения распределенный кэш невозможно.

Подсказка. Пакет datajoin в его существующем виде также плохо подходит для этой задачи, но ее можно решить, написав собственный распределитель и редуктор, поток данных в которых устроен примерно так же, как в datajoin.

Подсказка №2. Во всех рассмотренных до сих пор MapReduce-программах ключи только извлекались и передавались, тогда как со значениями производились различные вычисления. Подумайте о том, чтобы вычислять ключ для выходных данных распределителя.

7 *Пространственное соединение в сочетании с фильтром Блума*. Решив предыдущую задачу, подумайте, как можно ускорить соединение, применив фильтр Блума. Предположите, что *бяк* гораздо меньше, чем *бук*, но все равно слишком много для размещения в памяти.

5.5. Резюме

Часто простую MapReduce-программу можно написать в виде задачи, работающей с одним набором данных. В более сложных случаях можно написать программу в виде нескольких задач или в виде задачи, работающей с несколькими наборами данных. В Hadoop имеются разные способы координации нескольких задач, в том числе последовательное сцепление и исполнение в соответствии с заранее заданными зависимостями. Для распространенного частного случая сцепления задач, содержащих только распределители, с полной задачей MapReduce, в Hadoop есть специальные оптимизированные классы.

Соединение – канонический пример обработки данных из нескольких источников. В состав Hadoop входит универсальный пакет datajoin для вычисления произвольных соединений, но за его общность приходится расплачиваться эффективностью. Есть и другие методы соединения, позволяющие выполнить соединение быстрее, если размеры источников данных существенно различаются, как то часто бывает. В одном из этих методов используется фильтр Блума – структура данных, полезная для многих задач обработки данных.

Имеющихся у вас сейчас знаний о программировании в парадигме MapReduce достаточно, чтобы начать писать собственные программы. Но любой программист знает, что программирование не сводится к написанию кода. Существуют различные приемы и процессы – от разработки и развертывания до тестирования и отладки. Сама природа парадигмы MapReduce и распределенных вычислений вносит в эти процессы дополнительные сложности и нюансы. Мы рассмотрим их в следующей главе.

5.6. Дополнительные ресурсы

http://portal.acm.org/citation.cfm?doid=1247480.1247602. Недостаточная поддержка соединения наборов данных в MapReduce – хорошо известная проблема. Существует целый ряд средств, дополняющих Hadoop (например, Pig, Hive и CloudBase), в которых соединение данных реализовано как полноценная операция. Формальное освещение этого вопроса имеется в статье Хунь Чи Яня с соавторами «Map-reduce-merge: simplified relational data processing on large clusters», где предложена модифицированная форма MapReduce с допол-

нительным шагом «слияния», в которой соединение данных поддерживается естественным образом.

http://umiacs.umd.edu/~jimmylin/publications/Lin_etal_TR2009.pdf. В разделе 5.2.2 описано использование распределенного кэша для предоставления операциям вспомогательных данных. Ограничение этой техники состоит в том, что вспомогательные данные реплицируются на каждый узел TaskTracker и должны помещаться в памяти. Джимми Лин (Jimmy Lin) с коллегами исследовали применение программы memcached, распределенной системы кэширования объектов в памяти, для предоставления глобального доступа к вспомогательным данным. Их опыт обобщен в статье «Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework».

ГЛАВА 6.
Практическое программирование

В этой главе:

- Специфические рекомендации по разработке программ для Hadoop.
- Отладка программ в автономном, псевдораспределенном и полностью распределенном режимах.
- Контрольные проверки и регрессионное тестирование программ.
- Протоколирование и мониторинг.
- Оптимизация производительности.

Познакомив вас с различными приемами программирования в MapReduce, мы в этой главе вернемся назад и рассмотрим вопросы практического программирования.

Программирование в Hadoop имеет два существенных отличия от традиционного. Во-первых, программы для Hadoop занимаются в основном обработкой данных. Во-вторых, эти программы исполняются на распределенном наборе компьютеров. Из-за этих отличий по-другому выглядят некоторые аспекты процессов разработки и отладки, о чем мы будем говорить в разделах 6.1 и 6.2.

Методы оптимизации производительности всегда зависят от программной платформы, и Hadoop здесь не исключение. Инструменты и подходы к оптимизации программ для Hadoop мы рассмотрим в разделе 6.3.

Начнем с приемов разработки, применимых к Hadoop. Надо полагать, вы уже знакомы со стандартными методами программотехники

в Java. Мы же сосредоточимся на разработке программ, ориентированных на обработку данных, в рамках каркаса Hadoop.

6.1. Разработка MapReduce-программ

В главе 2 рассматривались три режима работы Hadoop: локальный (автономный), псевдораспределенный и полностью распределенный. Грубо говоря, они соответствуют конфигурациям для разработки, промежуточного развертывания и промышленной эксплуатации. В процессе разработки вы неизбежно столкнетесь со всеми тремя режимами. Поэтому должна быть возможность легко переключаться между ними. На практике даже несколько полностью распределенных кластеров не редкость. Так, в больших организациях иногда создают специальный кластер для разработки, чтобы дополнительно «погонять» MapReduce-программы перед тем, как запускать их на реальном производственном кластере. Несколько кластеров можно организовать и для различных работ. Скажем, внутри компании настраивается кластер для выполнения небольших и средних задач, а в более экономичном облаке – кластер для запуска больших, но редко выполняемых задач.

В разделе 2.3 говорилось, что можно иметь разные версии файла `hadoop-site.xml` для различных конфигураций, а переключаться между ними путем создания символической ссылки на тот файл, с которым вы работаете в данный момент. Нужный конфигурационный файл можно также указать с помощью флага `-conf` в командной строке для запуска Hadoop. Например, команда

```
bin/hadoop fs -conf conf.cluster/hadoop-site.xml -lsr
```

выведет список всех файлов в вашем полностью распределенном кластере, хотя в этот момент вы, возможно, работаете в другом режиме или в другом кластере (в предположении, что файл `conf.cluster/hadoop-site.xml` содержит конфигурацию полностью распределенного кластера).

Перед тем как приступить к тестированию Hadoop-программы, необходимо подготовить данные для рабочей конфигурации. В разделе 3.1 описаны различные способы поместить данные в файловую систему HDFS и извлечь их оттуда. Для локального и псевдораспределенного режима понадобится небольшое подмножество всех данных. В разделе 4.4 была представлена Streaming-программа

(`RandomSample.py`), которая случайным образом отбирает заданный процент записей из набора данных, находящегося в HDFS. Поскольку это скрипт, написанный на Python, его можно также использовать для выборки данных из локального файла с помощью конвейера Unix. Так, команда

```
cat datafile | RandomSample.py 10
```

выберет 10 процентов записей из файла `datafile`.

Итак, вы настроили все конфигурации и знаете, как подготовить данные для каждой из них. Теперь поговорим о том, как разрабатывать и отлаживать программы локальном и псевдораспределенном режиме. Приемы, применяемые для более сложных режимов, основаны на тех, что используются в более простых. Вопрос об отладке в полностью распределенном режиме мы отложим до следующего раздела.

6.1.1. Локальный режим

В локальном режиме Hadoop исполняется целиком в одной виртуальной машине Java (JVM) и работает только с локальной файловой системой (без HDFS). Это позволяет использовать весь набор знакомых инструментов разработки для Java, в том числе отладчик. Поскольку все файлы находятся в локальной файловой системе, к входным и выходным данным можно применять команды Unix и простые скрипты. С другой стороны, при операциях с файлами в HDFS вы ограничены командами, включенными в состав Hadoop. Например, для подсчета количества записей в выходном файле можно воспользоваться командой `wc -l`, если файл находится в локальной файловой системе. Если же он размещен в HDFS, то придется либо написать MapReduce-программу, либо скопировать файл в локальную файловую систему, а потом уже применять к нему команды Unix. Как будет ясно из дальнейшего, наличие простого доступа к входным и выходным файлам весьма важно для разработки в локальном режиме.

> **Примечание.** Локальный режим вполне соответствует модели программирования MapReduce, которая реализована в Hadoop, но поддерживает не все возможности. В частности, не поддерживается распределенный кэш и допускается не более одного редуктора.

Программа, работающая в локальном режиме, выводит все сообщения – информационные и об ошибках – на консоль. Кроме того, в конце работы она уведомляет о том, сколько всего данных было об-

работано. Например, запустив нашу модельную задачу MapReduce (`MyJob.java`), которая инвертирует данные о патентах, мы получим весьма объемную распечатку; на рис. 6.1 приведен снимок экрана в середине работы. В конце Hadoop печатает значения различных внутренних счетчиков: число записей и байтов, прошедших через различные этапы MapReduce:

```
09/05/27 03:34:37 INFO mapred.TaskRunner: Task
➥ attempt_local_0001_r_000000_0' done.
09/05/27 03:34:37 INFO mapred.TaskRunner: Saved output of task
➥ attempt_local_0001_r_000000_0' to
➥ file:/Users/chuck/Projects/Hadoop/hadoop-0.18.1/output/test
09/05/27 03:34:37 INFO mapred.LocalJobRunner: reduce > reduce
09/05/27 03:34:37 INFO mapred.JobClient: Job complete: job_local_0001
09/05/27 03:34:37 INFO mapred.JobClient: Counters: 11
09/05/27 03:34:37 INFO mapred.JobClient:  Map-Reduce Framework
09/05/27 03:34:37 INFO mapred.JobClient:    Map output records=16522439
09/05/27 03:34:37 INFO mapred.JobClient:    Reduce input records=33044878
09/05/27 03:34:37 INFO mapred.JobClient:    Map output bytes=264075431
09/05/27 03:34:37 INFO mapred.JobClient:    Map input records=16522439
09/05/27 03:34:37 INFO mapred.JobClient:    Combine output records=0
09/05/27 03:34:37 INFO mapred.JobClient:    Map input bytes=264075431
09/05/27 03:34:37 INFO mapred.JobClient:    Combine input records=0
09/05/27 03:34:37 INFO mapred.JobClient:    Reduce input groups=6517968
09/05/27 03:34:37 INFO mapred.JobClient:    Reduce output records=6517968
09/05/27 03:34:37 INFO mapred.JobClient:  File Systems
09/05/27 03:34:37 INFO mapred.JobClient:    Local bytes written=4246405780
09/05/27 03:34:37 INFO mapred.JobClient:    Local bytes read=4276658154
```

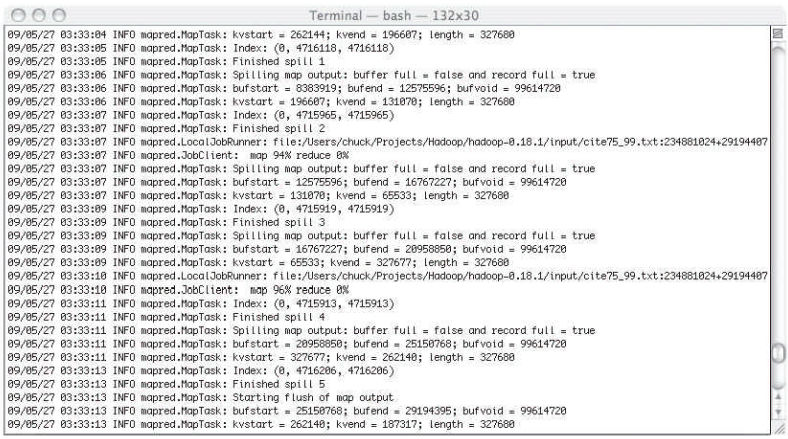


Рис. 6.1. При запуске Hadoop-программы в локальном режиме все сообщения выводятся на консоль.

Входные и выходные файлы задачи MapReduce находятся в локальной файловой системе. Мы можем применять к ним стандартные команды Unix, например `wc -l` или `head`. Поскольку на этапе разработки мы сознательно используем небольшие наборы данных, их можно даже загрузить в текстовый редактор или электронную таблицу и воспользоваться средствами этих приложений для проверки правильности работы нашей программы.

Контрольные проверки

В большинстве MapReduce-программ встречаются какие-то арифметические операции, в частности подсчет, а ошибки (и особенно опечатки) при программировании таких операций не привлекают к себе особого внимания – не возбуждают исключений и не выдают грозных сообщений. Математика может быть запрограммирована неверно, но при этом технически программа будет «корректной» и отработает без ошибок, только вот ее конечный результат бесполезен. Простого способа обнаружить ошибки в арифметике не существует, но некоторые контрольные проверки помогут приблизиться к этой цели. На верхнем уровне можно посмотреть, соответствуют ли ожиданиям счетчики, средние и максимальные значения различных метрик. На нижнем уровне можно проверить правильность генерации конкретной выходной записи. Например, при создании инвертированного списка ссылок первые несколько строк выходного файла выглядели так:

```
"CITED" "CITING"
1       3964859,4647229
10000   4539112
100000  5031388
1000006 4714284
1000007 4766693
1000011 5033339
1000017 3908629
1000026 4043055
1000033 4190903,4975983
```

Наша задача пришла к выводу, что патент с номером 1 цитировался дважды: в патентах 3964859 и 4647229. Это можно проверить, запустив для входных данных утилиту `grep`, чтобы найти все записи, в которых цитируется патент 1:

```
grep ",1$" input/cite75_99.txt
```

Действительно таких записей 2. Чтобы окончательно увериться в правильности логики и математических вычислений в своей прог-

рамме, можете проверить еще несколько записей[1].

В этом инвертированном списке ссылок колет глаза первая строка, вообще не являющаяся данными:

```
"CITED" "CITING"
```

Она появилась из первой строки входных данных, которая служит для описания столбцов. Давайте включим в распределитель код, отфильтровывающий строки с нечисловыми ключами и значениями, и попутно продемонстрируем технику регрессионного тестирования.

Регрессионное тестирование

Наш ориентированный на данные подход к регрессионному тестированию – это различные вариации на тему сравнения выходных файлов, получающихся до и после изменения кода. В рассматриваемом случае из выходного файла должна исчезнуть ровно одна строка. Чтобы проверить, так ли это, сначала сохраним результат, генерируемый текущей версией задачи. В локальном режиме может быть не более одного редуктора, поэтому на выходе задачи получится только один файл, который мы назовем `job_1_output`.

Для целей регрессионного тестирования полезно также сохранить выходные данные, полученные на этапе распределения. Это поможет понять, на каком этапе – распределения или редукции, – произошла ошибка. Чтобы сохранить эти данные, нужно запустить задачу MapReduce без редукторов, для чего достаточно задать флаг `-D mapred.reduce.tasks=0`. Задача без редукторов порождает несколько выходных файлов, так как каждый распределитель пишет в свой собственный файл. Скопируем их в каталог `job_1_intermediate`.

Сохранив выходные файлы, внесем нужные изменения в метод `map()` класса `MapClass`. Сам код тривиален, интерес представляет только его тестирование.

```
public void map(Text key, Text value,
                OutputCollector<Text, Text> output,
                Reporter reporter) throws IOException {
   try
   {
      if (Integer.parseInt(key.toString()) > 0 &&
          Integer.parseInt(value.toString()) > 0)
```

[1] В данном случае могут возникнуть сомнения, *действительно* ли на патент 1 есть ссылки из двух других патентов. Число 1 выглядит подозрительно, оно выпадает из диапазона номеров цитируемых патентов. Возможно, в исходных данных имеются ошибки. Если мы хотим это проверить, то придется заглянуть в сами патенты. Так или иначе, обеспечение качества данных – задача важная, но находящаяся за рамками нашего обсуждения Hadoop.

```
        {
            output.collect(value, key);
        }
    } catch (NumberFormatException e) { }
}
```

Откомпилируем и выполним новую программу с теми же входными данными. Сначала запустим ее без редуктора и сравним промежуточные данные. Поскольку мы изменили только распределитель, то любая ошибка должна проявиться в различии между старыми и новыми промежуточными данными.

```
diff output/job_1_intermediate/ output/test/
```

Утилита diff печатает следующее:

```
Binary files output/job_1_intermediate/.part-00000.crc and
➥ output/test/.part-00000.crc differ
diff output/job_1_intermediate/part-00000 output/test/part-00000
1d0
< "CITED" "CITING"
```

Мы обнаружили различия в двоичном файле `.part-00000.crc`. Это внутренний файл, в котором хранится контрольная сумма файла `part-00000` (он нужен для HDFS). Изменение контрольной суммы означает, что изменился и файл `part-00000`, и ниже diff сообщает о том, в каком месте имеется различие. В новом промежуточном файле в каталоге `output/test` отсутствуют закавыченные описатели полей. Важнее, однако, то, что никаких других различий нет. Пока все хорошо. Можно ожидать, что при запуске задачи с одним редуктором окончательный результат тоже будет отличаться всего в одной строке.

Но оказывается, что это не так. Если запустить задачу с одним редуктором и сравнить файл `job_1_output` с первоначальным, то обнаружится много различий. И как вы это объясните? Разберемся, изучив первые несколько строк, напечатанных diff.

```
$ diff output/job_1_output output/test/part-00000 | head -n 15
1,2c1
< "CITED"       "CITING"
< 1     3964859,4647229
---
> 1     4647229,3964859
19c18
< 1000067       5312208,4944640,5071294
---
> 1000067       4944640,5071294,5312208
22,23c21,22
< 1000076       4867716,5845593
```

```
< 1000083       5566726,5322091
---
> 1000076       5845593,4867716
> 1000083       5322091,5566726
```

Строка с описателями полей («CITED» и «CITING») исчезла, как мы и ожидали. В остальных же различиях видна отчетливая закономерность. В методе `reduce()` мы конкатенировали список значений для каждого ключа в том порядке, в котором Hadoop их передавал. Но Hadoop не дает никаких гарантий относительно порядка следования этих значений. Как видите, исключение одной строки из промежуточных данных повлияло на порядок следования значений для многих ключей. Но, как мы знаем, правильность этой задачи не зависит от упорядочения, поэтому такие различия можно игнорировать. При регрессионном тестировании часто возникают ложные тревоги, имейте это в виду.

Мы ратовали за использование на стадии разработки выборки из реального набора данных, потому что она точнее отражает структуру и свойства данных, которые встретятся на стадии эксплуатации. Для регрессионного тестирования мы взяли именно такую выборку, но можно также вручную создать отдельные входные данные для тестирования граничных случаев, не типичных для реальных данных. Например, можно включить пустые значения или лишние знаки табуляции или другие необычные записи. Назначение такого набора данных – убедиться в том, что по мере развития программа корректно обрабатывает граничные случаи. В нем может быть всего несколько десятков записей. Выходной файл можно просто просмотреть глазами и убедиться, что программа по-прежнему работает, как задумано.

Используйте long вместо int

Большинство программистов, пишущих на Java, инстинктивно используют для представления целых чисел тип `int` (или `Integer` или `IntWritable`). В Java тип `int` позволяет хранить целые числа от $2^{31}-1$ до -2^{31} (то есть от 2 147 483 647 до –2 147 483 ,648). Для обычных приложений вполне достаточно, и программисты редко задумываются об этом. Но при обработке массивных данных в Hadoop счетчики нередко выходят за эти пределы. На стадии разработки вы вряд ли столкнетесь с этой проблемой, потому что сознательно взят небольшой набор данных. Не исключено, что и для текущих производственных данных проблема не возникнет, но по мере роста бизнеса растет и объем данных. И в какой-то момент некая переменная может выйти за границы диапазона `int`, что приведет к арифметичес-

ким ошибкам. Если количество обрабатываемых документов исчисляется миллионами, то счетчик слов вряд ли превысит 2 миллиарда[2], так что типа `int` достаточно. Но когда число документов достигает десятков и сотен миллионов, счетчики таких часто встречающихся слов, как *the*, вполне могут выйти за границу диапазона, представимого типом `int`. Чем пассивно ждать, когда такая ошибка проявится в производственной системе, где отлаживать ее будет гораздо труднее, а исправлять дороже, не лучше ли прямо сейчас просмотреть весь свой код и подумать, не надо ли описать те или иные переменные как `long` или `LongWritable` в расчете на будущий рост[3].

6.1.2. Псевдораспределенный режим

В локальном режиме полностью отсутствует *распределенность*, характерная для производственного кластера Hadoop. При запуске программы в локальном режиме многие ошибки могут остаться незамеченными. Поэтому в Hadoop имеется псевдораспределенный режим, располагающий всей функциональностью и «узлами» производственного кластера – Name Node, SecondaryNameNode, DataNode, JobTracker и TaskTracker, – каждый из которых работает в отдельной JVM. Все программные компоненты распределены, а различия между псевдораспределенным и производственным режимом заключаются лишь в масштабе системы и оборудования. Задействована только одна физическая машина – ваш локальный компьютер. Перед тем как развертывать задачи в производственном кластере, необходимо убедиться, что они правильно работают в псевдораспределенном режиме.

В главе 2 были описаны конфигурация псевдораспределенного режима и команды для его запуска. Все демоны запускаются на одном компьютере, чтобы он работал подобно кластеру. Взаимодействие с этим компьютером осуществляется так, будто это отдельный кластер Hadoop. Данные помещаются в организованную на нем же файловую систему HDFS. Задачи передаются для исполнения кластеру, а не запускаются в одном и том же пользовательском окружении. При этом вы можете следить за их исполнением «удаленно», с помощью фай-

[2] Это, конечно, зависит от размера и содержимого документов.

[3] Проблема выхода за границы числового диапазона возникает не только в Hadoop. Вспомните знаменитую проблему 2000 года, связанную с тем, что в старых программах для представления года использовалось только две цифры. А сравнительно недавно почти все Интернет-компании, испытавшие экспоненциальный рост (например, Facebook, Twitter и RockYou), вынуждены были переработать свои системы, расширив диапазон идентификаторов пользователей или документов, который оказался больше, чем ожидалось.

лов журналов и веб-интерфейса. То есть применяются те же инструменты, что для мониторинга производственного кластера.

Протоколирование

Давайте запустим в псевдораспределенном режиме ту же задачу, что ранее запускали в локальном режиме. Поместите входной файл в HDFS командой `hadoop fs`. Отправьте задачу на исполнение той же командой `hadoop jar`, что в локальном режиме. Вы сразу же заметите, что с консоли исчез нескончаемый поток сообщений. Остались только сообщения о том, как продвигается работа на этапах распределения и редукции. Кроме того, в самом конце выводятся суммарные счетчики, как и в локальном режиме. Картина в целом показана на рис. 6.2.

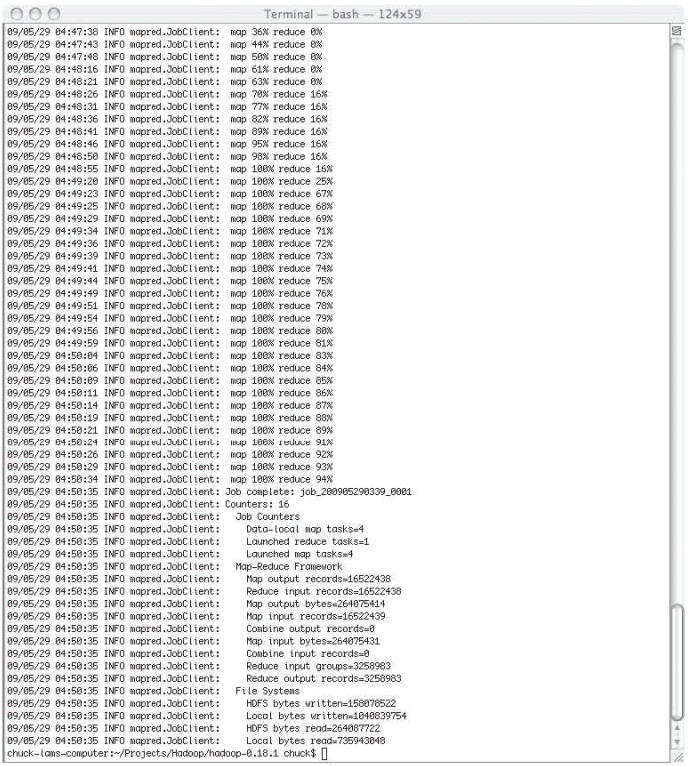


Рис. 6.2. В псевдораспределенном режиме на консоль выводятся только сообщения о проценте выполнения и суммарные счетчики.

Hadoop не перестал выводить отладочные сообщения, их количество даже существенно выросло. Просто они теперь выводятся не на экран, а в файлы журналов.

Эти файлы находятся в каталоге `/logs`. Каждая служба (NameNode, JobTracker и т. д.) создает свой журнал. Какой именно службе принадлежит файл журнала, можно определить по его имени. Hadoop ротирует журналы ежедневно. Последний файл имеет расширение `.log`. В конец имени более старых файлов добавляется дата их создания. По умолчанию Hadoop не удаляет старые журналы автоматически. Поэтому вы должны заранее продумать стратегию архивации и удаления, чтобы журналы не занимали слишком много места.

Журналы демонов NameNode, SecondaryNameNode, DataNode и JobTracker используются для отладки соответствующих служб. В псевдораспределенном режиме они не особенно важны. Администратор производственного кластера может заглядывать в них, когда нужно разобраться с проблемой, возникшей на соответствующем узле. Однако вас, как программиста, должен интересовать журнал TaskTracker, поскольку в нем протоколируются исключения.

Ваша MapReduce-программа может выводить собственные сообщения на `STDOUT` и `STDERR` (в Java это `System.out` и `System.err`). Hadoop записывает их соответственно в файлы с именами `stdout` и `stderr`. При каждой попытке запуска задания создается новый файл (попыток может быть несколько, если первая завершилась неудачно). Эти файлы пользовательских журналов находятся в подкаталоге `/logs/userlogs`.

Помимо вывода на `STDOUT` и `STDERR`, ваша программа может посылать сообщения о своем состоянии с помощью метода `setStatus()` объекта `Reporter`, который передается методам `map()` и `reduce()`. (Для Streaming-программ информация о состоянии обновляется путем вывода строки вида `reporter:status:`*`message`* на `STDERR`.) Это полезно для мониторинга долго работающих задач. Сообщения о состоянии показываются в веб-интерфейсе демона JobTracker, который описывается ниже.

Веб-интерфейс JobTracker

По определению события распределенной программы происходят в разных местах. Поэтому следить за ними труднее. Система выглядит, как черный ящик, и нам необходимы специализированные средства мониторинга, позволяющие наблюдать за ее состояниями. JobTracker предоставляет веб-интерфейс для мониторинга хода ис-

полнения и различных состояний задач. В конфигурации по умолчанию веб-интерфейс доступен по адресу

```
http://localhost:50030/jobtracker.jsp
```

Перейдя по этому адресу в браузере, вы увидите начальную страницу инструмента администрирования псевдораспределенного кластера[4]. На ней представлена сводная информация о кластере Hadoop, а также списки исполняющихся, нормально и аварийно завершенных задач (рис. 6.3).

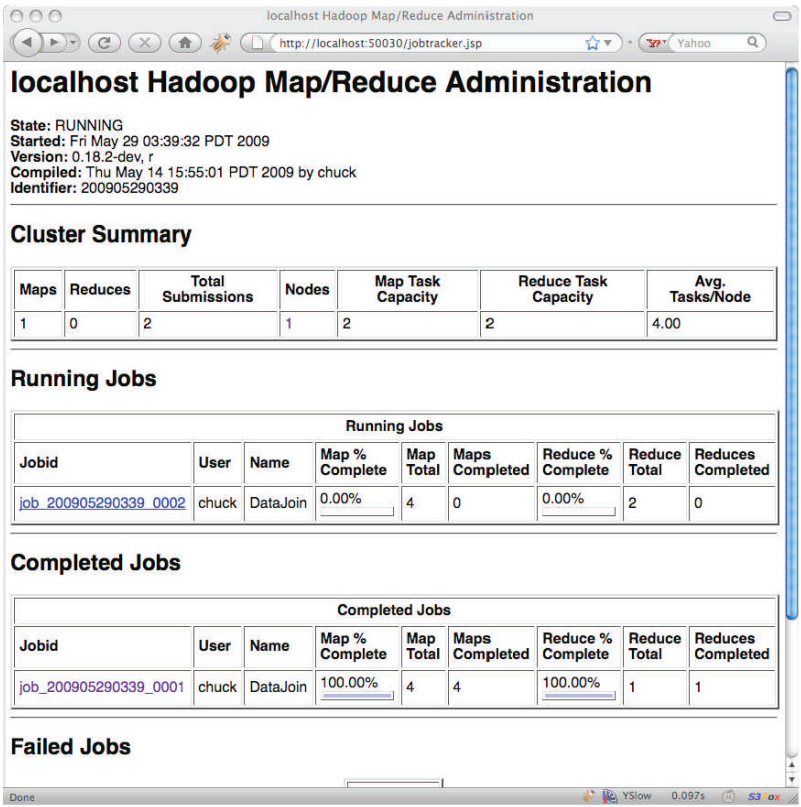


Рис. 6.3. Главная страница веб-интерфейса JobTracker

Hadoop использует для различения задач их внутренние идентификаторы. Идентификатор задачи – это строка с префиксом `job_`, за

[4] В полностью распределенном режиме замените «localhost» доменным именем главного узла JobTracker.

которым следует идентификатор кластера (время запуска кластера) и автоматически инкрементируемый номер задачи. В веб-интерфейсе для каждой задачи указывается имя пользователя и имя задачи. В псевдораспределенном режиме опознать задачу, с которой вы сейчас работаете, относительно легко, поскольку в каждый момент исполняется только одна задача. В многопользовательской производственной среде выделить свои задачи можно по имени пользователя в Hadoop и по имени текущей задачи. Имя задачи задается в программе путем вызова метода `setJobName()` объекта `JobConf`. Имя Streaming-задачи задает с помощью конфигурационного свойства, показанного в табл. 6.1.

Таблица 6.1. Конфигурационное свойство для задания имени задачи

Свойство	Описание
`mapred.job.name`	Строковое свойство для задания имени задачи

На начальной странице для каждой задачи показан процент выполнения этапа распределения, количество операций map, в том числе завершенных. Такие же показатели выводятся для этапа редукции. Таким образом, вы имеете общее представление о ходе выполнения своей задачи. Чтобы получить о задаче более подробную информацию, щелкните по ее идентификатору; в результате вы попадете на страницу администрирования задачи (рис. 6.4).

На странице задачи показаны объемы различных операций ввода/вывода, выполненных от имени данной задачи. Периодически страница обновляется автоматически, но ее можно обновить и вручную. Для ознакомления с различными аспектами задачи на странице имеются многочисленные ссылки. Например, перейдя по ссылке *map*, вы попадете на страницу со списком всех заданий map для данной задачи (рис. 6.5).

У каждого задания имеется идентификатор. Строится он так: берется идентификатор задачи, которой принадлежит задание, и префикс `job_` заменяется на `task_`; затем добавляется суффикс `_m` для задания `map` или `_r` для задания `reduce`; в конец дописывается автоматически инкрементируемый номер, причем нумерация независима для каждой группы. В веб-интерфейсе TaskTracker для каждого задания указано состояние, которое можно установить из программы с помощью описанного выше метода `setStatus()`.

Щелкнув по идентификатору задания, вы попадете на страницу с описанием *попыток* его запуска. Hadoop несколько раз пытается повторно запустить аварийно завершающиеся задания перед тем, как снять задачу целиком.

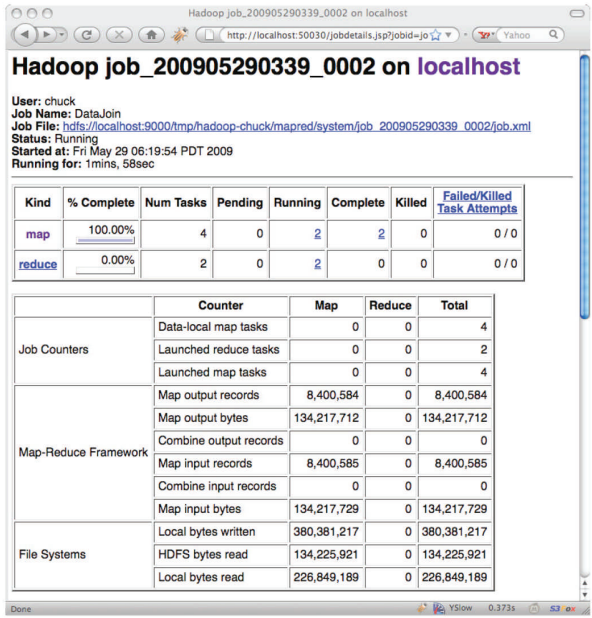


Рис. 6.4. Страница администрирования задачи в веб-интерфейсе JobTracker.

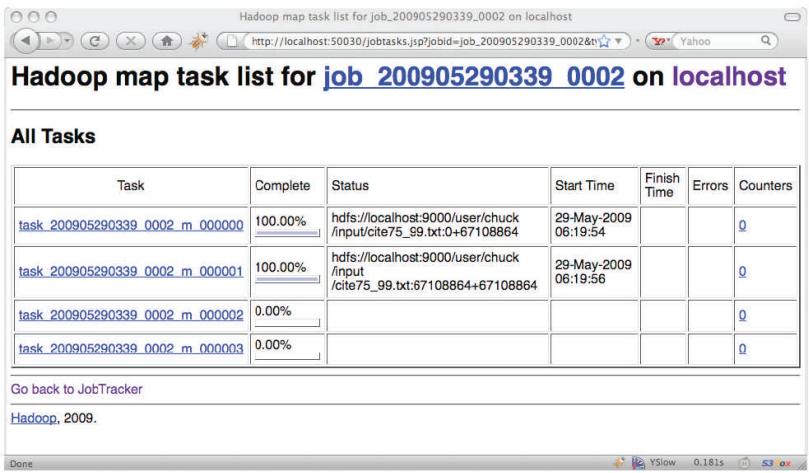


Рис. 6.5. Список задач в веб-интерфейсе TaskTracker. Представлены все задания map для одной задачи. Каждое задание может обновлять информацию о своем состоянии.

В веб-интерфейсах JobTracker и TaskTracker имеется много других ссылок и метрик. Большая их часть в пояснениях не нуждается.

Снятие задач

К сожалению, бывает, что запущенная задача начинает делать что-то не то, но аварийно не завершается. Она может работать очень долго и даже зациклиться. В распределенном (и в псевдораспределенном) режиме задачу можно снять вручную командой

```
bin/hadoop job -kill job_id
```

где *job_id* – идентификатор задачи, который показывается в веб-интерфейсе JobTracker.

6.2. Мониторинг и отладка в производственном кластере

Добившись успеха при запуске задачи в псевдораспределенном кластере, вы можете переходить к запуску ее в производственном кластере с реальными данными. К производственному кластеру применимы все рассмотренные выше приемы разработки и отладки, хотя порядок их использования может слегка отличаться. В вашем кластере должен быть веб-интерфейс JobTracker, но его доменное имя уже не `localhost`. Теперь это адрес узла JobTracker в кластере. Номер порта по-прежнему 50030, если только вы не сконфигурировали его по-другому.

В псевдораспределенном режиме, когда есть только один узел, все файлы журналов находятся в единственном каталоге `/logs`, в который можно зайти локально. В полностью распределенном кластере на каждом узле имеется собственный каталог `/logs`, где хранятся журналы этого узла. Диагностировать проблемы на некотором узле можно только по его журналам.

В дополнение к ранее упомянутым приемам разработки и тестирования, в вашем распоряжении имеются средства мониторинга и отладки, более подходящие для производственного кластера с реальными данными. Их мы и рассмотрим в этом разделе.

6.2.1. Счетчики

Задачу Hadoop можно оснастить *счетчиками* для профилирования ее работы. В своей программе вы определяете различные счетчики

и увеличиваете их значения в ответ на те или иные события. Hadoop автоматически суммирует одинаковые счетчики из разных заданий (в одной и той же задаче), так что получается общая картина задачи. Значения определенных вами счетчиков отображаются в веб-интерфейсе JobTracker наряду с внутренними счетчиками Hadoop.

Каноническое применение счетчиков – подсчет входных записей разных типов, особенно «плохих» записей. Вспомните пример подсчета среднего числа пунктов формулы изобретения в одном патенте для каждой страны (раздел 4.4). Мы знаем, что число пунктов во многих записях отсутствует. Наша программа такие записи пропускает, и было бы полезно знать, сколько именно записей пропущено. И дело тут не только в удовлетворении праздного любопытства; такая информация позволяет лучше понять работу программы и оценить, нет ли «расхождения с реальностью».

Для работы со счетчиками предназначен метод `Reporter.incrCounter()`. Объект `Reporter` передается методам `map()` и `reduce()`. При вызове `incrCounter()` мы указываем имя счетчика и величину, на которую его следует увеличить. Для каждого события создается счетчик с уникальным именем. Если при обращении к `incrCounter()` указан счетчик с ранее не встречавшимся именем, он создается и инициализируется переданным значением инкремента.

Есть два варианта метода `Reporter.incrCounter()`, отличающиеся способом задания имени счетчика:

```
public void incrCounter(String group, String counter, long amount)
public void incrCounter(Enum key, long amount)
```

Первый вариант более общий, так как позволяет задавать имя счетчика динамически во время выполнения. Комбинация строк `group` и `counter` однозначно определяет счетчик. При выводе накопленной информации (в веб-интерфейсе или в текстовом виде по завершении задачи) счетчики, принадлежащие одной группе, располагаются рядом.

Во втором варианте для задания имен счетчиков используется перечисление `enum`, а это значит, что счетчики должны быть определены на этапе компиляции. Зато при этом проверяются типы. Имя перечисления выступает в роли имени группы, а его значение – в роли имени счетчика.

В листинге 6.1 класс `MapClass` из листинга 4.12 дополнен счетчиками для отслеживания количества отсутствующих и «закавы-

ченных» значений (закавыченным может быть только значение в первой строке, содержащее описание столбца). В перечислении `enum ClaimsCounters` определены значения MISSING и QUOTED. Программа увеличивает счетчики, встречая соответствующие записи.

Листинг 6.1.
Класс MapClass со счетчиками для подсчета отсутствующих значений

```
public static class MapClass extends MapReduceBase
   implements Mapper<LongWritable, Text, Text, Text> {

   static enum ClaimsCounters { MISSING, QUOTED };

   public void map(LongWritable key, Text value,
                   OutputCollector<Text, Text> output,
                   Reporter reporter) throws IOException {

      String fields[] = value.toString().split(",", -20);
      String country = fields[4];
      String numClaims = fields[8];

      if (numClaims.length() == 0) {
         reporter.incrCounter(ClaimsCounters.MISSING, 1);
      } else if (numClaims.startsWith("\"")) {
         reporter.incrCounter(ClaimsCounters.QUOTED, 1);
      } else {
        output.collect(new Text(country), new Text(numClaims + ",1"));
      }
   }
}
```

После прогона программы в веб-интерфейсе JobTracker определенные нами счетчики будут показаны наряду с внутренними счетчиками Hadoop (рис. 6.6).

Как видите, в качестве имени группы фигурирует полное имя перечисления `enum` (знак `$` разделяет имена классов). Для самих счетчиков показаны определенные нами имена MISSING и QUOTED. Как и следовало ожидать, программа увеличила счетчик QUOTED только один раз. А счетчик MISSING – 939 867 раз. Действительно ли в наборе данных так много строк с отсутствующим количеством пунктов в формуле изобретения? Автор набора данных утверждает, что этой информации нет только для патентов, выданных до 1975 года. Глядя на рис. 4.3, можно предположить, что треть всех патентов, представленных в этом наборе, выдана до 1975. Из рис. 6.6 видно, что всего входных записей чуть больше 2,9 миллионов. Числа согласуются, так что у нас появляется дополнительная уверенность в правильности обработки.

	Counter	Map	Reduce	Total
Job Counters	Data-local map tasks	0	0	4
	Launched reduce tasks	0	0	2
	Launched map tasks	0	0	4
Map-Reduce Framework	Reduce input records	0	151	151
	Map output records	1,984,055	0	1,984,055
	Map output bytes	18,862,764	0	18,862,764
	Combine output records	1,063	151	1,214
	Map input records	2,923,923	0	2,923,923
	Reduce input groups	0	151	151
	Combine input records	1,984,625	493	1,985,118
	Map input bytes	236,903,179	0	236,903,179
	Reduce output records	0	151	151
File Systems	HDFS bytes written	0	2,658	2,658
	Local bytes written	20,554	2,510	23,064
	HDFS bytes read	236,915,470	0	236,915,470
	Local bytes read	21,112	2,510	23,622
AveragingWithCombiner$MapClass$ClaimsCounters	QUOTED	1	0	1
	MISSING	939,867	0	939,867

Рис. 6.6. В веб-интерфейсе JobTracker показана информация о счетчиках

Streaming-процессы тоже могут пользоваться счетчиками. Нужно только вывести на `STDERR` строку в специальном формате:

```
reporter:counter:group,counter,amount
```

где *group*, *counter* и *amount* соответствуют аргументам, передаваемым методу `incrCounter()` в программе на Java. Например, в скрипте на Python для инкремента счетчика `ClaimsCounters.MISSING` нужно написать:

```
sys.stderr.write("reporter:counter:ClaimsCounters,MISSING,1\n")
```

Не забудьте только добавить в конец символ новой строки ('\n'), иначе Hadoop Streaming не сможет правильно интерпретировать строку.

6.2.2. Пропуск плохих записей

При работе с большими наборами данных в некоторых записях неизбежно будут присутствовать ошибки. Очень часто приходится посвящать несколько итераций цикла разработки тому, чтобы сделать про-

грамму устойчивой к неожиданным данным[5]. Но сделать программу абсолютно «непробиваемой» все равно не удастся. Программе будет подаваться новые данные, в которых будут встречаться непредвиденные ошибки. Возможно даже, что вы используете анализатор, зависящий от сторонних библиотек вне вашего контроля. Приложив все усилия к тому, чтобы сделать программу устойчивой к неправильно отформатированным записям, вы также должны включить механизм восстановления после непредусмотренных ошибок. Ведь вы же не хотите, чтобы программа аварийно завершилась из-за одной-единственной плохой записи.

Встроенный в Hadoop механизм восстановления после аппаратных сбоев непригоден для восстановления после детерминированных программных ошибок, вызванных плохими записями. Однако имеется возможность пропускать записи, которые, по мнению Hadoop, могут привести к краху задания. Если этот механизм включен, то задание входит в режим пропуска после того, как несколько раз подряд завершилось аварийно. В режиме пропуска TaskTracker будет следить за тем, какой диапазон записей приводит к сбою, а при последующей попытке запуска обойдет этот диапазон.

Конфигурирование пропуска записей в программе на Java

Механизм пропуска появился в версии 0.19, но по умолчанию отключен. В программах на Java им управляет класс `SkipBadRecords`, в котором имеются только статические методы. Драйвер задачи должен вызвать один или оба следующих метода:

```
public static void setMapperMaxSkipRecords(Configuration conf,
       long maxSkipRecs)
public static void setReducerMaxSkipGroups(Configuration conf,
       long maxSkipGrps)
```

чтобы включить пропуск записей для заданий map и reduce соответственно. Обоим методам передается конфигурационный объект и максимальное количество записей в пропускаемом диапазоне. Если размер диапазона равен 0 (по умолчанию), то механизм пропуска записей отключается. Hadoop определяет, какой диапазон пропустить, методом деления пополам. При каждом запуске задания размер пропускаемого диапазона уменьшается вдвое и выясняется,

[5] Неожиданные данные – не всегда ошибки. Мне как-то рассказывали о программе, которая «падала» при обработке географической информации о пользователях. Расследование показало, что один пользователь проживал в реально существующем городе с названием Null.

в какой половине находятся плохие записи. Эта процедура продолжается, пока размер пропускаемого диапазона не окажется приемлемым. Операция довольно дорогая, особенно если задан небольшой размер диапазона. Возможно, понадобится увеличить максимальное количество попыток перезапуска, задаваемое для обычного механизма восстановления после сбоя задания. Это можно сделать с помощью методов `JobConf.setMaxMapAttempts()` и `JobConf.setMaxReduceAttempts()` или путем установки свойств `mapred.map.max.attempts` и `mapred.reduce.max.attempts`.

Если механизм пропуска включен, то Hadoop входит в режим пропуска после того, как задание аварийно завершится два раза подряд. Число попыток перезапуска, после которого активируется режим пропуска, можно увеличить, вызвав метод `setAttemptsToStartSkipping()` класса `SkipBadRecords`:

```
public static void setAttemptsToStartSkipping(Configuration conf,
      int attemptsToStartSkipping)
```

Hadoop копирует пропущенные записи в HDFS для последующего анализа. Они записываются в файлы последовательностей в каталоге `_log/skip`. Подробнее мы будем рассматривать файлы последовательностей в разделе 6.3.3, а пока можете считать, что это специфичный для Hadoop сжатый формат. Для распаковки и чтения такого файла предназначена команда:

```
bin/hadoop fs -text <filepath>
```

Каталог для хранения пропущенных записей можно изменить, вызвав метод `SkipBadRecords.setSkipOutputPath(JobConf conf, Path path)`. Если `path` равно `null` или объекту `Path` со строковым значением "`none`", то Hadoop не будет протоколировать пропущенные записи.

Конфигурирование пропуска записей в обход Java

Для конфигурирования механизма пропуска записей в программе, написанной на Java, достаточно вызвать из драйвера методы класса `SkipBadRecords`, но иногда это необходимо сделать с помощью общих флагов, распознаваемых классом `GenericOptionsParser`. Дело в том, что человек, запускающий программу, может знать о диапазоне ожидаемых плохих записей больше, чем разработчик, и, следовательно, способен правильнее задать параметры. К тому же, Streaming-программы просто не имеют доступа к классу `SkipBadRecords`. Механизм пропуска записей можно сконфигурировать с помощью флага

Streaming `-D` (`-jobconf` в версии 0.18). В табл. 6.2 показано, какие свойства класса `JobConf` устанавливаются различными методами `SkipBadRecords`.

Таблица 6.2. Эквивалентность свойств JobConf и методов класса SkipBadRecords.

Метод SkipBadRecords	Свойство JobConf
`setAttemptsToStartSkipping()`	`mapred.skip.attempts.to.start.skipping`
`setMapperMaxSkipRecords()`	`mapred.skip.map.max.skip.records`
`setReducerMaxSkipGroups()`	`mapred.skip.reduce.max.skip.groups`
`setSkipOutputPath()`	`mapred.skip.out.dir`
`setAutoIncrMapperProcCount()`	`mapred.skip.map.auto.incr.proc.count`
`setAutoIncrReducerProcCount()`	`mapred.skip.reduce.auto.incr.proc.count`

Мы еще не рассматривали последние два свойства. Их значения по умолчанию годятся для большинства программ на Java, но должны быть изменены для Streaming-программ. При определении диапазона пропускаемых записей Hadoop должен точно знать, сколько записей обработало задание. Hadoop ведет внутренний счетчик, который по умолчанию увеличивается на 1 после каждого обращения к функции map (или reduce). Для программ на Java этот способ подсчета обработанных записей прекрасно подходит. В некоторых случаях он может сбоить, например, когда программа обрабатывает записи асинхронно (скажем, в нескольких потоках) или буферизует их для обработки блоками, но обычно работает нормально. Однако в Streaming-программах такой подход вообще не работает, потому что не существует эквивалента функции map (или reduce), которая вызывалась бы при обработке каждой записи. В такой ситуации поведение по умолчанию следует отключить, присвоив этим булевским свойствам значение false, и самостоятельно обновлять счетчики записей в своем задании.

Задание map, написанное на Python, может обновить счетчик следующим образом:

```
sys.stderr.write(
➥ "reporter:counter:SkippingTaskCounters,MapProcessedRecords,1\n")
```

а задание reduce – так:

```
sys.stderr.write(
➥ "reporter:counter:SkippingTaskCounters,ReduceProcessedGroups,1\n")
```

Программа на Java, несовместимая с подразумеваемым по умолчанию способом подсчета записей, должна содержать обращения:

```
reporter.incrCounter(SkipBadRecords.COUNTER_GROUP,
   SkipBadRecords.COUNTER_MAP_PROCESSED_RECORDS, 1);
```

и

```
reporter.incrCounter(SkipBadRecords.COUNTER_GROUP,
   SkipBadRecords.COUNTER_REDUCE_PROCESSED_GROUPS, 1);
```

после завершения обработки пары ключ/значение в классах `Mapper` и `Reducer` соответственно.

6.2.3. Перезапуск сбойных заданий с помощью IsolationRunner

Отладка с использованием файлов журналов – это по сути дела реконструкция событий на основе обобщенной исторической информации. Но иногда информации в журнале недостаточно для выявления причины ошибки. В Hadoop имеется утилита IsolationRunner, которая с точки зрения отладки ведет себя, как машина времени. Она позволяет изолировать и перезапустить сбойное задание с теми же входными данными и на том же узле, что в момент сбоя. После этого к заданию можно присоединить отладчик и заняться изучением обстоятельств появления ошибки.

Для использования IsolationRunner необходимо запустить задачу, присвоив конфигурационному свойству `keep.failed.tasks.files` значение `true`. Тем самым вы просите каждый узел TaskTracker сохранить все данные, необходимые для перезапуска сбойных заданий.

После аварийного завершения задачи найдите в веб-интерфейсе JobTracker узел, идентификатор задачи и идентификатор попытки запуска сбойного задания. Затем зайдите на тот узел, где произошел сбой задания, и перейдите в подкаталог `work` каталога, относящегося к этой попытке запуска, точнее

local_dir`/taskTracker/jobcache/`***job_id/attempt_id***`/work`

где *job_id* и *attempt_id* – идентификатор задачи и идентификатор попытки запуска сбойного задания (идентификатор задачи должен иметь префикс «job_», идентификатор попытки запуска – префикс «attempt_»). Корневой каталог *local_dir* задается в свойстве `mapred.local.dir`. Отметим, что Hadoop допускает наличие нескольких локальных каталогов на одном узле (для этого список каталогов нужно

перечислить в `mapred.local.dir` через запятую); это позволяет распределить операции ввода/вывода по нескольким дискам. Если узел сконфигурирован таким образом, то придется поискать нужный подкаталог *attempt_id* во всех локальных каталогах.

Находясь в каталоге `work`, выполните команду IsolationRunner для перезапуска сбойного задания с теми данными, с которыми оно было запущено первоначально. При перезапуске нам нужно, чтобы виртуальная машина Java (JVM) была подготовлена к удаленной отладке. Поскольку JVM запускается не напрямую, а посредством скрипта `bin/hadoop`, параметры для активации отладки нужно задавать в переменной окружения HADOOP_OPTS:

```
export HADOOP_OPTS="-agentlib:jdwp=transport=dt_socket,server=y,
address=8000"
```

В данном случае мы говорим, что JVM должна прослушивать порт 8000 и не начинать исполнение кода до присоединения отладчика[6]. Теперь можно воспользоваться IsolationRunner для перезапуска задачи:

```
bin/hadoop org.apache.hadoop.mapred.IsolationRunner ../job.xml
```

В файле job.xml file содержится вся конфигурационная информация, необходимая IsolationRunner. Следуя нашим указаниям, JVM не начнет исполнять задание, пока не будет присоединен отладчик. Присоединить можно любой отладчик Java, поддерживающий протокол Java Debug Wire Protocol (JDWP). Все основные среды разработки на Java (IDE) его поддерживают. Например, отладчик jdb присоединяется к JVM командой

```
jdb -attach 8000
```

(Разумеется, это только пример. Надеюсь, вы пользуетесь чем-то получше jdb!) О том, как присоединить к JVM отладчик из вашей любимой IDE, см. прилагаемую к ней документацию.

6.3. Оптимизация производительности

Разработав и отладив MapReduce-программу, вы, наверное, захотите оптимизировать ее производительность. Но прежде чем присту-

[6] Параметры настройки Sun JVM для отладки подробно описаны в документации Sun по адресу http://java.sun.com/javase/6/docs/technotes/guides/jpda/conninv.html#Invocation.

пать к этому, вспомните, что одно из главных достоинств Hadoop – линейная масштабируемость. Ускорить работу многих задач можно просто добавлением новых машин. Если ваш кластер невелик, то это имеет прямой смысл с точки зрения экономии средств. Подумайте, сколько времени вам понадобится, чтобы повысить быстродействие программы на 10 процентов. А ведь в кластере из 10 узлов того же результата можно добиться, добавив всего одну машину (и при этом выиграют *все* задачи, работающие в кластере). Ваше рабочее время стоит дороже этого компьютера. С другой стороны, в кластер из 1000 узлов для получения 10-процентного ускорения придется добавить 100 машин. При таком масштабе повышение производительности с помощью грубой силы может оказаться экономически невыгодным.

В Hadoop имеется ряд средств для настройки производительности, причем некоторые из них дают эффект для всего кластера. Мы рассмотрим их в следующей главе при обсуждении вопросов администрирования. А в этом разделе поговорим о приемах, применимых к одной задаче.

6.3.1. Уменьшение сетевого трафика с помощью комбинатора

Комбинатор позволяет уменьшить объем данных, передаваемых от распределителей редукторам в ходе тасования, а снижение сетевого трафика напрямую связано с уменьшением времени выполнения. Подробно об использовании и преимуществах комбинаторов мы говорили в разделе 4.6. Здесь мы напомнили о них только для полноты картины.

6.3.2. Уменьшение объема выходных данных

Иногда при обработке больших наборов данных на чтение с диска тратится заметное время. Уменьшив количество считываемых байтов, мы сможем повысить общую пропускную способность. Для этого есть несколько способов.

Самое простое – сократить объем обрабатываемых данных. Можно, скажем, обрабатывать только некоторую выборку из всего набора. Для некоторых аналитических приложений это вполне разумный подход, поскольку уменьшается лишь количественная точность, но

не качественная картина. Во многих системах поддержки принятия решений ценность результатов от этого не страдает.

Часто задачи MapReduce используют не всю информацию, присутствующую в наборе данных. Вспомните данные с описаниями патентов из главы 4. В каждой записи почти два десятка полей, но в большинстве программ мы использовали только малую их часть. Неэффективно, когда каждая задача, работающая с этим набором, вынуждена всякий раз считывать все неиспользуемые поля. В таком случае можно разделить набор входных данных на несколько меньших, оставив в каждом только поля, необходимые для конкретного вида обработки. Как именно провести такую реорганизацию, зависит от приложения. Эта техника аналогична технологиям вертикального секционирования и хранения по столбцам, применяемым в реляционных СУБД.

Наконец, объем дискового и сетевого ввода/вывода можно уменьшить за счет сжатия данных. Эту технику можно применять как к промежуточным, так и к выходным наборам данных. В Hadoop есть много способов сжатия данных, и мы посвятим этой теме весь следующий подраздел.

6.3.3. Использование сжатия

Даже при использовании комбинатора результаты стадии распределения все равно могут оказаться очень велики. Эти промежуточные данные необходимо сохранить на диске и затем передать по сети. Сжатие промежуточных данных способно повысить производительность большинства задач MapReduce и при этом достигается малой кровью.

В Hadoop встроена поддержка сжатия и распаковки. Чтобы включить режим сжатия выходных данных распределителя, нужно установить два конфигурационных свойства, показанных в табл. 6.3.

Таблица 6.3. Конфигурационные свойства, управляющие сжатием выходных данных распределителя

Свойство	Описание
`mapred.compress.map.output`	Булевское свойство, определяющее, нужно ли производить сжатие выходных данных распределителя.
`mapred.map.output.compression.codec`	Свойство, определяющее, какой класс `CompressionCodec` использовать для сжатия.

Чтобы включить режим сжатия выходных данных распределителя, следует присвоить свойству `mapred.compress.map.output` значение `true`. Кроме того, в свойство `mapred.map.output.compression.codec` нужно записать класс кодека. Классы всех кодеков в Hadoop реализуют интерфейс `CompressionCodec`. Hadoop поддерживает несколько кодеков для сжатия (табл. 6.4). Например, чтобы использовать алгоритм GZIP, настройте конфигурационный объект следующим образом:

```
conf.setBoolean("mapred.compress.map.output", true);
conf.setClass("mapred.map.output.compression.codec",
              GzipCodec.class,
              CompressionCodec.class);
```

Таблица 6.4. Список кодеков в пакете org.apache.hadoop.io.compress

Кодек	Версия Hadoop	Описание
`DefaultCodec`	0.18, 0.19, 0.20	Работает с файлами в формате zlib. По принятому в Hadoop соглашению, такие файлы имеют расширение .deflate.
`GzipCodec`	0.18, 0.19, 0.20	Работает с файлами в формате gzip. Такие файлы имеют расширение .gz.
`BZip2Codec`	0.19, 0.20	Работает с файлами в формате bzip2. Такие файлы имеют расширение .bz2. Этот формат отличается тем, что допускает разбиение на порции для Hadoop, даже если используется не в файлах последовательностей.

Вместо того чтобы устанавливать эти свойства напрямую, можно воспользоваться вспомогательными методами `setCompressMapOutput()` и `setMapOutputCompressorClass()` класса `JobConf`.

Выходные данные распределителей используются только внутри задачи, поэтому включение для них сжатия прозрачно для разработчика и не требует никаких усилий. Поскольку многие приложения MapReduce состоят из нескольких задач, хранение входных и выходных данных в сжатой форме вполне оправданно. Для передачи данных от одной задачи Hadoop другой настоятельно рекомендуется использовать специальный формат *файла последовательности*.

Файл последовательности – это допускающий сжатие двоичный формат для хранения пар ключ/значение. Он спроектирован так, чтобы даже в случае сжатия файл можно было разбивать на порции. Напомним, что один из способов достижения параллелизма в Hadoop –

возможность разбить входной файл на порции, которые могут читать и обрабатывать одновременно несколько задач. Если входной файл хранится в сжатом виде, то Hadoop должен разбивать его так, чтобы распределители могли независимо распаковать свои порции, иначе сам Hadoop должен будет предварительно распаковать весь файл, что сводит на нет выгоды от распараллеливания. Не каждый формат сжатия поддерживает разбиение на куски с возможностью независимой распаковки. Для решения этой задачи и были разработаны файлы последовательностей. В таком файле имеются *синхромаркеры*, обозначающие допустимые границы порций[7].

Помимо сжатия и разбиения, файлы последовательностей поддерживают двоичные ключи и значения. Поэтому они часто применяются для обработки двоичных документов, например изображений, но не менее хорошо подходят и для текстовых и других документов, в которых ключом или значением является большой объект. Каждый документ рассматривается как отдельная запись файла последовательности.

Чтобы заставить задачу MapReduce выводить файл последовательности, нужно в качестве выходного формата задать `SequenceFileOutputFormat`. При этом имеет смысл изменить тип сжатия с подразумеваемого по умолчанию RECORD на BLOCK. В первом случае каждая запись сжимается независимо, а во втором сжатию подвергается блок записей, так что коэффициент сжатия оказывается выше. Наконец, нужно вызвать статические методы `setCompressOutput()` и `setOutputCompressorClass()` класса `FileOutputFormat` (или `SequenceFileOutputFormat`, который ему наследует), чтобы включить режим сжатия и задать кодек. Поддерживаются те же кодеки, что перечислены в табл. 6.4. Добавьте в драйвер такие строки:

```
conf.setOutputFormat(SequenceFileOutputFormat.class);
SequenceFileOutputFormat.setOutputCompressionType(conf,
CompressionType.BLOCK);
FileOutputFormat.setCompressOutput(conf, true);
FileOutputFormat.setOutputCompressorClass(conf, GzipCodec.class);
```

В табл. 6.5 перечислены эквивалентные свойства для конфигурирования вывода в файл последовательности. Streaming-программа будет выводить файлы последовательности, если запустить ее со следующими аргументами:

```
-outputformat org.apache.hadoop.mapred.SequenceFileOutputFormat
-D mapred.output.compression.type=BLOCK
```

[7] До сих пор мы работали только с несжатыми текстовыми файлами, в которых каждая запись – это строка. Символ новой строки (‘\n’) можно считать тривиальным синхромаркером, обозначающим одновременно границы записей и границы порций.

```
-D mapred.output.compress=true
-D mapred.output.compression.codec=org.apache.hadoop.io.compress.
GzipCode
```

Таблица 6.5. Конфигурационные свойства для вывода сжатого файла последовательности

Свойство	Описание
`mapred.output.compression.type`	Строковое свойство, задающее тип сжатия файла последовательности. Может принимать значения NONE, RECORD, BLOCK. По умолчанию RECORD, но BLOCK часто обеспечивает более плотное сжатие. **Вспомогательный метод:** SequenceFileOutputFormat. ➥ setOutputCompressionType()
`mapred.output.compress`	Булевское свойство, определяющее, нужно ли сжимать выходные данные задачи. **Вспомогательный метод:** FileOutputFormat.setCompressOutput()
`mapred.output.compression.codec`	Свойство, задающее класс кодека, применяемого для сжатия выходных данных задачи. **Вспомогательный метод:** FileOutputFormat. ➥ setOutputCompressorClass()

Для чтения файла последовательности задайте входной формат `SequenceFileInputFormat`. В программе на Java напишите

```
conf.setInputFormat(SequenceFileInputFormat.class);
```

а при запуске Streaming-программы:

```
-inputformat org.apache.hadoop.mapred.SequenceFileInputFormat
```

Конфигурировать тип сжатия и класс кодека не нужно, так как класс `SequenceFile.Reader` (используемый в `SequenceFileRecordReader`) автоматически определяет то и другое из заголовка файла.

6.3.4. Повторное использование JVM

По умолчанию демон TaskTracker запускает каждое задание `Mapper` и `Reducer` в отдельной JVM как дочерний процесс. При этом неизбежны накладные расходы на запуск JVM. Если распределитель выполняет собственную инициализацию, например считывает в память большую структуру данных (см. пример соединения с использованием распределенного кэша в разделе 5.2.2), то и эти издержки следует

причислить к накладным расходам. Если каждое задание работает очень недолго или инициализация распределителя занимает много времени, то время запуска может составлять значительную долю общего времени работы задания.

Начиная с версии 0.19.0, Hadoop допускает повторное использование JVM несколькими заданиями, входящими в одну и ту же задачу. Таким образом, стоимость запуска раскладывается на много задач. Новое свойство `mapred.job.reuse.jvm.num.tasks` определяет максимальное число заданий (в одной задаче), исполняемых одной JVM. По умолчанию оно равно 1, то есть JVM не используется повторно. Чтобы включить режим повторного использования, укажите значение, большее 1. Значение –1 означает, что число заданий, исполняемых одной JVM, не ограничено. В классе `JobConf` есть вспомогательный метод `setNumTasksToExecutePerJvm(int)` позволяющий установить это свойство для задачи.

Таблица 6.6. Конфигурационное свойство для включения режима повторного использования JVM

Свойство	Описание
`mapred.job.reuse.jvm.num.tasks`	Целочисленное свойство, определяющее максимальное количество заданий, исполняемых одной JVM. Значение –1 означает «не ограничено».

6.3.5. Наблюдаемое исполнение

Одно из допущений, положенных в основу дизайна MapReduce (как утверждается в оригинальной статье Google), – это предположение о том, что все узлы ненадежны, и каркас должен уметь обрабатывать случай отказа узла в середине исполнения задачи. Исходя из этого допущения, каркас MapReduce постулирует, что все задания распределения и редукции должны быть *идемпотентными*. Это означает, что в случае сбоя задания Hadoop может его перезапустить, а результат задачи в целом при этом не изменится. Hadoop умеет следить за состоянием работающих узлов и автоматически перезапускать задания на отказавших узлах. Таким образом, механизм отказоустойчивости прозрачен для разработчика.

Но часто узел не просто бесповоротно отказывает, а начинает работать медленно из-за частичного выхода из строя устройств ввода/вывода. А иногда скорость обработки падает вследствие временной

перегрузки. Это не отражается на правильности результатов задачи, но, конечно, сказывается на производительности. Даже одно медленно работающее задание задерживает задачу MapReduce. Пока *все* распределители не закончат работу, не может быть запущен ни один редуктор. И аналогично до завершения всех редукторов задача не может считаться завершенной.

В Hadoop свойство идемпотентности применяется еще и для сглаживания проблемы медленных заданий. Hadoop не только перезапускает сбойные задания, но также обнаруживает медленно работающее и запускает его же *параллельно* на другом узле. Идемпотентность гарантирует, что параллельное задание порождает тот же самый результат. Hadoop отслеживает параллельные задания. Как только одно из них успешно завершается, Hadoop снимает остальные его экземпляры и использует результаты первого завершившегося. Этот механизм называется *наблюдаемым исполнением* (speculative execution).

Отметим, что режим наблюдаемого исполнения задания распределения включается только после того, как все такие задания запланированы на выполнение, и только для тех, которые сильно отстают от средних показателей. Точно так же обстоит дело с наблюдаемым исполнением заданий редукции. Этот режим не преследует цели устроить «состязание» нескольких экземпляров задания, чтобы получить наилучшее время завершения. Он лишь не дает медленным заданиям затормозить задачу в целом.

По умолчанию механизм наблюдаемого исполнения включен. Его можно отключить по отдельности для заданий распределения и редукции. Для этого присвойте одному или обоим свойствам в табл. 6.7 значение `false`. Эти свойства применяются на уровне задачи, но можно также изменить их для всего кластера, задав в конфигурационном файле.

Таблица 6.7. Конфигурационные свойства для включения режима наблюдаемого исполнения

Свойство	Описание
`mapred.map.tasks.speculative.execution`	Булевское свойство, определяющее, включен ли режим наблюдаемого исполнения для заданий распределения.
`mapred.reduce.tasks.speculative.execution`	Булевское свойство, определяющее, включен ли режим наблюдаемого исполнения для заданий редукции.

В общем случае, лучше оставить режим наблюдаемого исполнения включенным. Отключать его имеет смысл тогда, когда задания распределения или редукции имеют *побочные эффекты*, то есть не являются идемпотентными. Например, если задание что-то записывает во внешние файлы, то при включенном режиме наблюдаемого исполнения несколько экземпляров задания могут попытаться создать одни и те же внешние файлы. В таком случае надо отключить этот режим, чтобы в каждый момент времени работал только один экземпляр задания.

Примечание. Если задания имеют побочные эффекты, то следует продумать взаимодействие с механизмом восстановления. Например, если задание пишет во внешний файл, то не исключено, что оно аварийно завершится сразу после записи в этой файл. В таком случае Hadoop перезапустит задание, и оно попытается снова записать в тот же внешний файл. Вы должны позаботиться о том, чтобы задание и в таких условиях функционировало правильно.

6.3.6. Переработка кода и модификация алгоритмов

Если вы готовы переписать свою MapReduce-программу ради оптимизации производительности, то должны знать как о простых приемах, так и о менее тривиальных подходах к повышению быстродействия, зависящих от конкретного приложения.

Один простой способ, относящийся к Streaming-программам, – переписать их на Java с использованием Hadoop API. Интерфейс Streaming – прекрасная вещь для быстрого создания задачи MapReduce с целью однократного анализа данных, но работают такие задачи медленнее, чем написанные на Java. Бывает, что Streaming-программа, поначалу созданная для одноразового запроса, оказывается полезной и часто запускается; тогда имеет смысл переписать ее на Java.

Если имеется несколько задач, работающих с одними и теми же входными данными, то, возможно, удастся переписать их так, чтобы общее число задач стало меньше. Например, если нужно вычислять максимум и минимум в наборе данных, то лучше написать одну задачу MapReduce, которая будет вычислять обе величины, чем считать их двумя разными задачами. Это может показаться очевидным, но на практике многие задачи изначально писались для выполнения какой-то одной функции. И это правильный подход к проектирова-

нию. В силу узкой специализации задача будет применима к разным наборам данных и для разных целей. И лишь накопив опыт применения, вы начинаете смотреть, нельзя ли объединить некоторые группы задач в одну, чтобы ускорить их работу.

Но едва ли не самым важным способом ускорения MapReduce-программы является переосмысление лежащего в ее основе алгоритма и попытка отыскать более эффективный алгоритм, дающий те же результаты за меньшее время. Это справедливо для программирования вообще, но для MapReduce-программ особенно. В стандартных учебниках по алгоритмам и структурам данных (сортировка, списки, словари и т. п.) эти вопросы подробно рассматриваются в применении к традиционному программированию. Но программы для Hadoop затрагивают такие «экзотические» области, как распределенные вычисления, функциональное программирование, математическая статистика и обработка больших объемов данных, с которыми большинство программистов знакомы слабо и где по сей день еще ведутся активные исследования в поисках новых подходов.

Выше мы уже встречались с одним примером новой структуры данных, ускоряющей работу MapReduce-программ, – применение фильтров Блума для вычисления полусоединений (раздел 5.3). Фильтр Блума хорошо известен специалистам по распределенным вычислениям, но гораздо хуже вне этого узкого круга.

Еще один классический пример применения нового алгоритма для ускорения MapReduce-программ – вычисление дисперсии. Люди, далекие от статистики, наверное, станут вычислять ее по канонической формуле:

$$(1/N) * Sum_i[(X_i - X_{avg})^2],$$

где Sum_i обозначает суммирование по всему набору данных, а X_{avg} – среднее значение этого набора. Если среднее заранее неизвестно, то неспециалист, скорее всего, сначала запустит одну задачу для его вычисления, а затем другую – для вычисления дисперсии. Но человек, знакомый с вычислительной статистикой, воспользуется эквивалентной формулой:

$$(1/N) * Sum_i[(X_i)^2] - ((1/N) * Sum_i[X_i])^2.$$

Теперь нам нужно вычислить сумму X и сумму X^2, но обе суммы можно вычислить за один проход с использованием единственной задачи MapReduce (по аналогии с одновременным вычислением

максимума и минимума). Знание некоторых основ математической статистики позволило сократить время обработки вдвое[8].

Следует также обращать внимание на вычислительную сложность алгоритмов. Hadoop обеспечивает линейное масштабирование, но и его можно поставить на колени, если подать большой набор данных на вход алгоритму с квадратичной или еще худшей сложностью. В таких случаях поищите более эффективный алгоритм. Иногда приходится соглашаться на более быстрый алгоритм, хотя он дает лишь приблизительные результаты.

6.4. Резюме

Методики разработки для Hadoop построены на базе передовых практик программирования на Java, в частности автономного тестирования и разработки через тестирование. Из того факта, что главную роль в Hadoop играет обработка данных, вытекает потребность в процедурах тестирования, в большей степени ориентированных на данные. В программах, работающих с данными, чаще встречаются математические и логические ошибки, а они часто остаются незамеченными. Из-за распределенной природы Hadoop отладка резко усложняется. Чтобы справиться с этой проблемой, тестировать необходимо поэтапно, сначала в локальном режиме, потом в псевдораспределенном на одном узле и только в конце переходить к полностью распределенному режиму.

Знаменитый ученый в области информатики Дональд Кнут однажды сказал, что «преждевременная оптимизация – корень всех зол». Приступать к оптимизации программы для Hadoop нужно только после того, как она полностью отлажена. Помимо общих вопросов выбора алгоритма и выполнения вычислений, у проблемы повышения производительности есть и платформенно-зависимые аспекты, и Hadoop предлагает целый ряд приемов, позволяющих сделать задачи более эффективными.

---

[8] Вычислительные операции с большими наборами данных характеризуются разнообразными нюансами. Так, альтернативный вариант вычисления дисперсии дает меньшую точность и с большей вероятностью может привести к переполнению.

ГЛАВА 7.
Сборник рецептов

В этой главе:

- Передача нестандартных параметров задаче.
- Получение информации о конкретном задании.
- Создание нескольких выходных наборов данных.
- Интерфейс с реляционными базами данных.
- Глобальная сортировка выходных данных.

До сих пор мы рассматривали базовые приемы создания MapReduce-программ. Но Hadoop – это большой каркас, функциональность которого отнюдь не ограничивается простыми приемами. В век поисковых машин, таких, как Bing и Google, несложно найти информацию о специализированных методах программирования в парадигме MapReduce, поэтому мы не будем пытаться создать энциклопедический справочник. Но наш собственный опыт и обсуждения с другими пользователями Hadoop позволили выделить несколько общеполезных приемов, например, использование стандартных реляционных баз данных в качестве входа или выхода задачи MapReduce. Некоторые из своих излюбленных «рецептов» мы и собрали в этой главе.

7.1. Передача нестандартных параметров задаче

При написании классов, реализующих интерфейсы `Mapper` и `Reducer` часто желательно сделать некоторые аспекты их поведения конфигурируемыми. Например, в код нашей программы соединения из главы 5 зашито, что ключом соединения является первый столбец.

Сферу применения программы можно было бы расширить, дав пользователю возможность задавать ключ во время выполнения. Сам Hadoop использует для хранения конфигурационных свойств задачи специальный объект. Но этот же объект позволяет передавать параметры вашим классам `Mapper` и `Reducer`.

Мы видели, как драйвер MapReduce-программы конфигурирует объект `JobConf`, задавая различные свойства: входной и выходной формат, класс распределителя и т. д. Для определения собственного свойства вы должны присвоить ему уникальное имя и задать значение в том же конфигурационном объекте. Этот объект передается всем демонам TaskTracker, поэтому заданные в нем свойства доступны всем заданиям, входящим в состав задачи. Ваши классы `Mapper` и `Reducer` могут обратиться к нему и получить значение нужного свойства.

Класс `Configuration` (родитель `JobConf`) содержит несколько общих методов установки свойств. Свойство – это пара ключ/значение, в которой ключом является строка типа `String`, а значение может иметь один из стандартных типов. Сигнатуры общих методов установки таковы:

```
public void set(String name, String value)
public void setBoolean(String name, boolean value)
public void setInt(String name, int value)
public void setLong(String name, long value)
public void setStrings(String name, String... values)
```

Обратите внимание, что в Hadoop свойства хранятся в виде строк. Все остальные методы – не более чем обертки вокруг `set(String, String)`. Например, метод `setStrings(String, String...)` принимает массив строк, преобразует его с одну строку, в которой значения разделены запятыми, и делает эту строку значением свойства. Метод `getStrings()` разбивает конкатенированную строку, получая исходный массив. Поэтому в исходных строках, передаваемых методу `setStrings`, не должно быть запятых. Если это все-таки необходимо, то вам придется написать собственную функцию кодирования строк.

Ваш драйвер сначала должен установить свойства в конфигурационном объекте, чтобы сделать их доступными всем заданиям. Ваши классы `Mapper` и `Reducer` смогут получить доступ к конфигурационному объекту с помощью метода `configure()`. На этапе инициализации задание вызывает метод `configure()`, который вы должны переопределить для получения и сохранения значений свойств.

Впоследствии к сохраненным свойствам смогут обращаться методы `map()` и `reduce()`. В следующем примере определяется новое свойство `myjob.myproperty`, целочисленное значение которого задает пользователь.

```
public int run(String[] args) throws Exception {

   Configuration conf = getConf();
   JobConf job = new JobConf(conf, MyJob.class);

   ...

   job.setInt("myjob.myproperty", Integer.parseInt(args[2]));

   JobClient.runJob(job);
   return 0;
}
```

Установить нестандартное свойство 1

Метод `configure()` класса `MapClass` извлекает значение свойства и сохраняет его в объекте. Любой метод чтения в классе `Configuration` принимает значение по умолчанию, которое возвращается, если запрошенное свойство не установлено в конфигурационном объекте. В данном случае значение по умолчанию равно 0:

```
public static class MapClass extends MapReduceBase
   implements Mapper<Text, Text, Text, Text> {

   int myproperty;

   public void configure(JobConf job) {
      myproperty = job.getInt("myjob.myproperty", 0);
   }

   ...
}
```

Получить нестандартное свойство 1

Чтобы использовать это свойство в классе `Reducer`, его также нужно предварительно извлечь:

```
public static class Reduce extends MapReduceBase
   implements Reducer<Text, Text, Text, Text> {

   int myproperty;

   public void configure(JobConf job) {
      myproperty = job.getInt("myjob.myproperty", 0);
   }

   ...
}
```

В классе `Configuration` методов чтения больше, чем методов установки, но большая их часть не нуждается в пояснениях. Почти все методы чтения ожидают получить в качестве аргумента значение по умолчанию. Исключение составляет метод `get(String)`, который возвращает `null`, если свойство с указанным именем не было установлено.

```
public String get(String name)
public String get(String name, String defaultValue)
public boolean getBoolean(String name, boolean defaultValue)
public float getFloat(String name, float defaultValue)
public int getInt(String name, int defaultValue)
public long getLong(String name, long defaultValue)
public String[] getStrings(String name, String... defaultValue)
```

При условии, что наш класс задачи реализует интерфейс `Tool` и использует класс `ToolRunner`, мы можем предоставить пользователю возможность задавать в командной строке нестандартные свойства точно так же, как стандартные конфигурационные свойства Hadoop:

```
bin/hadoop jar MyJob.jar MyJob -D myjob.myproperty=1 input output
```

Можно убрать из драйвера строку, в которой требуется, чтобы пользователь обязательно задал значение этого свойства в качестве аргумента. Это удобно, когда имеется значение по умолчанию, используемое в большинстве случаев.

```
public int run(String[] args) throws Exception {

   Configuration conf = getConf();
   JobConf job = new JobConf(conf, MyJob.class);

   ...

   int myproperty = job.getInt("myjob.myproperty", 0);
   if (myproperty < 0) {
      System.err.println("Invalid myjob.myproperty: " + myproperty);
      System.exit(0);
   }

  JobClient.runJob(job);
   return 0;
}
```

Если пользователю разрешено задавать нестандартные свойства, то рекомендуется проверять заданные значения в драйвере. В примере выше проверяется, что указанное значение свойства `myjob.myproperty` неотрицательно.

7.2. Получение информации о конкретном задании

Помимо получения нестандартных свойств и глобальной конфигурации, методы чтения, имеющиеся в конфигурационном объекте, можно использовать для получения информации о состояния текущего задания и задачи. Например, в классе `Mapper` можно прочитать из свойства `map.input.file` путь к файлу текущего задания распределения. Именно так метод `configure()` класса `DataJoinMapperBase` из пакета datajoin получает тег, в котором записана информация об источнике данных.

```
this.inputFile = job.get("map.input.file");
this.inputTag = generateInputTag(this.inputFile);
```

В табл. 7.1 перечислены некоторые свойства, сообщающие информацию о состоянии задания.

Таблица 7.1. Информация о состоянии задания, хранящаяся в конфигурационном объекте.

Свойство	Тип	Описание
`mapred.job.id`	String	Идентификатор задачи
`mapred.jar`	String	Местоположение jar-файла в каталоге задачи
`job.local.dir`	String	Место для хранения временных данных задачи
`mapred.tip.id`	String	Идентификатор задания
`mapred.task.id`	String	Идентификатор попытки запуска задания
`mapred.task.is.map`	boolean	Флаг, определяющий, является ли это задание распределителем
`mapred.task.partition`	int	Идентификатор задания внутри задачи
`map.input.file`	String	Путь к файлу, из которого читает распределитель
`map.input.start`	long	Смещение первого байта порции текущего распределителя от начала файла
`map.input.length`	long	Количество байтов в порции текущего распределителя
`mapred.work.output.dir`	String	Рабочий (то есть временный) каталог для выходных данных

Конфигурационные свойства также доступны Streaming-программам в виде переменных окружения. Перед тем как исполнять скрипт, Streaming API добавляет все конфигурационные свойства в окружение программы. Имена свойств изменяются, так что все символы, кроме алфавитно-цифровых, заменяются знаками подчеркивания (_). Вот, например, как запущенный через Streaming скрипт может опросить переменную окружения `map_input_file`, чтобы получить полный путь к файлу, который читает текущий распределитель.

```
import os
filename = os.environ["map_input_file"]
localdir = os.environ["job_local_dir"]
```

Этот пример иллюстрирует доступ к конфигурационным свойствам из скрипта на языке Python.

7.3. Разбиение на несколько выходных файлов

Все рассмотренные до сих пор задачи MapReduce выводили один набор файлов. Но часто бывает удобнее вывести несколько наборов файлов или разбить один набор данных на несколько. Популярный пример – разбиение одного большого файла журнала на несколько журналов, по одному на каждый день.

Класс `MultipleOutputFormat` дает простой способ сгруппировать похожие записи и поместить каждую группу в отдельный набор данных. Перед тем как выводить запись, этот класс выходного формата вызывает внутренний метод для определения имени файла, в который следует производить запись. Точнее говоря, вы сами должны расширить один из подклассов `MultipleOutputFormat` и реализовать метод `generateFileNameForKeyValue()`. Расширенный вами подкласс определяет выходной формат. Например, `MultipleTextOutputFormat` выводит текстовые файлы, а `MultipleSequenceFileOutputFormat` – файлы последовательностей. В любом случае следует переопределить метод, возвращающий имя файла для переданной пары, содержащей выходные ключ и значение:

```
protected String generateFileNameForKeyValue(K key, V value, String name)
```

Реализация по умолчанию возвращает аргумент `name`, то есть имя файла без пути. Вы можете вернуть из этого метода имя файла, зависящее от содержимого записи.

В примере ниже мы производим разбиение по стране, взятой из метаданных о патенте (листинг 7.1). Все патенты, выданные изобретателям из США, попадут в один набор файлов, из Японии – в другой и т. д. Приведенный набросок программы представляет собой задачу без редуктора, которая принимает входные данные и сразу же выводит их. Основное внесенное нами изменение – это создание собственного подкласса `MultipleTextOutputFormat`, который мы назвали `PartitionbyCountryMTOF` (*MTOF* – сокращение от `MultipleTextOutputFormat`.) Наш подкласс сохраняет каждую запись в файле, местоположение которого зависит от страны проживания изобретателя. Поскольку мы трактуем значение, возвращенное `generateFileNameForKeyValue()` как путь к файлу, то можем создать для каждой страны свой каталог, вернув `country + "/" + filename`.

Листинг 7.1.
Разнесение метаданных о патентах по нескольким каталогам в зависимости от страны

```
public class MultiFile extends Configured implements Tool {

   public static class MapClass extends MapReduceBase
      implements Mapper<LongWritable, Text, NullWritable, Text> {

      public void map(LongWritable key, Text value,
                      OutputCollector<NullWritable, Text> output,
                      Reporter reporter) throws IOException {
         output.collect(NullWritable.get(), value);
      }
   }

   public static class PartitionByCountryMTOF
      extends MultipleTextOutputFormat<NullWritable,Text>
   {
      protected String generateFileNameForKeyValue(NullWritable key,
                                                   Text value,
                                                   String filename)
      {
         String[] arr = value.toString().split(",", -1);
         String country = arr[4].substring(1,3);
         return country + "/" + filename;
      }
   }

   public int run(String[] args) throws Exception {
      Configuration conf = getConf();

      JobConf job = new JobConf(conf, MultiFile.class);

      Path in = new Path(args[0]);
```

```
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setJobName("MultiFile");
        job.setMapperClass(MapClass.class);
        job.setInputFormat(TextInputFormat.class);
        job.setOutputFormat(PartitionByCountryMTOF.class);
        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Text.class);

        job.setNumReduceTasks(0);

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new MultiFile(),
                                 args);
        System.exit(res);
    }
}
```

Когда эта программа завершится, мы увидим, что в выходном каталоге появились отдельные подкаталоги для каждой страны.

```
ls output/
AD    BN    CS    GE    IN    LC    MT    PH    SV    VE
AE    BO    CU    GF    IQ    LI    MU    PK    SY    VG
AG    BR    CY    GH    IR    LK    MW    PL    SZ    VN
AI    BS    CZ    GL    IS    LR    MX    PT    TC    VU
AL    BY    DE    GN    IT    LT    MY    PY    TD    YE
AM    BZ    DK    GP    JM    LU    NC    RO    TH    YU
AN    CA    DO    GR    JO    LV    NF    RU    TN    ZA
AR    CC    DZ    GT    JP    LY    NG    SA    TR    ZM
AT    CD    EC    GY    KE    MA    NI    SD    TT    ZW
AU    CH    EE    HK    KG    MC    NL    SE    TW
AW    CI    EG    HN    KN    MG    NO    SG    TZ
AZ    CK    ES    HR    KP    MH    NZ    SI    UA
BB    CL    ET    HT    KR    ML    OM    SK    UG
BE    CM    FI    HU    KW    MM    PA    SM    US
BG    CN    FO    ID    KY    MO    PE    SN    UY
BH    CO    FR    IE    KZ    MQ    PF    SR    UZ
BM    CR    GB    IL    LB    MR    PG    SU    VC
```

А внутри каталога страны находятся файлы, содержащие только записи о патентах, выданных в этой стране.

```
ls output/AD
part-00003 part-00005 part-00006

head output/AD/part-00006
5765303,1998,14046,1996,"AD","",,1,12,42,5,59,11,1,0.4545,0,0,1,67.3636,,,,
5785566,1998,14088,1996,"AD","",,1,9,441,6,69,3,0,1,,0.6667,,4.3333,,,,
5894770,1999,14354,1997,"AD","",,1,,82,5,51,4,0,1,,0.625,,7.5,,,,
```

Эта простенькая программка осуществляет только распределение. Но ту же технику можно применить и к выходу редукторов. Однако не путайте ее с разбивателем из каркаса MapReduce. Разбиватель анализирует ключи *промежуточных* записей и решает, какому редуктору передать запись. А программа разбиения, написанная нами, анализирует пару ключ/значение *выходной* записи и решает, в какой файл ее записать.

Формат `MultipleOutputFormat` прост, но ограничен. Например, мы сумели разнести по разным файлам строки входных данных, а что если надо было бы разнести столбцы? Допустим, мы хотим создать из набора метаданных о патентах два других набора: в одном для каждого патента находится информация, относящаяся ко времени (например, дата публикации), а в другом – к географии (например, страна проживания изобретателя). Выходные форматы и типы ключей и значений в обоих наборах данных могут различаться. В таком случае следует обратиться к классу `MultipleOutputs`, появившемуся в версии Hadoop 0.19 и предоставляющему более развитые возможности.

В классе `MultipleOutputs` принят иной подход, чем в `MultipleOutputFormat`. Вместо того чтобы запрашивать имя файла, в который выводить запись, `MultipleOutputs` создает несколько объектов `OutputCollector`. Каждый такой объект может содержать свой собственный объект `OutputFormat` и определять типы ключа и значения. Что выводить в каждый `OutputCollector`, решает ваша MapReduce-программа. В листинге 7.2 приведена программа, которая принимает на входе метаданные о патентах и выводит два набора данных. В один помещается хронологическая информация (дата выдачи), в другой – географическая. Это тоже программа с одними лишь распределителями, но применить несколько выходных коллекторов к редукторам элементарно.

Листинг 7.2.
Программа проецирования различных столбцов входных данных на разные файлы

```
public class MultiFile extends Configured implements Tool {

   public static class MapClass extends MapReduceBase
```

```
    implements Mapper<LongWritable, Text, NullWritable, Text> {

    private MultipleOutputs mos;
    private OutputCollector<NullWritable, Text> collector;

    public void configure(JobConf conf) {
       mos = new MultipleOutputs(conf);
    }

    public void map(LongWritable key, Text value,
                    OutputCollector<NullWritable, Text> output,
                    Reporter reporter) throws IOException {

       String[] arr = value.toString().split(",", -1);
       String chrono = arr[0] + "," + arr[1] + "," + arr[2];
       String geo = arr[0] + "," + arr[4] + "," + arr[5];

       collector = mos.getCollector("chrono", reporter);
       collector.collect(NullWritable.get(), new Text(chrono));
       collector = mos.getCollector("geo", reporter);
       collector.collect(NullWritable.get(), new Text(geo));
    }

    public void close() throws IOException {
       mos.close();
    }
}

public int run(String[] args) throws Exception {
    Configuration conf = getConf();

    JobConf job = new JobConf(conf, MultiFile.class);

    Path in = new Path(args[0]);
    Path out = new Path(args[1]);
    FileInputFormat.setInputPaths(job, in);
    FileOutputFormat.setOutputPath(job, out);

    job.setJobName("MultiFile");
    job.setMapperClass(MapClass.class);
    job.setInputFormat(TextInputFormat.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);
    job.setNumReduceTasks(0);

    MultipleOutputs.addNamedOutput(job,
                                   "chrono",
                                   TextOutputFormat.class,
                                   NullWritable.class,
                                   Text.class);
```

```
        MultipleOutputs.addNamedOutput(job,
                                       "geo",
                                       TextOutputFormat.class,
                                       NullWritable.class,
                                       Text.class);

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new MultiFile(),
                                 args);

        System.exit(res);
    }
}
```

Для работы с `MultipleOutputs` драйвер MapReduce-программы должен настроить выходные коллекторы, которые собирается использовать. Для создания коллекторов служит статический метод `addNamedOutput()` класса `MultipleOutputs`. Мы создали два выходных коллектора: *chrono* и *geo*. Оба настроены на формат `TextOutputFormat` и работают с одинаковыми типами ключей и значений, но можно было бы выбрать другие выходные форматы и типы данных.

После того как выходные коллекторы настроены в драйвере, нам необходим объект `MultipleOutputs`, который будет ими управлять; он создается в методе `configure()` на этапе инициализации распределителя. Этот объект должен существовать на протяжении всего времени работы распределителя. В самом методе `map()` мы вызываем метод `getCollector()` объекта `MultipleOutputs`, чтобы получить от него выходные коллекторы chrono и geo. Мы будем выводить различные данные, согласованные с каждым коллектором.

Поскольку мы присвоили имя каждому коллектору в `MultipleOutputs`, то `MultipleOutputs` сгенерирует соответствующие имена выходных файлов. Как именно `MultipleOutputs` генерирует имена, видно из распечатки списка файлов, созданных программой:

```
ls -l output/
total 101896
-rwxrwxrwx 1 Administrator None 9672703 Jul 31 06:28 chrono-m-00000
-rwxrwxrwx 1 Administrator None 7752888 Jul 31 06:29 chrono-m-00001
-rwxrwxrwx 1 Administrator None 6884496 Jul 31 06:29 chrono-m-00002
```

```
-rwxrwxrwx 1 Administrator None 6933561 Jul 31 06:29 chrono-m-00003
-rwxrwxrwx 1 Administrator None 7164558 Jul 31 06:29 chrono-m-00004
-rwxrwxrwx 1 Administrator None 7273561 Jul 31 06:29 chrono-m-00005
-rwxrwxrwx 1 Administrator None 8281663 Jul 31 06:29 chrono-m-00006
-rwxrwxrwx 1 Administrator None 9428951 Jul 31 06:28 geo-m-00000
-rwxrwxrwx 1 Administrator None 7464690 Jul 31 06:29 geo-m-00001
-rwxrwxrwx 1 Administrator None 6580482 Jul 31 06:29 geo-m-00002
-rwxrwxrwx 1 Administrator None 6448648 Jul 31 06:29 geo-m-00003
-rwxrwxrwx 1 Administrator None 6432392 Jul 31 06:29 geo-m-00004
-rwxrwxrwx 1 Administrator None 6546828 Jul 31 06:29 geo-m-00005
-rwxrwxrwx 1 Administrator None 7450768 Jul 31 06:29 geo-m-00006
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:28 part-00000
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:28 part-00001
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:29 part-00002
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:29 part-00003
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:29 part-00004
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:29 part-00005
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:29 part-00006
```

Как видите, имеются два набора файлов: с префиксами *chrono* и *geo*. Отметим, что программа по умолчанию создала выходные файлы *part-**, хотя явно мы ничего в них не писали. Но ничто не мешает производить запись в эти файлы с помощью объекта `OutputCollector`, переданного методу `map()`. На самом деле, если бы программа не ограничивалась одним лишь распределением, то редукторам для дальнейшей обработки были бы переданы записи, выведенные с помощью этого коллектора `OutputCollector`, и только эти записи.

Одно из ограничений класса `MultipleOutputs` по сравнению с `MultipleOutputFormat` – жесткая структура имен. Ваш выходной коллектор не должен называться *part*, потому что это имя уже зарезервировано для коллектора по умолчанию. Имена выходных файлов также фиксированы: имя выходного коллектора, затем буква *m* или *r* в зависимости от того, кто сгенерировал файл – распределитель или редуктор, – и в конце номер раздела.

```
head output/chrono-m-00000
"PATENT","GYEAR","GDATE"
3070801,1963,1096
3070802,1963,1096
3070803,1963,1096
3070804,1963,1096
3070805,1963,1096
3070806,1963,1096
3070807,1963,1096
3070808,1963,1096
3070809,1963,1096

head output/geo-m-00000
```

```
"PATENT","COUNTRY","POSTATE"
3070801,"BE",""
3070802,"US","TX"
3070803,"US","IL"
3070804,"US","OH"
3070805,"US","CA"
3070806,"US","PA"
3070807,"US","OH"
3070808,"US","IA"
3070809,"US","AZ"
```

Открыв парочку выходных файлов, мы увидим, что столбцы данных о патентах действительно спроецированы на разные файлы.

7.4. Ввод и вывод в базу данных

Хотя Hadoop полезен для обработки больших наборов данных, основным инструментом во многих приложениях остаются реляционные базы данных. И часто возникает необходимость в интерфейсе между Hadoop и базами данных.

В принципе можно настроить MapReduce-программу так, что она будет получать входные данные непосредственно из базы, а не из файла в HDFS, но производительность при этом будет далека от идеала. Чаще предварительно копируют данные из базы в HDFS. Сделать это позволяет стандартная утилита выгрузки базы данных, создающая плоский файл. Затем этот файл загружается в HDFS командой `put`.

Однако иногда бывает разумно записывать данные из MapReduce-программы напрямую в базу. Многие MapReduce-программы порождают в результате обработки больших наборов данных меньшие наборы, с которыми СУБД вполне способна справиться. Например, мы часто применяли MapReduce в ETL-процессах[1] для обработки гигантских файлов журналов с целью получения гораздо меньших по объему наборов статистических показателей, с которыми могли бы работать аналитики.

Для доступа к базам данных основную роль играет класс `DBOutputFormat`. Этот класс указывается в качестве выходного формата в драйвере. Необходимо также задать параметры соединения с базой данных, для чего служит статический метод `configureDB()` класса `DBConfiguration`:

```
public static void configureDB(JobConf job, String driverClass,
    ➥String dbUrl, String userName, String passwd)
```

[1] ETL (extraction, transform, load) – извлечение, преобразование, загрузка. *Прим. перев.*

Затем нужно указать таблицу, в которую вы собираетесь писать, и поля этой таблицы. Это делает статический метод `setOutput()` класса `DBOutputFormat`.

```
public static void setOutput(JobConf job, String tableName,
    ➥String... fieldNames)
```

В драйвере должны быть примерно такие строки:

```
conf.setOutputFormat(DBOutputFormat.class);
DBConfiguration.configureDB(job,
                            "com.mysql.jdbc.Driver",
                            "jdbc:mysql://db.host.com/mydb",
                            "username",
                            "password")
DBOutputFormat.setOutput(job, "Events", "event_id", "time");
```

Для использования класса `DBOutputFormat` тип выходного ключа должен реализовывать интерфейс `DBWritable`. В базу данных записывается только ключ. Ключи, как обычно, реализуют интерфейс `Writable`. Сигнатуры интерфейсов `Writable` и `DBWritable` похожи, различаются только типы аргументов. Метод `write()` интерфейса `Writable` принимает аргумент типа `DataOutput`, а одноименный метод интерфейса `DBWritable` – аргумент типа `PreparedStatement`. Аналогично в интерфейсе `Writable` метод `readFields()` принимает аргумент типа `DataInput`, а в интерфейсе `DBWritable` – аргумент типа `ResultSet`. Если вы не планируете выбирать данные непосредственно из базы с помощью формата `DBInputFormat`, то метод `readFields()` интерфейса `DBWritable` вызываться не будет.

```
public class EventsDBWritable implements Writable, DBWritable {
   private int id;
   private long timestamp;

   public void write(DataOutput out) throws IOException {
      out.writeInt(id);
      out.writeLong(timestamp);
   }

   public void readFields(DataInput in) throws IOException {
      id = in.readInt();
      timestamp = in.readLong();
   }

   public void write(PreparedStatement statement) throws SQLException {
      statement.setInt(1, id);
      statement.setLong(2, timestamp);
   }
```

```
    public void readFields(ResultSet resultSet) throws SQLException {
        id = resultSet.getInt(1);
        timestamp = resultSet.getLong(2);
    }
}
```

Мы хотим еще раз подчеркнуть, что чтение и запись в базы данных из Hadoop оправданы только для сравнительно небольших по меркам Hadoop наборов данных. Если ваша база данных не распараллелена в такой же мере, как Hadoop (это возможно, если кластер Hadoop относительно мал, а в СУБД организовано много секций), то СУБД станет узким местом, и вы не получите от кластера Hadoop никакого выигрыша в масштабируемости. Зачастую лучше организовать массовую пакетную загрузку в базу данных, а не писать в нее напрямую из Hadoop. Для очень больших баз данных придется придумывать нестандартные решения[2].

7.5. Сортировка выходных данных

Каркас MapReduce гарантирует, что данные, поступающие на вход редуктора, отсортированы по ключу. Часто редуктор производит только простые вычисления со значениями в каждой паре ключ/значение, и тогда выходные данные остаются отсортированными. Однако имейте в виду, что MapReduce не дает никаких гарантий отсортированности выходных данных; это всего лишь побочный эффект того, что входные данные отсортированы, и характера операций типичного редуктора.

Для некоторых приложений упорядоченность данных неважна, и иногда поднимается вопрос, а не стоит ли вообще отключить сортировку и избежать тем самым ненужного шага в редукторе. Но дело в том, что сортировка нужна не столько для обеспечения какой-то конкретной упорядоченности входных данных редуктора, сколько для эффективной группировки всех записей с одинаковым ключом. Если группировка несущественна, то можно напрямую порождать выходную запись из входной. В таком случае производительность можно повысить, исключив этап редукции целиком. Для этого достаточно

[2] В блоге на сайте LinkedIn есть интересное сообщение о проблемах, возникавших при переносе больших объемов данных, сгенерированных автономными процессами (то есть Hadoop) в работающие системы: http://project-voldemort.com/blog/2009/06/building-a-1-tb-data-cycle-at-linkedin-with-hadoop-and-project-voldemort/.

положить количество редукторов равным 0 и, стало быть, оставить в задаче только этап распределения.

С другой стороны, в некоторых приложениях желательна *глобальная* сортировка всех выходных данных. Каждый отдельный выходной файл (созданный одним редуктором) уже отсортирован, но было бы хорошо, чтобы все записи в файле `part-00000` были меньше записей в файле `part-00001`, а записи в `part-00001` меньше записей в `part-00002` и т. д. Сделать это можно, написав подходящий разбиватель.

Назначение разбивателя – детерминированное назначение редуктора каждому ключу. Все записи с одинаковым ключом группируются и обрабатываются редуктором совместно. Важно проектировать разбиватель так, чтобы нагрузка на редукторы была сбалансирована, – ни один редуктор не должен получать намного больше ключей, чем все остальные. Не имея априорной информации о распределении ключей, разбиватель по умолчанию применяет хеш-функцию, которая равномерно распределяет ключи между редукторами. С задачей равномерной загрузки редукторов такое решение справляется, но редуктор назначается случайно. Если мы заранее знаем, что ключи распределены приблизительно равномерно, то можно использовать разбиватель, который будет назначать каждому редуктору диапазоны ключей, не опасаясь, что нагрузка окажется несбалансированной.

Совет. Разбиватель, в котором применяется хеш-функция, может распределять нагрузку неравномерно, если для некоторых ключей время обработки существенно больше, чем для остальных. Например, в сильно асимметричных наборах данных может присутствовать много записей с одним и тем же ключом. По возможности для снижения нагрузки на редукторы следует использовать комбинатор, чтобы перенести как можно больше работы на этап распределения. Кроме того, можно написать специальный разбиватель, который будет распределять ключи неравномерно, так чтобы компенсировать изначальную асимметрию набора данных.

Класс `TotalOrderPartitioner` реализует разбиватель, обеспечивающий упорядоченность данных во всех выходных файлах, а не только внутри каждого в отдельности. При сортировке массивных данных (например, в эталонном тесте TeraSort) первоначально использовался похожий класс. Этот класс принимает файл последовательности, содержащий отсортированный набор ключей разделов и распределяет ключи разделов из разных диапазонов по редукторам.

7.6. Резюме

В этой главе мы рассмотрели различные инструменты и приемы, позволяющие сделать задачу Hadoop более дружественной к пользователю или улучшить интерфейс между ней и другими компонентами инфраструктуры обработки данных. Полностью средства, имеющиеся в распоряжении задачи Hadoop, описаны в документации по Hadoop API по адресу http://hadoop.apache.org/common/docs/current/api/index.html. Возможно, вам будет интересно познакомиться и с дополнительными абстракциями, упрощающими программирование, например Pig и Hive. Мы рассмотрим их в главах 10 и 11. Если ваша работа заключается в администрировании кластера Hadoop, то в следующей главе вы найдете полезные советы по этому поводу.

ГЛАВА 8. Администрирование Hadoop

В этой главе:

- Конфигурирование производственной системы.
- Обслуживание файловой системы HDFS.
- Настройка планировщика задач.

Приведенные в главе 2 инструкции по установке позволяют довольно быстро настроить и запустить кластер Hadoop. Эта конфигурация относительно проста, но, к сожалению, недостаточна для производственного кластера, постоянно испытывающего высокую нагрузку. Существуют разнообразные конфигурационные параметры для оптимизации производственного кластера, и в разделе 8.1 мы их рассмотрим.

Как и любая система, кластер Hadoop со временем претерпевает изменения, и администратор должен знать, как поддерживать его в «хорошей форме». Особенно это относится к файловой системе HDFS. В разделах 8.2–8.5 мы рассмотрим ряд стандартных задач обслуживания файловой системы: проверка состояния, установка прав доступа, задание квот и восстановление удаленных файлов (корзина). В разделах 8.6–8.10 обсуждаются более сложные, но редко выполняемые задачи, специфичные для HDFS: добавление и удаление узлов (управление емкостью) и восстановление после сбоя узла NameNode. И закончим главу разделом о настройке планировщика для управления запуском нескольких задач.

8.1. Практическая настройка параметров

У Hadoop много разных параметров. По умолчанию они настроены для работы в автономном режиме и к тому же стремятся обеспечить защиту «от дурака». Как правило, значений по умолчанию достаточно для работы системы без ошибок. Но в производственном кластере они зачастую далеко не оптимальны. В табл. 8.1 перечислены некоторые системные свойства, которые имеет смысл изменить для производственного кластера.

Таблица 8.1. Свойства Hadoop, которые полезно настроить для производственного кластера

Свойство	Описание	Рекомендуемое значение
`dfs.name.dir`	Каталог в локальной файловой системе узла NameNode для хранения метаданных HDFS	`/home/hadoop/dfs/name`
`dfs.data.dir`	Каталог в локальной файловой системе узла DataNode для хранения файловых блоков HDFS	`/home/hadoop/dfs/data`
`mapred.system.dir`	Каталог в HDFS для хранения разделяемых системных файлов MapReduce	`/hadoop/mapred/system`
`mapred.local.dir`	Каталог в локальной файловой системе узла TaskNode для хранения временных данных	
`mapred.tasktracker.{map|reduce}.tasks.maximum`	Максимальное число заданий map и reduce, одновременно обслуживаемых демоном TaskTracker	
`hadoop.tmp.dir`	Временные каталоги Hadoop	`/home/hadoop/tmp`
`dfs.datanode.du.reserved`	Минимальный объем свободного места, которым должен располагать узел DataNode	`1073741824`
`mapred.child.java.opts`	Размер кучи, выделяемой каждому заданию	`-Xmx512m`
`mapred.reduce.tasks`	Количество заданий reduce для одной задачи	

По умолчанию свойства `dfs.name.dir` и `dfs.data.dir` указывают на подкаталоги каталога `/tmp`, который почти во всех системах Unix предназначен для хранения временных данных. В производственном кластере эти свойства безусловно необходимо изменить[1]. В случае `dfs.name.dir` разумно организовать несколько каталогов для целей резервного копирования. Если узел DataNode оснащен несколькими дисками, то на каждом из них следует завести каталог для данных и перечислить их все в свойстве `dfs.data.dir`. Тогда DataNode будет параллельно использовать все каталоги для ускорения ввода/вывода[2]. В свойстве `mapred.local.dir` также следует прописать несколько каталогов, чтобы повысить скорость обработки временных данных.

Подразумеваемое по умолчанию значение свойства `hadoop.tmp.dir`, определяющего, где Hadoop хранит временные файлы, зависит от имени пользователя. Однако следует избегать любых зависимостей свойств Hadoop от имени пользователя, так как пользователь, отправляющий задачу, и пользователь, от имени которого запускается узел Hadoop, не обязаны совпадать. Выбирайте имя типа `/home/hadoop/tmp`, не зависящее от пользователя. Тот факт, что по умолчанию `hadoop.tmp.dir` указывает на каталог `/tmp`, также не сулит ничего хорошего. Хотя для хранения временных файлов этот каталог вполне подходит, но в большинстве дистрибутивов Linux определенная для него квота слишком мала для работы с Hadoop. Вместо того чтобы увеличивать квоту для каталога `/tmp`, лучше указать на другой каталог, в котором заведомо достаточно места.

По умолчанию HDFS не требует, чтобы на каждом узле DataNode было зарезервировано свободное место. Но на практике система, в которой свободного места на диске слишком мало, обычно становится нестабильной. Свойство `dfs.datanode.du.reserved` следует настроить так, чтобы в узлах DataNode было зарезервировано 1 ГБ свободного места.

---

1 Настройка по умолчанию являет пример «защиты от дурака». Поскольку каталог /tmp есть в любой системе Unix, то можно не опасаться возникновения ошибки «каталог не найден».

2 В форумах Hadoop была дискуссия о том, следует ли конфигурировать несколько жестких дисков в узле DataNode как RAID или как JBOD. Hadoop не нуждается в резервировании, обеспечиваемом RAID-массивами, поскольку HDFS и так реплицирует данные на несколько машин. Кроме того, Yahoo утверждает, что за счет использования JBOD удалось добиться заметного повышения производительности. В качестве причины указывается, что быстродействие жестких дисков, даже одной модели, имеет большой разброс. RAID-массив ограничивает производительность ввода/вывода скоростью самого медленного диска. С другой стороны, если каждый диск работает независимо от остальных, то его быстродействие будет максимально возможным, что увеличивает производительность системы в целом.

Если объем свободного места опускается ниже указанного порога, то DataNode перестает принимать запросы на запись блока.

Каждому узлу TaskTracker разрешено запускать некое максимальное количество заданий map и reduce; эту величину можно настраивать. По умолчанию предполагается четыре задания (два map и два reduce). Правильная величина зависит от многих факторов, хотя в большинстве случаев рекомендуют задавать одно или два задания на каждое процессорное ядро. Для четырехъядерной машины задается не более шести заданий (по три задания map и reduce), так как по одному заданию каждого вида уже выделено для TaskTracker и DataNode, так что в сумме получается восемь. Аналогично для машины с двумя четырехъядерными процессорами можно сконфигурировать не более четырнадцати заданий map и reduce. Эта оценка основана на предположении, что большинство задач MapReduce заняты в основном вводом/выводом. Если ожидается интенсивная процессорная обработка, то максимально допустимое количество заданий следует снизить.

При решении вопроса о допустимом количестве заданий следует также принимать во внимание объем памяти в куче, выделяемой каждому заданию. Принимаемая по умолчанию в Hadoop величина 200 МБ отнюдь не поражает воображение. Часто ее увеличивают до 512 МБ, а иногда даже до 1 ГБ. Это не окончательный вердикт. Любая задача может запросить больший (или меньший) размер кучи для своих заданий. Проверьте, что общий объем оперативной памяти в ваших компьютерах не меньше, чем затребовано в конфигурационных параметрах. Имейте в виду, что демоны DataNode и TaskTracker уже используют 1 ГБ памяти.

Хотя разрешается задавать количество редукторов отдельно для каждой задачи MapReduce, лучше определить значение по умолчанию, которое годится в большинстве случаев. Принятое в Hadoop соглашение об одном редукторе на каждую задачу, как правило, не оптимально. В общем случае рекомендуется задавать значение, равное максимальному количеству узлов TaskTracker, выполняющих редукцию, умноженному на 0,95 либо на 1,75. Это означает, что число заданий reduce в одной задаче должно быть равно произведению 0,95 или 1,75 на *количество рабочих узлов* и на величину `mapred.tasktracker.reduce.tasks.maximum`. Если коэффициент равен 0,95, то все задания reduce запускаются немедленно и начинают копировать порожденные распределителями результаты, как только те завершатся. Если же коэффициент равен 1,75, то некоторые редукторы запустятся сразу, а другие будут вынуждены ждать. Более быстрые

узлы закончат первый круг редукции раньше и приступят ко второму. Более медленным не придется обрабатывать задания reduce из второго круга. Тем самым достигается лучшее балансирование нагрузки.

8.2. Проверка состояния системы

В состав Hadoop входит утилита проверки файловой системы `fsck`. При ее вызове указывается путь к каталогу, а она рекурсивно проверяет состояние всех файлов, расположенных внутри этого каталога. Если указать путь /, то будет проверена вся файловая система. Ниже приведен пример итоговой распечатки:

```
bin/hadoop fsck /
Status: HEALTHY
Total size: 143106109768 B
Total dirs: 9726
Total files: 41532
Total blocks (validated): 42419 (avg. block size 3373632 B)
Minimally replicated blocks: 42419 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 8
Number of racks: 1
```

Большая часть строк не нуждается в пояснениях. По умолчанию `fsck` игнорирует файлы, открытые в данный момент для записи. Список таких файлов можно получить, запустив `fsck` с флагом `-openforwrite`.

По ходу проверки файловой системы `fsck` выводит одну точку для каждого обнаруженного неповрежденного файла (в примере выше не показаны). Для файлов с теми или иными дефектами, например с избыточным или недостаточным количеством реплик блоков, с неправильно реплицированными или поврежденными блоками, а также в случае отсутствия реплик, выводится сообщение. На избыточность, недостаточность или неправильность реплик можно не обращать особого внимания, потому что HDFS способна к самовосстановлению. По умолчанию `fsck` не предпринимает никаких действий в отношении поврежденных файлов, но, если запустить ее с флагом `-delete`, то файлы будут удалены. Еще лучше задать флаг `-move`, тог-

да поврежденные файлы будут перемещены в каталог `/lost+found` для последующего восстановления.

Можно попросить `fsck` выводить больше информации, задав флаги `-files`, `-blocks`, `-locations` или `-racks`. Каждый последующий флаг требует также задания всех предыдущих. Так, вместе с флагом `-blocks` необходимо задавать флаг `-files`, вместе с флагом `-locations` — оба флага `-files` и `-blocks` и т. д. Флаг `-files`, говорит, что для каждого проверяемого файла `fsck` должна печатать строку, содержащую путь к файлу, размер файла в байтах и блоках и его состояние. Флаг `-blocks` означает, что надо пойти еще дальше и печать информацию о каждом блоке в файле: имя блока, его длину и количество реплик. При задании флага `-locations` будет включена также информация о местоположении реплик блока. Флаг `-racks` добавляет в информацию о местоположении еще и имя стойки. Например, для короткого файла с одним блоком отчет будет выглядеть так:

```
bin/hadoop fsck /user/hadoop/test -files -blocks -locations -racks
/user/hadoop/test/part-00000 35792 bytes, 1 block(s): OK
0. blk_-4630072455652803568_97605 len=35792 repl=3
➥ [/default-rack/10.130.164.71:50010, /default-rack/10.130.164.177:50010,
➥ /default-rack/10.130.164.186:50010]
Status: HEALTHY
Total size:    35792 B
Total dirs:    0
Total files:   1
Total blocks (validated):     1 (avg. block size 35792 B)
Minimally replicated blocks:  1 (100.0 %)
Over-replicated blocks:       0 (0.0 %)
Under-replicated blocks:      0 (0.0 %)
Mis-replicated blocks:        0 (0.0 %)
Default replication factor:   3
Average block replication:    3.0
Corrupt blocks:               0
Missing replicas:             0 (0.0 %)
Number of data-nodes:         8
Number of racks:              1
```

Если `fsck` выводит информацию о каждом файле в системе HDFS, то утилита `dfsadmin` формирует отчет об узлах DataNode. Получить его можно, задав флаг `-report`:

```
bin/hadoop dfsadmin -report
Total raw bytes: 535472824320 (498.7 GB)
Remaining raw bytes: 33927731366 (31.6 GB)
Used raw bytes: 379948188541 (353.85 GB)
% used: 70.96%

Total effective bytes: 0 (0 KB)
```

```
Effective replication multiplier: Infinity
-------------------------------------------------
Datanodes available: 8

Name: 123.45.67.89:50010
State : In Service
Total raw bytes: 76669841408 (71.4 GB)
Remaining raw bytes: 2184594843(2.03 GB)
Used raw bytes: 56598956650 (52.71 GB)
% used: 73.82%
Last contact: Sun Jun 21 16:13:32 PDT 2009

Name: 123.45.67.90:50010
State : In Service
Total raw bytes: 76669841408 (71.4 GB)
Remaining raw bytes: 6356175381(5.92 GB)
Used raw bytes: 54220537856 (50.5 GB)
% used: 70.72%
Last contact: Sun Jun 21 16:13:33 PDT 2009

Name: 123.45.67.91:50010
State : In Service
Total raw bytes: 76669841408 (71.4 GB)
Remaining raw bytes: 6106387206(5.69 GB)
Used raw bytes: 52412190091 (48.81 GB)
% used: 68.36%
Last contact: Sun Jun 21 16:13:33 PDT 2009

...
```

Чтобы узнать о том, что сейчас делается на узле NameNode, следует вызвать `dfsadmin` с флагом `-metasave`:

```
bin/hadoop dfsadmin -metasave filename
```

В результате некоторые метаданные об узле NameNode будут сохранены в его каталоге журналов в файле с именем *filename*. Среди них вы найдете списки блоков, ожидающих репликации, реплицируемых в данный момент и ожидающих удаления. Для каждого реплицируемого блока перечисляются также узлы DataNode, на которые он реплицируется. Наконец, в файле метаданных сохраняется также сводная статистика для каждого узла DataNode.

8.3. Установка прав доступа

Для файловой системы HDFS реализована простая система прав доступа к файлам по аналогии с моделью POSIX. Для каждого файла определены девять разрешений: чтения (`r`), записи (`w`) и исполнения (`x`) для владельца, группы и всех остальных. Не все разрешения

осмысленны. Например, в HDFS нельзя исполнять файлы, поэтому разрешения `x` задавать запрещено.

Система разрешений для каталогов также следует модели POSIX. Разрешение `r` позволяет получать список объектов в каталоге, разрешение `w` – создавать и удалять файлы и каталоги, разрешение `x` – получать доступ к объектам, содержащимся в каталоге.

В современных версиях HDFS безопасности уделено не слишком много внимания. Систему разрешений следует использовать только для предотвращения случайных ошибок и перезаписи данных в предположении, что всем пользователям кластера Hadoop можно доверять. HDFS не аутентифицирует пользователей и полагает, что идентификатор пользователя действительно совпадает с тем, который сообщает объемлющая операционная система. Ваше имя пользователя в Hadoop совпадает с именем, под которым вы вошли в систему, – тем, что выводит утилита `whoami`. Список групп, в которые вы входите, – это список, печатаемый командой `bash -c groups`. Исключение составляет пользователь, запустивший узел NameNode. У него в Hadoop есть специальное имя *superuser*. Этот суперпользователь имеет право выполнять любые операции с файлами вне зависимости от заданных разрешений. Кроме того, администратор может определять, какие пользователи входят в супергруппу, с помощью параметра `dfs.permissions.supergroup`. Все члены супергруппы также являются суперпользователями.

Для изменения разрешений и принадлежности служат команды `bin/hadoop fs -chmod`, `-chown` и `-chgrp`. Они ведут себя так же, как одноименные команды Unix.

8.4. Управление квотами

По умолчанию в HDFS нет квот, ограничивающих общий объем данных в одном каталоге. Можно активировать и задать *квоту на имена* для отдельных каталогов, то есть ограничить количество имен файлов и подкаталогов, находящихся в этом каталоге. Основное назначение квоты на имена – воспрепятствовать созданию очень большого числа мелких файлов, что привело бы к перегрузке узла NameNode. Следующие команды устанавливают и отменяют квоту на имена:

```
bin/hadoop dfsadmin -setQuota <N> directory [...directory]
bin/hadoop dfsadmin -clrQuota directory [...directory]
```

Начиная с версии 0.19, HDFS поддерживает также квоту на пространство на уровне отдельных каталогов. Это позволяет контроли-

ровать, сколько места может занять данный пользователь или приложение.

```
bin/hadoop dfsadmin -setSpaceQuota <N> directory [...directory]
bin/hadoop dfsadmin -clrSpaceQuota directory [...directory]
```

Команда `setSpaceQuota` принимает в качестве аргумента количество *байтов*, интерпретируемое как квота для указанного каталога. Число может сопровождаться суффиксом, обозначающим единицу измерения. Например, `20g` означает 20 гигабайт, а `5t` – 5 терабайт. Реплики также включаются в квоту.

Чтобы узнать, какие квоты заданы для каталога, а заодно вывести количество имен и байтов в нем, воспользуйтесь командой HDFS `count` с флагом `-q`:

```
bin/hadoop fs -count -q directory [...directory]
```

8.5. Включение корзины

Помимо разрешений, дополнительной защитой от случайного удаления файлов в HDFS служит корзина. По умолчанию эта функция отключена. Если же включить ее, то командные утилиты, предназначенные для удаления файлов, не удаляют их окончательно, а сначала перемещают в каталог .Trash/, откуда файл можно восстановить, скопировав туда, где он находился изначально.

Чтобы включить корзину и задать время, в течение которого файл находится в корзине до окончательного удаления, установите свойство `fs.trash.interval` в файле core-site.xml, записав в него срок хранения в минутах. Например, чтобы дать пользователю 24 часа (1440 минут) на отмену решения об удалении файла, в core-site.xml следует включить такие строки:

```
<property>
  <name>fs.trash.interval</name>
  <value>1440</value>
</property>
```

Значение 0 отключает корзину.

8.6. Удаление узлов DataNode

В какой-то момент может возникнуть необходимость удалить из кластера узлы DataNode, например, чтобы перевести машину в автономный режим для модернизации или технического

обслуживания. Удалить узлы можно методом «грубой силы». Хотя это и не рекомендуется, но узел можно просто отключить от кластера или остановить работающий на нем демон. HDFS спроектирована так, чтобы продолжать функционирование в таких условиях. Изъятие одного или двух узлов не сильно скажется на текущей работе. Узел NameNode обнаружит выход узлов из строя и инициирует репликацию блоков, для которых количество реплик оказалось ниже сконфигурированного коэффициента репликации. Но чтобы обезопасить себя, особенно в случае, когда изымается много узлов DataNode, следует воспользоваться встроенным в Hadoop механизмом демонтажа. Он гарантирует, что для всех блоков количество реплик на оставшихся узлах будет равно коэффициенту репликации. Чтобы активировать этот механизм, следует создать (первоначально пустой) файл исключений в локальной файловой системе узла NameNode и прописать путь к нему в конфигурационном параметре `dfs.hosts.exclude` до запуска NameNode. Чтобы демонтировать узлы DataNode, перечислите их в файле исключений по одному узлу в строке. Узел задается полным именем или IP-адресом хоста или в виде IP:порт. Затем выполните команду

```
bin/hadoop dfsadmin -refreshNodes
```

которая заставляет демон NameNode перечитать файл исключений и запустить процесс демонтажа. По завершении демонтажа в журналы NameNode будут занесены сообщения вида «Decommission complete for node 172.16.1.55:50010», и в этот момент узел можно исключать из кластера.

Если в момент запуска HDFS файла свойство `dfs.hosts.exclude` не указывало на файл исключений, то для демонтажа узлов DataNode следует поступить следующим образом. Остановите демон NameNode. Пропишите в свойстве `dfs.hosts.exclude` путь к *пустому* файлу исключений. Снова запустите NameNode. После успешного перезапуска NameNode выполните описанную выше процедуру. Отметим, что если перечислить в файле исключений демонтируемые узлы DataNode до перезапуска NameNode, то NameNode будет сбит с толку и запишет в свои журналы сообщения вида «ProcessReport from unregistered node: node055:50010». NameNode считает, что к нему обратился узел DataNode, не являющийся частью системы, а не узел, подлежащий демонтажу.

Если впоследствии демонтированную машину понадобится снова включить в кластер, то нужно будет удалить ее имя из файла исключений и повторно запустить команду `bin/hadoop dfsadmin`

`-refreshNodes`, которая известит NameNode об изменениях. Когда машина будет готова к «воссоединению» с кластером, добавьте ее, следуя процедуре, описанной в следующем разделе.

8.7. Добавление узлов DataNode

Иногда нужно не только вернуть в кластер машину, прошедшую техобслуживание, но и добавить новые узлы DataNode, поскольку появились новые задачи или увеличился объем обрабатываемых данных. Установите на новый узел Hadoop и настройте конфигурационные файлы, как на любом другом узле DataNode. Вручную запустите демон DataNode (`bin/hadoop datanode`). Он автоматически свяжется в узлом NameNode и присоединится к кластеру. Новый узел нужно также добавить в файл conf/slaves на главном сервере.

Вновь добавленный узел DataNode первоначально пуст, тогда как существующие узлы частично заполнены. Такая файловая система называется *несбалансированной*. Новые файлы будут попадать на новый узел, но их реплицированные блоки будут по-прежнему распределяться между старыми узлами. Для достижения оптимальной производительности кластера нужно самостоятельно запустить балансировщик HDFS:

```
bin/start-balancer.sh
```

Этот скрипт будет работать в фоновом режиме, пока кластер не окажется сбалансированным. При желании администратор может снять его раньше, выполнив команду

```
bin/stop-balancer.sh
```

Кластер считается сбалансированным, когда коэффициент заполненности дисков на всех узлах DataNode отличается от среднего не более чем на пороговую величину. По умолчанию порог составляет 10 процентов. При запуске скрипта балансирования можно указать другое значение. Например, чтобы получить более равномерно сбалансированный кластер с порогом 5 процентов, выполните команду

```
bin/start-balancer.sh -threshold 5
```

Поскольку процесс балансирования потребляет много сетевых ресурсов, мы рекомендуем запускать его ночью или в выходные, когда кластер меньше загружен. Или можно задать конфигурационный параметр `dfs.balance.bandwidthPerSec`, чтобы ограничить полосу пропускания, отведенную под балансирование.

8.8. Управление узлами NameNode и Secondary NameNode

NameNode – один из важнейших компонентов архитектуры HDFS. На этом узле хранятся метаданные о файловой системе и карта блоков кластера, которая для повышения производительности кэшируется в памяти. Если ваш кластер хоть сколько-нибудь велик, то следует отвести под NameNode отдельную машину и не размещать на ней никаких других узлов – DataNode, JobTracker или TaskTracker. Это должна быть самая мощная машина во всем кластере с максимально возможным объемом оперативной памяти. Хотя узлы DataNode могут достигать более высокой производительности, когда диски объединены в массив JBOD, узел NameNode безусловно должен быть оснащен RAID-массивом для защиты от отказа одного диска.

Один из путей снижения нагрузки на узел NameNode состоит в том, чтобы уменьшить объем метаданных о файловой системе за счет увеличения размера блока. При удвоении размера блока объем метаданных снижается почти вдвое. Но к сожалению, одновременно уменьшается степень параллелизма для относительно небольших файлов. Идеальный размер блока зависит от конкретных условий работы вашего кластера. Размер блока задается с помощью конфигурационного параметра `dfs.block.size`. Например, чтобы увеличить его с подразумеваемых по умолчанию 64 МБ до 128 МБ, задайте `dfs.block.size` равным 134217728.

По умолчанию узлы Secondary NameNode[3] и NameNode работают на одной машине. Для кластеров среднего размера (10 и более узлов) следует разместить Secondary NameNode на отдельной машине, по мощности сравнимой с той, где работает NameNode. Но прежде чем описывать конфигурирование отдельного сервера в качестве Secondary NameNode, я должен объяснить, что делает и чего не делает этот узел, а это в свою очередь потребует знания некоторых механизмов работы узла NameNode.

Из-за неудачного названия часто ошибочно думают, что узел Secondary NameNode (SNN) играет роль резервного ресурса на случай

[3] Когда писалась эта книга, было принято решение отказаться от узла Secondary NameNode в версии Hadoop 0.21, которая должна выйти, когда книга поступит в продажу. Вместо него будет реализована более надежная конструкция, обеспечивающая работу в режиме горячего резерва. Чтобы узнать, используется ли в вашей версии Hadoop узел Secondary NameNode или уже нет, обратитесь к онлайновой документации. Код этого изменения находится по адресу https://issues.apache.org/jira/browse/HADOOP-4539.

выхода из строя NameNode. Это ни в коем случае не так. У SNN только одно назначение – периодически очищать и сжимать информацию о состоянии файловой системы, хранящуюся в NameNode, помогая тем самым узлу NameNode работать более эффективно. NameNode хранит эту информацию в двух файлах: `FsImage` и `EditLog`. Файл `FsImage` представляет собой снимок файловой системы в некоей контрольной точке, а в файле `EditLog` находятся записи обо всех изменениях (*дельтах*), произошедших в файловой системе с момента снятия последней контрольной точки. В совокупности эти два файла полностью определяют текущее состояние файловой системы. На этапе инициализации NameNode оба файла объединяются и создается новый снимок. В конце инициализации `FsImage` содержит новый снимок, а `EditLog` пуст. После любой операции, изменяющей состояние HDFS, информация об изменении добавляется в `EditLog`, тогда как `FsImage` остается тем же самым. После останова и перезапуска NameNode снова производится консолидация и формируется новый снимок. Отметим, что эти файлы существуют лишь для сохранения информации о файловой системе в то время, когда узел NameNode не работает (либо остановлен намеренно, либо произошел системный сбой). А во время работы NameNode хранит всю информацию о файловой системе в памяти, чтобы быстро отвечать на запросы о ее состоянии.

В сильно загруженном кластере файл `EditLog` становится очень большим, поэтому для объединения `EditLog` и `FsImage` при следующем запуске NameNode требуется длительное время. Но и перезапуск загруженного кластера производится реже, поэтому желательно делать снимки более часто. Вот для этого и предназначен узел SNN. Он объединяет `FsImage` и `EditLog` в новый снимок, оставляя на долю NameNode обслуживание текущего потока запросов. Поэтому правильнее представлять себе SNN как сервер контрольных точек. Для объединения `FsImage` и `EditLog` требуется много памяти, примерно столько же, сколько для обычных операций NameNode. Поэтому лучше выделять для SNN отдельный сервер, не менее мощный, чем основной узел NameNode.

Чтобы сконфигурировать HDFS для работы с отдельным сервером SNN, сначала пропишите доменное имя или IP-адрес этого сервера в файле `conf/masters`. К сожалению, имя этого файла способно сбить с толку. Главными серверами (masters) в Hadoop называются узлы (NameNode и JobTracker), на которых запущен скрипт `bin/start-dfs.sh` или `bin/start-mapred.sh`. А в файле `conf/masters` указан SNN, а вовсе не «master».

Необходимо также модифицировать файл conf/hdfs-site.xml на узле SNN, так чтобы свойство `dfs.http.address` указывало на порт 50070 сервера NameNode:

```
<property>
  <name>dfs.http.address</name>
  <value>namenode.hadoop-host.com:50070</value>
</property>
```

Это свойство необходимо установить, потому что SNN получает файлы `FsImage` и `EditLog` от узла NameNode, посылая HTTP-запрос типа Get на следующие URL-адреса:

- *FsImage* – http://*namenode.hadoop-host.com*:50070/getimage?getimage=1
- *EditLog* – http://*namenode.hadoop-host.com*:50070/getimage?getedit=1

Для возврата объединенных метаданных на узел NameNode SNN обращается к тому же адресу и порту.

8.9. Восстановление после сбоя узла NameNode

Отказы случаются, тут ничего не поделаешь, и Hadoop спроектирован с учетом этого обстоятельства. Но, к сожалению, узел NameNode остается слабым звеном. Если он выходит из строя, то HDFS оказывается неработоспособной. Типичное решение для резервирования сервера NameNode – задействовать SNN[4]. В конце концов, SNN по мощности сопоставим с NameNode, и Hadoop по идее уже установлен на обоих машинах со сходными конфигурациями каталогов. Если проделать небольшую дополнительную работу по организации на SNN функционального зеркала NameNode, то можно будет быстро запустить новый экземпляр NameNode в случае отказа основного узла. Для запуска NameNode на резервном узле потребуется ручное вмешательство и некоторое время, но, по крайней мере, мы не потеряем никаких данных.

NameNode хранит все метаданные о файловой системе, в том числе файлы `FsImage` и `EditLog` в каталоге `dfs.name.dir`. Отметим, что сервер SNN этот каталог вообще не использует. Он загружает системные метаданные в каталог `fs.checkpoint.dir` и там же объединя-

[4] К сожалению, это типичное решение вносит свой вклад в неправильное понимание назначения Secondary NameNode. Резервный узел можно организовать и на машине, отличной от NameNode и SNN, но такая машина большую часть времени будет простаивать.

ет файлы `FsImage` и `EditLog`. Поскольку каталог `dfs.name.dir` на сервере SNN не используется, мы можем экспортировать его на узел NameNode с помощью сетевой файловой системы (NFS). Мы настроим NameNode так, чтобы он писал в этот смонтированный каталог в дополнение к записи в локальный каталог с метаданными. HDFS поддерживает возможность писать метаданные в несколько каталогов. Нужно лишь перечислить в свойстве `dfs.name.dir` на узле NameNode несколько каталогов через запятую:

```
<property>
  <name>dfs.name.dir</name>
  <value>/home/hadoop/dfs/name,/mnt/hadoop-backup</value>
  <final>true</final>
</property>
```

Такая конфигурация будет работать в предположении, что свойство `dfs.name.dir` на узлах NameNode и Secondary NameNode указывает на `/home/hadoop/dfs/name`, причем этот каталог в узле SNN смонтирован на точку `/mnt/hadoop-backup` на узле NameNode. Когда HDFS видит в свойстве `dfs.name.dir` список путей, разделенных запятыми, она пишет метаданные в каждый из указанных в списке каталогов. Теперь, если NameNode выходит из строя, содержимое каталогов на узле NameNode и резервном узле (SNN) должно быть одинаково. Чтобы резервный узел мог выступать в роли замены NameNode, необходимо подменить его IP-адрес адресом исходного NameNode. (К сожалению, подменить одно лишь имя хоста недостаточно, потому что узлы DataNode кэшируют записи DNS.) Кроме того, нужно будет превратить резервный узел в NameNode, запустив на нем скрипт `bin/start-dfs.sh`.

Для большей безопасности, этот новый NameNode *также* должен иметь резервный узел и настроить его необходимо перед запуском. В противном случае возникнут проблемы, если новый NameNode тоже откажет. Если нет машины, которую можно было бы использовать как резервную, то по крайней мере следует организовать каталог, монтируемый по NFS. Тогда информация о состоянии файловой системы будет храниться в нескольких местах.

Поскольку HDFS записывает метаданные во все каталоги, перечисленные в свойстве `dfs.name.dir`, то при условии, что NameNode оснащен несколькими дисками, можно назначить каталоги на разных дисках для хранения реплик метаданных. Тогда если один диск выйдет из строя, то будет проще перезапустить NameNode, отключив дефектный диск, чем переключаться на резервный узел со всеми со-

путствующими этому хлопотами: сменой IP-адреса, настройкой нового резервного узла и т. д.

Напомним, что SNN создает снимок метаданных файловой системы в каталоге `fs.checkpoint.dir`. Поскольку контрольная точка снимается периодически (по умолчанию раз в час), то полагаться на устаревшие метаданные при переключении на резервный узел не стоит. Тем не менее, все равно рекомендуется время от времени архивировать этот каталог в удаленной системе хранения. В случае катастрофы лучше уж восстановиться с устаревшей копии, чем вообще потерять данные. Это относится и к случаю, когда и NameNode, и резервный узел выходят из строя одновременно (например, после всплеска напряжения, затронувшего обе машины). Еще один печальный сценарий – повреждение метаданных о файловой системе (к примеру, из-за человеческой оплошности или ошибки в программном обеспечении), сделавшее бесполезными все реплики. Процедура восстановления с образа контрольной точки описана на странице http://issues.apache.org/jira/browse/HADOOP-2585.

Механизм резервного копирования и восстановления HDFS продолжает активно разрабатываться и совершенствоваться. Последние новости смотрите в онлайновой документации по HDFS. Имеются также примеры применения специализированного программного обеспечения Linux, к примеру DRBD (Distributed Replicated Block Device – распределённое реплицируемое блочное устройство)[5] для построения кластеров Hadoop с высокой степенью доступности. См. в частности http://www.cloudera.com/blog/2009/07/22/hadoop-ha-configuration/.

8.10. Проектирование топологии сети и осведомленность о стойках

По мере роста кластера Hadoop растет и количество стоек, в которых находятся узловые компьютеры, а на надежность и производительность начинает влиять топология сети. Крайне желательно, чтобы кластер продолжал функционировать и после выхода из строя целой стойки. Для этого следует поместить резервный сервер, описанный в предыдущем разделе, в стойку, отличную от той, где находится сам узел NameNode. Тогда в случае отказа одной стойки пострадают не все копии метаданных о файловой системе.

При наличии нескольких стоек размещение реплик блоков и заданий усложняется. Реплики должны храниться на компьютерах из

[5] http://www.drbd.org

разных стоек, чтобы свести к минимуму возможность потери данных. Для стандартного коэффициента репликации 3 по умолчанию применяется следующая политика записи блока: если клиент, выполняющий операцию записи, является частью кластера Hadoop, то первая реплика помещается на тот узел DataNode, где находится клиент. В противном случае для хранения этой реплики выбирается случайный узел. Вторая реплика сохраняется в случайно выбранной стойке, отличной от той, где была сохранена первая. Третья реплика записывается на другой узел в той же стойке, где сохранена вторая реплика. Если коэффициент репликации больше 3, то последующие реплики записываются на случайно выбранные узлы. На момент работы над этой книгой описанная политика размещения блоков «зашита» в код NameNode. В версии 0.21 намечено реализовать сменную политику[6].

Помимо размещения блоков, информация о стойках учитывается также в механизме размещения заданий. Обычно задание помещается на тот узел, где находится копия блока, который данное задание должно обрабатывать. Если ни один узел, удовлетворяющий этому условию, не готов принять новое задание, то для его размещения случайным образом выбирается узел в той стойке, где имеется хотя бы одна копия блока. Таким образом, если локальность данных невозможно обеспечить на уровне узла, то Hadoop пытается обеспечить ее хотя бы на уровне стойки. Если и это не получается, то для задания выбирается какой-то из оставшихся узлов.

Возможно, у вас возник вопрос, откуда Hadoop знает, в какой стойке находится узел. Ему надо об этом сказать. Предполагается, что сеть в кластере Hadoop построена иерархически, как показано на рис. 8.1. С каждым узлом связано имя стойки, аналогичное пути в файловой системе. Например, для узлов H1, H2 и H3 на рис. 8.1 имя стойки `/D1/R1`. На рисунке показаны два центра обработки данных (D1 и D2), в каждом из которых несколько стоек (R1—R4). В большинстве случаев стойки находятся в одном ЦОД, тогда они имеют одноуровневые имена, например `/R1` и `/R2`.

Чтобы Hadoop узнал о местоположении каждого узла, вы должны написать исполняемый скрипт, сопоставляющий IP-адреса именам стоек. Этот *скрипт сетевой топологии* должен находиться на главном узле, а путь к нему прописывается в свойстве `topology.script.file.name` в файле `core-site.xml`. Hadoop будет вызывать этот скрипт, передавая IP-адреса в виде отдельных аргументов командной

[6] Описание этого изменения см. на странице http://issues.apache.org/jira/browse/HDFS-385.

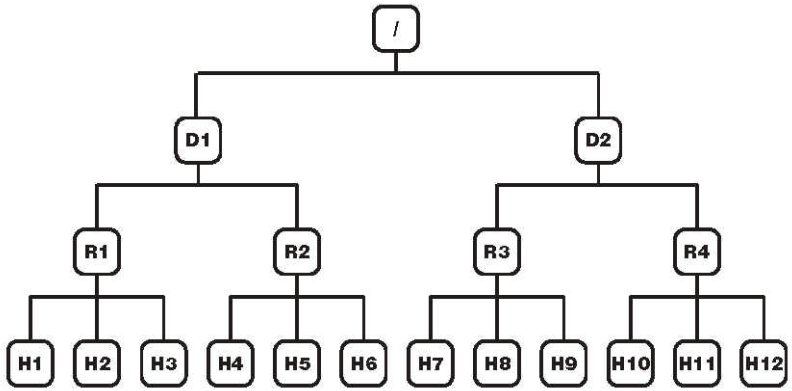


Рис. 8.1. Кластер с иерархической топологией сети, который охватывает два центра обработки данных (D1 и D2). В каждом ЦОД имеется несколько стоек (R), а в каждой стойке – несколько компьютеров.

строки. А скрипт должен вывести на STDOUT имена стоек, соответствующих каждому адресу, в том же порядке, разделив их пробелами. С помощью свойства `topology.script.number.args` задается максимальное количество IP-адресов, информацию о которых Hadoop может запросить в одном обращении. Для упрощения скрипта удобно задать это значение равным 1. Ниже приведен пример скрипта сетевой топологии.

```
#!/bin/bash

ipaddr=$1
segments='echo $ipaddr | cut --delimiter=. --fields=4'
if [ "$segments" -lt 128 ]; then
   echo /rack-1
else
   echo /rack-2
fi
```

Это bash-скрипт принимает один IPv4-адрес и анализирует последний из четырех октетов (в предположении, что адрес записан в точечно-десятичной нотации). Считается, что узел находится в стойке 1, если последний октет его адреса меньше 128, а в противном случае – в стойке 2. Для кластеров с более сложной топологией, задачу, наверное, лучше решать с помощью поиска в таблице. С другой стороны, если скрипт сетевой топологии не задан, то Hadoop предполагает, что топология плоская, то есть все узлы находятся в стойке `/default-rack`.

8.11. Планирование задач, поступающих от нескольких пользователей

По мере того как растет количество задач, поступающих в кластер Hadoop от разных пользователей, возникает необходимость в стратегии предотвращения конкуренции. Встроенный в Hadoop FIFO-планировщик обслуживает задачи в порядке поступления: получив очередную задачу, JobTracker назначает ей столько узлов TaskTrackers, сколько необходимо для обработки. Эта стратегия работает удовлетворительно, пока кластер не очень загружен и обрабатывающих мощностей достаточно. Но большие задачи могут занять кластер на длительное время, и тогда более мелкие задачи будут вынуждены ждать. Было бы здорово, если бы для мелких задач существовало что-то вроде экспресс-кассы.

8.11.1. Организация нескольких узлов JobTracker

До выхода версии Hadoop 0.19 для реализации рудиментарного механизма распределения ЦП между задачами необходимо было физически настраивать несколько кластеров MapReduce. Но чтобы обеспечить приемлемую эффективность системы хранения, кластер HDFS по-прежнему был один. Предположим, что в кластере Hadoop имеется Z подчиненных узлов и есть один узел NameNode, который рассматривает все Z узлов как DataNode. Кроме того, все Z узлов выполняют функции TaskTracker. До сих пор мы считали, что все TaskTracker'ы указывают на один и тот же (он же единственный) узел JobTracker.

Идея многокластерной конфигурации заключается в том, чтобы настроить несколько узлов JobTracker, так чтобы каждый управлял своим подмножеством узлов TaskTracker, причем эти подмножества не должны пересекаться. Например, для создания двух кластеров MapReduce требуется, чтобы X TaskTracker'ов указывали на один JobTracker (с помощью свойства `mapred.job.tracker`), а Y TaskTracker'ов – на другой. Подчиненные узлы в двух кластерах разделены, а их общее число X+Y=Z. При работе с такой конфигурацией часть задач отправляется одному узлу JobTracker, а часть – другому. В результате ограничивается количество узлов TaskTracker,

доступных задачам каждого типа. Назначение того или иного пула узлов задаче необязательно диктуется ее типом. Чаще пул выделяется какой-то группе пользователей. Тем самым лимитируется объем ресурсов, доступных этой группе.

Описанный способ физической настройки нескольких кластеров MapReduce имеет много недостатков. Он неудобен для пользователей, которые должны помнить, какой пул использовать. Вероятность, что задание окажется размещенным там же, где данные, тоже снижается (может оказаться, что все реплики находятся в узлах DataNode, не попавших в выбранный пул). Этой конфигурации недостает гибкости при изменении требований к ресурсам. К счастью, начиная с версии 0.19, архитектура Hadoop предусматривает замену планировщиков, а для решения проблемы конкуренции между задачами появились два новых планировщика. Один из них – *справедливый планировщик* (Fair Scheduler), разработанный в Facebook, а другой – *планировщик по емкости* (Capacity Scheduler), разработанный в Yahoo.

8.11.2. Справедливый планировщик

В справедливом планировщике введено понятие *пула*. Каждая задача помечается признаком, относящим его к конкретному пулу, а в каждом пуле конфигурируется гарантированное число слотов для заданий map и для заданий reduce. При освобождении слотов заданий справедливый планировщик распределяет их, принимая во внимание, прежде всего, эти минимальные гарантии. После того как гарантии соблюдены, слоты выделяются задачам в соответствии со стратегией «справедливого доступа к ресурсам», то есть так, чтобы каждая задача получала примерно равный объем вычислительных ресурсов. Задачам можно назначать приоритеты, тогда более приоритетные задачи получат больше ресурсов (все задачи равны, но некоторые равнее других).

Справедливый планировщик находится в jar-файле `hadoop-*-fairscheduler.jar` в каталоге `contrib/fairscheduler` инсталляции Hadoop. Для его установки можно просто переместить этот jar-файл в каталог `lib/` дерева Hadoop. Вместо этого можно изменить переменную `HADOOP_CLASSPATH` в скрипте `conf/hadoop-env.sh`, так чтобы она включала путь к этому jar-файлу.

Чтобы сконфигурировать и активировать справедливый планировщик, необходимо установить несколько свойств в файле `hadoop-site.xml`. Сначала следует присвоить свойству `mapred.`

jobtracker.taskScheduler значение org.apache.hadoop.mapred.FairScheduler, чтобы Hadoop использовал справедливый планировщик вместо подразумеваемого по умолчанию. Затем нужно настроить несколько свойств самого справедливого планировщика. Самое важное из них, mapred.fairscheduler.allocation.file, указывает на файл, в котором определены различные пулы. Обычно этот файл называется pools.xml, в нем для каждого пула указаны имя и емкость. Свойство mapred.fairscheduler.poolnameproperty определяет, какое свойство jobconf должен использовать планировщик, когда ему нужно назначить задаче пул. Рекомендуется задавать имя какого-нибудь нового свойства, скажем pool.name, и по умолчанию присваивать pool.name значение ${user.name}. Справедливый планировщик автоматически выделяет каждому пользователю индивидуальный пул. При таком задании pool.name каждой задаче по умолчанию назначается пул ее владельца. Но при желании можно присвоить свойству pool.name в конфигурационном объекте задачи jobconf значение, определяющее другой пул[7]. Наконец, если свойство mapred.fairscheduler. assignmultiple равно true, то планировщик будет одновременно назначать задания map и reduce в каждом цикле планирования, что повышает скорость выхода на рабочий режим и общую пропускную способность. Итак, в файле mapred-site.xml должны быть установлены следующие свойства:

```
<property>
  <name>mapred.jobtracker.taskScheduler</name>
  <value>org.apache.hadoop.mapred.FairScheduler</value>
</property>
<property>
  <name>mapred.fairscheduler.allocation.file</name>
  <value>HADOOP_CONF_DIR/pools.xml</value>
</property>
<property>
  <name>mapred.fairscheduler.assignmultiple</name>
  <value>true</value>
</property>
<property>
  <name>mapred.fairscheduler.poolnameproperty</name>
  <value>pool.name</value>
</property>
<property>
  <name>pool.name</name>
```

[7] Да, разрешается запускать свою задачу в чужом пуле, но это не очень вежливо. Обычно рекомендуется назначать специальным задачам специальные пулы. Например, все задачи, запускаемые через cron, можно ассоциировать с отдельным пулом, а не с пулом конкретного пользователя.

```
  <value>${user.name}</value>
</property>
```

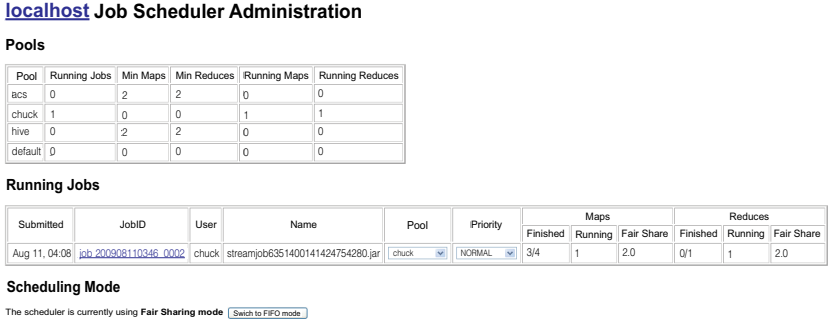
В файле `pools.xml` определяются пулы для планировщика. Для каждого пула задается имя и ограничения по емкости. В состав ограничений входит минимальное число слотов для заданий map или reduce. Можно также задать максимальное число исполняемых задач и максимальное число исполняемых задач, запущенных одним пользователем, причем последний параметр можно переопределять для конкретных пользователей. Ниже приведен пример файла `pools.xml`:

```
<?xml version="1.0"?>
<allocations>
  <pool name="ads">
    <minMaps>2</minMaps>
    <minReduces>2</minReduces>
  </pool>
  <pool name="hive">
    <minMaps>2</minMaps>
    <minReduces>2</minReduces>
    <maxRunningJobs>2</maxRunningJobs>
  </pool>
  <user name="chuck">
    <maxRunningJobs>6</maxRunningJobs>
  </user>
  <userMaxJobsDefault>3</userMaxJobsDefault>
</allocations>
```

В этом файле определены два специальных пула: «ads» и «hive». Гарантируется, что в каждом будет по меньшей мере два слота map и два слота reduce. В пуле «hive» не может одновременно выполняться более двух задач. Чтобы воспользоваться этими пулами, необходимо присвоить свойству `pool.name` в конфигурационном объекте задачи значение «ads» или «hive». Кроме того, в файле `pools.xml` прописано, что каждый пользователь может одновременно запускать не более трех задач, но для пользователя «chuck» лимит повышен до шести.

Отметим, что файл `pools.xml` перечитывается каждые 15 секунд, поэтому его можно отредактировать и тем самым динамически переопределить емкости, не останавливая узел. Для пулов, не описанных в этом файле, емкость не гарантирована, а ограничений на число одновременно выполняемых задач не накладывается.

Если в кластере работает справедливый планировщик, то в вашем распоряжении имеется веб-интерфейс для управления им. Он находится по адресу http://<jobtracker URL>/scheduler и позволяет не только узнать, как планируются задачи, но и изменить пул, назна-



localhost Job Scheduler Administration

Pools

Pool	Running Jobs	Min Maps	Min Reduces	Running Maps	Running Reduces
acs	0	2	2	0	0
chuck	1	0	0	1	1
hive	0	2	2	0	0
default	0	0	0	0	0

Running Jobs

Submitted	JobID	User	Name	Pool	Priority	Maps			Reduces		
						Finished	Running	Fair Share	Finished	Running	Fair Share
Aug 11, 04:08	job_200908110346_0002	chuck	streamjob6351400141424754280.jar	chuck	NORMAL	3/4	1	2.0	0/1	1	2.0

Scheduling Mode

The scheduler is currently using **Fair Sharing mode** Swich to FIFO mode

Рис. 8.2. Веб-интерфейс для мониторинга справедливого планировщика. В верхней таблице показаны все имеющиеся пулы и их текущее использование. В таблице «Running Jobs» имеется столбец Pool, в котором можно увидеть или изменить пул, назначенный каждой исполняемой задаче.

ченный задаче, и ее приоритет. Внешний вид интерфейса показан на рис. 8.2.

Планировщик по емкости решает те же задачи, что и справедливый планировщик, только работает не с *пулами*, а с *очередями*. Интересующийся читатель может найти дополнительные сведения об этом планировщике в онлайновой документации по адресу http://hadoop.apache.org/common/docs/r0.20.0/capacity_scheduler.html.

8.12. Резюме

Управление любыми распределенными кластерами – непростая задача, и Hadoop – не исключение. В этой главе мы рассмотрели ряд типичных задач администрирования. Если конфигурация вашего кластера особенно сложна и вы не нашли здесь ответов на свои вопросы, то можете обратиться к спискам рассылки Hadoop[8]. В этих списках принимают активное участие многие администраторы Hadoop с богатым опытом, и велики шансы, что кто-то уже сталкивался с подобной ситуацией. С другой стороны, если вам нужен простой кластер Hadoop без всех хлопот, связанных с администрированием, то можете взять дистрибутив Cloudera[9] или подумать об облачных службах, которые будут рассмотрены в следующей главе.

8 http://hadoop.apache.org/common/mailing_lists.html

9 http://www.cloudera.com/distribution

Часть 3

Hadoop в реальной жизни

В части 3 рассматриваются крупные системы, построенные на базе Hadoop. Облачные службы представляют собой альтернативу покупке собственного оборудования и организации кластера Hadoop на собственной площадке. Существует много дополнительных пакетов, предоставляющих высокоуровневые абстракции программирования, построенные поверх каркаса MapReduce. Мы расскажем о нескольких примерах применения Hadoop для решения реальных практических задач.

ГЛАВА 9.
Эксплуатация Hadoop в облаке

В этой главе:

- Настройка вычислительного облака с помощью служб Amazon Web Services (AWS).
- Эксплуатация Hadoop в облаке AWS.
- Перенос данных в облако AWS Hadoop и обратно.

В зависимости от того, какие данные обрабатываются, состав решаемых в кластере Hadoop задач может существенно изменяться. Возможно, вы изредка запускаете большие задачи, требующие сотен узлов, которые в остальное время простаивают без дела. Или только начинаете знакомство с Hadoop и хотите поэкспериментировать перед тем, как инвестировать средства в создание выделенного кластера. Или владеете компанией-стартапом и хотите сэкономить деньги, избежав капитальных затрат на организацию кластера Hadoop. В этих и других ситуациях имеет смысл *арендовать* кластер машин, а не покупать его.

Общая парадигма предоставления вычислительных ресурсов в виде удаленной службы гибким и экономичным способом называется «*облачные вычисления*». Наиболее известной из инфраструктурных платформ для облачных вычислений является набор служб Amazon Web Services (AWS). Вы можете арендовать службы вычислений и хранения по требованию, масштабируя их с ростом бизнеса. На момент написания этой книги аренда вычислительной единицы, по мощности эквивалентной компьютеру с 32-разрядным процессором Opteron частотой 1,0 Гц с оперативной памятью 1,7 ГБ и диском

емкостью 160 ГБ обходилась в $0,10 в час. Стало быть, час использования кластера из 100 таких машин будет стоить жалкие 10 долларов! А совсем недавно лишь немногие избранные имели доступ к кластерам такого масштаба. Благодаря AWS и другим подобным службам вся мощь крупномасштабных вычислений теперь предоставлена в распоряжение масс.

Благодаря гибкости и эффективности запуск Hadoop в облаке AWS стал весьма популярен, и в этой главе мы научимся устанавливать и конфигурировать такой кластер.

9.1. Введение в Amazon Web Services

Для рассказа обо всех возможностях Amazon Web Services понадобилась бы целая книга. Компания Amazon постоянно добавляет новые службы и функции. Мы рекомендуем заглянуть на сайт AWS (http://aws.amazon.com), где можно найти подробную информацию и последние новости. Здесь же мы рассмотрим лишь основы, достаточные для эксплуатации кластера Hadoop.

Из всех служб, предлагаемых AWS, для запуска Hadoop в облаке интерес представляют Elastic Compute Cloud (EC2) и Simple Storage Service (S3). Служба EC2 предоставляет вычислительные мощности, необходимые для работы узлов Hadoop. Можете считать, что это большая ферма виртуальных машин. *Экземпляром* EC2 в терминологии AWS называется виртуальная вычислительная единица. Для каждого узла Hadoop требуется один экземпляр EC2. Вы арендуете экземпляр EC2 только на нужное вам время и платите за каждый час.

Компания, сдающая в аренду автомобили, выбрасывает из багажника все, что вы там оставили, когда вернули машину. Так и после завершения работы с экземпляром EC2 все ваши данные с него удаляются. Если вы хотите, чтобы данные сохранялись до следующего использования, то должны выгрузить их в какое-то постоянное хранилище. Облачная служба хранения Amazon S3 как раз и предназначена для этой цели.

Каждый экземпляр EC2 функционирует как стандартный компьютер с процессором Intel, к которому можно обращаться через Интернет. Экземпляр загружается из *машинного образа Amazon* (*Amazon Machine Image*), который называют также *AMI*, или просто *образ*. Пользователи с особыми потребностями могут создавать собственные

образы, но большинству хватает одного из многих готовых. Некоторые образы содержат «голую» операционную систему. Поддерживается более шести вариантов Linux, Windows Server и OpenSolaris. На других образах, помимо операционной системы, находится предустановленное программное обеспечение, например, СУБД, HTTP-сервер Apache, сервер приложений Java и т. п. AWS предлагает готовые образы, содержащие Hadoop на платформе Linux, а в Hadoop в свою очередь встроена поддержка для работы в среде EC2 и S3.

9.2. Настройка AWS

Этот раздел содержит краткое введение в настройку AWS. Мы рассматриваем лишь то, что совершенно необходимо для эксплуатации кластера Hadoop. Если вы уже знакомы с запуском и использованием экземпляров EC2 то можете сразу перейти к следующему разделу, посвященному настройке Hadoop в AWS.

Чтобы начать работать с AWS, вы должны сначала завести учетную запись. Зайдите на сайт http://aws.amazon.com/ и нажмите кнопку «Sign Up Now». Сама процедура не нуждается в пояснениях. Это не сложнее, чем купить на Amazon книгу. В ходе регистрации вы создаете для себя учетную запись (если вы раньше что-нибудь покупали на Amazon, то она уже существует) и активируете ее для оплаты использования AWS.

Примечание. Компания Amazon добавила службу Elastic MapReduce (EMR), существенно упрощающую работу с Hadoop в AWS. Главное то, что вам больше не нужно настраивать и запускать собственный кластер из экземпляров EC2. Но в обмен на такое удобство вы частично утрачиваете контроль над работой кластера и должны доплачивать за использование EMR. Мы обсудим службу EMR в разделе 9.6. Но призываем вас читать дальше, поскольку понимать процедуру настройки кластера EC2 полезно даже, если вы не собираетесь выполнять ее самостоятельно. Как минимум, вы будете знать, что EMR делает за кулисами.

После активации учетной записи необходимо сделать еще несколько шагов, прежде чем вы сможете создавать и использовать экземпляры машин.

1. Получить идентификаторы своей учетной записи, а также ключи аутентификации и сертификаты. Их следует сохранить на своей локальной машине для защиты данных, передаваемых AWS и обратно. Тем самым гарантируется, что арендовать вычислительные ресурсы от имени вашей учетной записи сможете вы и только вы.

2 Скачать и установить набор командных утилит для управления своими экземплярами EC2. Это специальные программы, позволяющие запускать и останавливать экземпляры EC2 в виртуальном кластере.

3 Сгенерировать пару ключей для работы по протоколу SSH. После запуска экземпляра EC2 вы будете заходить на него с помощью SSH (напрямую или применяя специальные инструменты). По умолчанию для работы с SSH необходима пара ключей, позволяющих аутентифицироваться без использования паролей.

Мы рассмотрим все эти шаги ниже.

9.2.1. Получение учетных данных для аутентификации в AWS

AWS поддерживает два механизма аутентификации: по идентификатору ключа доступа AWS (AWS Access Key Identifier) и по сертификату X.509. Для запуска Hadoop в среде AWS потребуются оба механизма, и настроить их можно на странице Access Identifiers, где находятся все средства управления учетной записью AWS (http://aws.amazon.com/account/). Идентификатор ключа доступа AWS состоит из двух частей: собственно идентификатор ключа доступа (Access Key ID) и секретный ключ доступа (Secret Access Key). На рис. 9.1 изображена часть страницы Access Identifiers. Идентификатор ключа доступа – строка из 20 букв и цифр, а секретный ключ доступа – из 40. Никому не сообщайте свой секретный ключ доступа. Чтобы показать его на веб-странице, нужно дополнительно нажать кнопку Show (на случай, если кто-то стоит у вас за спиной и подглядывает). Если секретный ключ был скомпрометирован, то необходимо сгенерировать новый. Впоследствии, когда дело дойдет до настройки кластера Hadoop, вы должны будете ввести идентификатор ключа доступа и сам секретный ключ.

Совет. В тех случаях, когда необходимо обратиться из Hadoop к службе S3, вы должны будете сообщить Hadoop свой идентификатор ключа доступа и секретный ключ в специально составленном URI. К сожалению, AWS разрешает включать в секретный ключ символ косой черты (/), что приводит к формированию некорректного URI. Хотя существуют способы сообщить Hadoop идентификатор ключа доступа AWS Access Key ID и без использования URI, лучше повторить генерацию секретного ключа, пока не получится ключ без символа /.

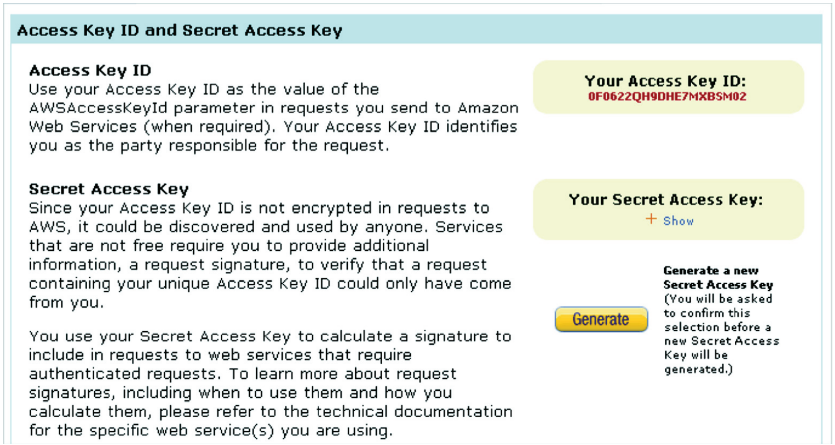


Рис. 9.1. Получение идентификатора ключа доступа AWS и секретного ключа доступа

Настроить сертификат X.509 чуть сложнее. На той же странице Access Identifiers имеется раздел *X.509 Certificate*, показанный на рис. 9.2. Для генерации нового сертификата нажмите кнопку Create New. Сертификат подразумевает наличие двух ключей: открытого и закрытого. В отличие от идентификатора ключа доступа и секретного ключа эти ключи состоят из нескольких сотен символов и должны храниться в виде файлов. После создания сертификата X.509 вы оказываетесь на странице, откуда можно скачать оба файла (рис. 9.3).

Рис. 9.2. Управление сертификатом X.509. Можно либо загрузить уже имеющийся у вас сертификат, либо попросить AWS сгенерировать новый

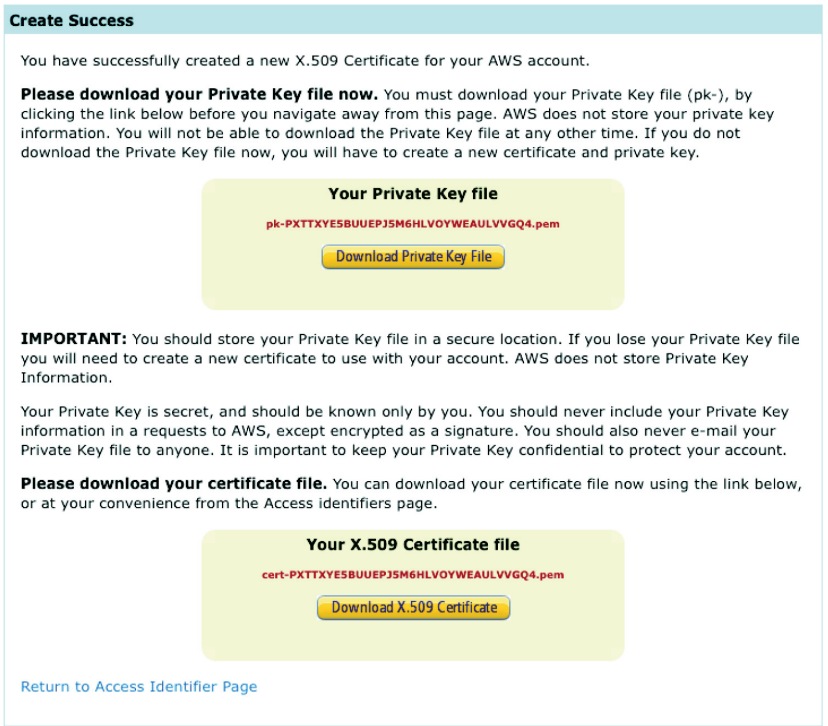


Рис. 9.3. Скачивание файлов с закрытым ключом и сертификатом X.509

Открытый ключ называется также *файлом сертификата*. Закрытый ключ, как следует из самого названия, не следует сообщать никому. Даже компания Amazon не сохраняет копию этого ключа. AWS сообщает имена файлов сертификата и закрытого ключа, и именно под этими именами вы должны сохранить файлы. В начале имен файлов стоят соответственно префиксы cert- и pk-, и оба файла имеют расширение .pem. Вы должны создать подкаталог .ec2 в своем домашнем каталоге и сохранить там оба файла. В частности, в Linux на вашей машине должны быть сохранены следующие два файла:

```
~/.ec2/cert-HKZYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem
~/.ec2/pk-HKZYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem
```

Наконец, запомните номер своей учетной записи в AWS. Он находится в правом верхнем углу страницы Access Identifiers и представляет собой 12-значное число с дефисами вида «4952-1993-3132».

Идентификатором вашей учетной записи является это же число без дефисов: «495219933132». Этот идентификатор понадобится при настройке Hadoop в среде EC2.

Не так уж мало всего придется сгенерировать и записать. В итоге у вас должно быть пять объектов:

- 20-символьный алфавитно-цифровой идентификатор ключа доступа;
- 40-символьный секретный ключ доступа;
- файл сертификата X.509 в каталоге .ec2;
- файл закрытого ключа в каталоге .ec2;
- 12-значный цифровой идентификатор учетной записи AWS (без дефисов).

Все они потребуются для аутентификации в AWS и управления кластером Hadoop.

9.2.2. Получение командных утилит

Получив учетные данные для обеспечения безопасности, вы должны будете скачать и настроить командные утилиты AWS для создания и управления экземплярами EC2. Эти утилиты написаны на языке Java, который, надо полагать, уже установлен на вашем локальном компьютере.

Все утилиты EC2 находятся в одном ZIP-файле, который можно скачать из центра ресурсов AWS EC2[1]. Распакуйте этот файл в каталог, который собираетесь использовать для работы с AWS. Среди распакованных файлов вы найдете как программы на Java, так и скрипты оболочки для Windows, Linux и Mac OS X.

Конфигурировать сами командные утилиты не нужно, но перед тем, как их использовать, следует задать несколько переменных окружения. Переменная `EC2_HOME` должна содержать путь к каталогу, в который распакованы командные утилиты. Если вы не переименовывали этот каталог, то он будет называться ec2-api-tools-A.B-nnnn, где A, B, n – номера версии и выпуска. Переменные `EC2_CERT` и `EC2_PRIVATE_KEY` должны указывать на файлы сертификата X.509 и открытого ключа соответственно. На мой взгляд, полезно написать скрипт для установки всех переменных окружения, необходимых для работы с командными утилитами AWS. В листинге 9.1 приведен текст

[1] http://developer.amazonwebservices.com/connect/entry.jspa?externalID=351&categoryID=88

такого скрипта `ec2-init.sh` для Linux, Unix и Mac OS X. Его следует запускать перед использованием любой утилиты AWS:

```
source ec2-init.sh
```

или

```
. ec2-init.sh
```

Листинг 9.1.
ec2-init.sh: скрипт установки переменных окружения для утилит EC2 (Unix)

```
export JAVA_HOME = /Library/Java/Home
export EC2_HOME = ~/Projects/Hadoop/aws/ec2-api-tools-1.3-30349
export PATH = $PATH:$EC2_HOME/bin:$HADOOP_HOME/src/contrib/ec2/bin
export EC2_PRIVATE_KEY = ~/.ec2/pk-HKZYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem
export EC2_CERT = ~/.ec2/cert-HKZYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem
```

Аналогичный скрипт для Windows приведен в листинге 9.2. Его можно запустить в окне консоли командой ec2-init.bat.

Листинг 9.2.
ec2-init.bat: скрипт установки переменных окружения для утилит EC2 (Windows)

```
set JAVA_HOME = "C:\Program Files\Java\jdk1.6.0_08"
set EC2_HOME = "C:\Program Files\Hadoop\aws\ec2-api-tools-1.3-30349"
set PATH = %PATH%;%EC2_HOME%\bin;%HADOOP_HOME%\src\contrib\ec2\bin
set EC2_PRIVATE_KEY = c:\ec2\pk-HKZYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem
set EC2_CERT = c:\ec2\cert-HKZYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem
```

Если вы собираетесь работать с AWS часто, то имеет смысл не писать отдельный скрипт, а включить эти команды непосредственно в скрипт запуска операционной системы (например, в файл `.profile` или `autoexec.bat`).

На вашей машине пути в этом скрипте будут другими. Переменная окружения `JAVA_HOME` должна быть установлена, иначе утилиты AWS вообще не будут работать. В примерах выше мы это делаем, хотя, скорее всего, она уже установлена где-то в другом месте. Кроме того, скрипт добавляет путь к каталогу `bin`, в котором находятся утилиты, в переменную `PATH`. После этого запускать утилиты будет гораздо проще, так как не придется всякий раз вводить полный путь. В `PATH` добавляется также путь к каталогу, где хранятся входящие в Hadoop средства работы с EC2, хотя использовать их мы начнем только в следующем разделе.

В облако AWS входят машины, физически расположенные в различных регионах. Когда писалась эта книга, поддерживались два

региона: США и ЕС. Если хотите, можете указать, в каком регионе должны запускаться ваши экземпляры EC2; это даст возможность снизить сетевые задержки. Установив переменные окружения, попробуем запустить первую из утилит AWS, которая запрашивает у Amazon список поддерживаемых регионов:

```
ec2-describe-regions
```

Будет напечатано примерно следующее:

```
REGION      us-east-1      us-east-1.ec2.amazonaws.com
REGION      eu-west-1      eu-west-1.ec2.amazonaws.com
```

Во втором столбце представлены названия регионов (us-east-1 и eu-west-1), а в третьем – соответствующие им *оконечные точки службы*. По умолчанию подразумевается регион us-east-1. Чтобы выбрать другой регион, запишите в переменную окружения `EC2_URL` URL-адрес его оконечной точки службы. Соответствующую команду можно включить в показанный выше скрипт инициализации AWS.

Совет. Помимо официальных командных утилит, имеются также графические программы для управления службами EC2 и S3 – более удобные для пользователя. Две наиболее популярных реализованы в виде расширений Firefox: Elasticfox и S3Fox. Elasticfox (http://developer.amazonwebservices.com/connect/entry.jspa?entryID=609) содержит основные функции управления EC2, в частности запуск новых экземпляров и вывод списка работающих в данный момент. Инструмент S3Fox (http://www.suchisoft.com/ext/s3fox.php) написан сторонним разработчиком и предназначен для организации хранения в службе S3. Выглядит он, как графический FTP-клиент для управления удаленной системой хранения.

9.2.3. Подготовка пары ключей для работы с SSH

Запустив экземпляр EC2, вы, наверное, захотите зайти на него и поработать. Предлагаемый по умолчанию (сконфигурированный в готовых образах) механизм входа в систему основан на использовании безопасной оболочки SSH, для чего необходима пара ключей. Один из двух ключей (открытый) находится на экземпляре EC2, а второй (закрытый) – на вашем локальном компьютере. Вместе они защищают данные, которыми обмениваются ваша машина и экземпляр EC2.

Примечание. Возможно, вы знакомы с тем, как заходить на удаленную машину, используя SSH с паролем. SSH с парой ключей – альтернативный механизм. Для аутентификации применяется не пароль, а закрытый ключ, хранящийся в виде файла на вашем локальном компьютере. Закрытый ключ, как и пароль, не должен знать никто, кроме вас.

У каждой пары ключей имеется имя, по которому ее можно идентифицировать. Обращаясь к службе Amazon EC2 с просьбой создать экземпляр, вы должны сообщить, какой открытый ключ встраивать в этот экземпляр, указав имя ключа. Открытый ключ SSH необходимо создать и зарегистрировать в Amazon еще до создания экземпляра EC2.

Для генерации пары ключей и регистрации открытого ключа в Amazon EC2 под именем `gsg-keypair` предназначена следующая команда:

```
ec2-add-keypair gsg-keypair
```

Эта команда не сохраняет закрытый ключ в локальном файле, а выводит его на стандартный вывод (stdout), как показано на рис. 9.4. Сохранить его в виде файла должны вы сами, воспользовавшись текстовым редактором. Точнее, скопируйте всё, что находится между строками

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

включительно, в новый файл с именем id_rsa-gsg-keypair.

Для простоты рекомендуем сохранить этот файл в том же каталоге .ec2, где находятся ваш закрытый ключ и сертификат X.509. Необходимо также установить для этого файла права доступа, разрешающие читать его только вам. В Linux и других системах, основанных на Unix, для этого служит такая команда:

```
chmod 600 ~/.ec2/id_rsa-gsg-keypair
```

Все экземпляры EC2, входящие в один кластер Hadoop, пользуются одним и тем же открытым ключом. Поэтому для входа на любой узел кластера достаточно одного закрытого ключа, так что вам потребуется лишь одна пара ключей. Дополнительные пары могут понадобиться, если вы собираетесь работать с несколькими кластерами Hadoop или с экземплярами EC2, не входящими в кластер.

На этом одноразовая процедура подготовки учетных данных и сертификатов завершается и можно приступить к созданию кластера машин в облаке Amazon. Для создания экземпляров EC2 и захода на них с целью запуска кластера можно воспользоваться утилитами AWS вручную. Но это долго и чревато ошибками. К счастью, в Hadoop уже включены средства для работы с AWS, которые мы обсудим в следующем разделе. Но перед тем как читать дальше, мы настоятельно рекомендуем потратить некоторое время на ознакомление с документацией по EC2, в том числе с руководством для начинающих

«Getting Started Guide»[2]. У службы EC2 есть много конфигурационных параметров и способов настройки. Приступая к оптимизации своего кластера Hadoop в среде AWS, полезно знать, что имеется в вашем распоряжении.

```
KEYPAIR gsg-keypair  1f:51:ae:28:bf:89:e9:d8:1f:25:5d:37:2d:7d:b8:ca:9f:f5:f1:6f
-----BEGIN RSA PRIVATE KEY-----
MIIEoQIBAAKCAQBuLFg5ujHrtm1jnutSuoO8Xe56LlT+HM8v/x<aa39EstM3/aFxTHgElQiJLChp
HungXQ29V¯c8rc1bW0lkdi23OH5eqkMHGhvEwqa0HWASUMll4o3o/IX+0f2UcPoKCOVUR+¸x71Sg
5AU52EQfanIn3ZQ8lFW7Edp5a3q4DhjGlUKToHVbicL5E+g45zfB95wIyywWZfeW/UUF3LpGZyq/
ebIUlq1qTbHkLbCC2r7RTn8vpQWp47BGVYGtGSBMpTRP5hnbzzuqj3itkiLHjU39S2sJCJ0TrJx5
i8BygR4s3nHKBj8l+ePQxG1kGbF6R4yg6sECmXn17MRQVXODNHZbAgMBAAECggEAY1tsiUsIwDl5
91CXirkYGuVfLyLflXenxfI50mDFms/mumTqloHO7tr0oriHDR5K7wMcY/YY5YkcXNo7mvUVD1pM
ZNUJs7rw9gZRTrf7LylaJ58kOcyajw8TsC4e4LPbFaHwS1d6K8rXh64o6WgW4SrsB6ICmr1kGQI7
3wcfgt5ec¯u4TZf0OE9IHjn+2eRlsrjBdeORi7KiUNC/pAG23I5MdDOFEQRcCSigCj+4/mciFUSA
SWS4dMbrpb9FNSIcf9dcLxVM7/6KxgJNfZc9XWzUw77Jg8x92Zd0fVhHOux5IZC+UvSKWB4dyfcI
tE8C3p9bbU9VGyY5vLCAiIb4qQKBgQDLiO24GXrIkswF32YtBBMuVgLGCwU9h9HlO9mKACc2m8Cm1
jUE5IpzRj¯edc9I2qiIMUTwtgnw42auSCzbUeYMURPtDqyQ7p6AjMujp9EPemcSVOK9vXYL0Ptco
xW9MC0dtV6iPkCN7gOqiZXPRKaFbWADp16p8UAIvS/a5XXk5jwKBgQCKkpHi2EISh1uRkhxljyWC
iDCiK6JBRsMvpLbc0v5dKwP5alo1fmdR5PJaV2qvZSj5CYNpMAy1/EDNTY5OSIJU+0KFmQbyhsbm
rdLNLDL4+¯cnT7c62/aH01ohYaf/VCbRhtLlBfqGoQc7+sAc8vmKkesnF7CqCEKDyF/dhrxYdQKB
gC0iZzzNAapayz1+JcVTwwEid6j9JqNXbBc+Z2YwMi+T0Fv/P/ɔwkX/ypeOXnIUcw0Ih/YtGBVAC
DQbsz7LcY1HqXiHKYNWNvXgwwO+oiChjxvEkSdsTTIfnK4VSCvJ9BxDbQHjdiNDJbL6oar92UN7V
rBYvChJZF7LvUH4YmVpHAoGAbZ2X7XvoeEO+uZ58/BGKOIGHByHBDiXtzMhdJr15HTYjxK7OgTZm
gK+8zp4L9¯bvLGDMJO8vft32XPEWuvI8twCzFH+CsWLQADZMZKSsBasOZ/h1FwhdMgCMcY+Qlzd4
JZKjTSu3i7vhvx6RzdSedXEMNTZWN4qlIx3kR5aHcukCgYA9T+Zrvm1F0seQPbLknn7EqhXIjBaT
P8TTvW/6bdPi23ExzxZn7KOdrfclYRph1LHMpAONv/x2xALIf91UB+v5ohy1oDoasL0gij1houRe
2ERKKdwz0ZL9SWq6VTdhr/5G994CK72fy5WhyERbDjUIdHaK3M849JJuf8cSrvSb4g==
-----END RSA PRIVATE KEY-----
```

Рис. 9.4. Пример данных, выведенных программой ec2-add-keypair. В первой строке находится сигнатура ключа, а в последующих – сам закрытый ключ.

9.3. Настройка Hadoop в EC2

Для запуска Hadoop в кластере EC2 необходимо сначала установить Hadoop на своей локальной машине и добиться, чтобы он нормально работал в автономном режиме. После установки на своей машине в каталоге `src/contrib/ec2/bin` внутри инсталляционного каталога вы найдете скрипты для создания кластера EC2 Hadoop и доступа к нему.

9.3.1. Задание параметров защиты

Необходимо отредактировать всего один скрипт инициализации в файле `src/contrib/ec2/bin/hadoopec2-env.sh`. Следующим трем переменным нужно присвоить значения, полученные в разделе 9.2.1:

[2] http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/

- *AWS_ACCOUNT_ID* – ваш 12-значный идентификатор учетной записи AWS;
- *AWS_ACCESS_KEY_ID* – ваш 20-значный алфавитно-цифровой идентификатор ключа доступа;
- *AWS_SECRET_ACCESS_KEY* – ваш 40-символьный секретный ключ доступа.

Программы для работы с Hadoop в среде EC2 получают остальные параметры защиты из переменных окружения (которые устанавливаются после выполнения команды `source aws-init.sh`) или берут значения по умолчанию, которых должно быть достаточно, если вы настраивали AWS, как описано в разделе 9.2.

9.3.2. Конфигурирование типа кластера

Конфигурационные параметры кластера Hadoop задаются в файле `hadoop-ec2-env.sh`. Основных параметра три: `HADOOP_VERSION`, `INSTANCE_TYPE` и `S3_BUCKET`. Но сначала немного вводной информации.

Для создания экземпляра служба Amazon EC2 должна знать его тип и образ, из которого экземпляр загружается. Тип экземпляра – это физическая конфигурация виртуальной машины (ЦП, объем оперативной памяти, емкость диска и т. д.). На момент написания этой книги было определено пять типов экземпляров, объединенных в два семейства: стандартные и с мощным ЦП (high-CPU). Типы с мощным ЦП предназначены для задач, ориентированных на вычисления. В приложениях Hadoop, ориентированных на обработку больших объемов данных, они применяются редко. В стандартное семейство входят три типа экземпляров, их характеристики приведены в табл. 9.1.

Чем мощнее тип экземпляра, тем дороже он стоит, действующие тарифы опубликованы на сайте AWS.

Образы для загрузки экземпляра EC2 могут храниться только в службе Amazon S3. Существует много образов для различных видов работ. Вы можете взять один из общедоступных образов, заплатить за создание специального образа под свои требования или даже создать образ самостоятельно. Похожие образы хранятся в одной и той же *корзине* (*bucket*) *S3*[3]. Стандартные общедоступные образы Hadoop

[3] Корзиной называется раздел верхнего уровня в пространстве имен S3. Корзиной может владеть ровно одна учетная запись AWS и у нее должно быть глобально уникальное имя.

находятся в корзинах hadoop-ec2-images и hadoop-images. Мы будем пользоваться только корзиной hadoop-images, поскольку последние версии Hadoop (начиная с 0.17.1) в корзине hadoop-ec2-images отсутствуют. Разработчики Hadoop помещают новые EC2-образы в корзину hadoop-images при выпуске очередной версии с существенными изменениями. Чтобы узнать, какие образы находятся в корзине, выполните следующую команду:

```
ec2-describe-images -x all | grep hadoop-images
```

Таблица 9.1. Характеристики различных типов экземпляров EC2

Тип	ЦП	Память	Диск	Платформа	Интенсивность ввода/вывода	Имя
Малый	1 вычислительная единица EC2	1,7 ГБ	160 ГБ	32-разрядная	Умеренная	m1.small
Большой	4 вычислительных единицы EC2	7,5 ГБ	850 ГБ	64-разрядная	Высокая	m1.large
Очень большой	8 вычислительных единиц EC2	15 ГБ	1690 ГБ	64-разрядная	Высокая	m1. xlarge

На рис. 9.5 приведен пример информации, выводимой этой командой. В каждой строке описан один образ EC2. У каждого образа есть одиннадцать свойств, большая часть которых интересна только опытным пользователям AWS. Все необходимое для наших целей находится в третьем столбце, это *местоположение манифеста* образа. Местоположение задается двухуровневым путем, в котором верхний уровень – имя корзины S3, где находится образ. Как уже отмечалось, нас будет интересовать только корзина hadoop-images.

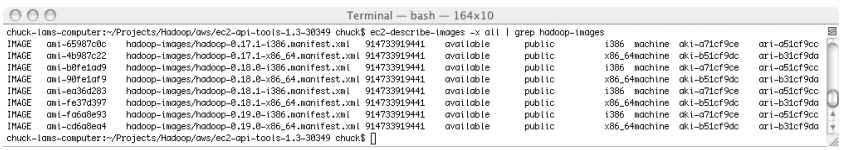


Рис. 9.5. Некоторые из имеющихся образов Hadoop для AWS

В состав местоположения манифеста входит номер версии Hadoop, а также обозначение платформы: *i386* или *x86_64*. Оно говорит о том, для какой архитектуры процессора предназначен образ:

32- или 64-разрядной. В качестве примера рассмотрим местоположение манифеста hadoop-images/hadoop-0.19.0-i386.manifest.xml. Этот образ содержит Hadoop версии 0.19.0 и может работать на 32-разрядных экземплярах EC2.

Узнав, какие образы Hadoop существуют, мы можем установить правильные значения переменных `HADOOP_VERSION`, `INSTANCE_TYPE` и `S3_BUCKET` в файле `hadoop-ec2-env.sh`. Если вы не собираетесь использовать нестандартный образ, то присвойте переменной `S3_BUCKET` значение hadoop-images. Переменная `INSTANCE_TYPE` по умолчанию равна m1.small, и этого должно быть достаточно. Главное, что нужно запомнить, – тип экземпляра определяет разрядность процессора и загружать экземпляр следует из совместимого образа (i386 или x86_64). Наконец, в переменную `HADOOP_VERSION` запишите версию Hadoop, которую собираетесь использовать. Выбранная комбинация `HADOOP_VERSION`, `INSTANCE_TYPE` и `S3_BUCKET` должна присутствовать в списке, который выводит команда `ec2-describe-images`.

9.4. Запуск MapReduce-программ в среде EC2

Все средства Hadoop для работы с EC2 находятся в каталоге `src/contrib/ec2/bin` внутри инсталляционного каталога Hadoop. Напомним, что наш скрипт `ec2-init.sh` добавил этот каталог в переменную `PATH`. Среди прочего в нем находится файл `hadoop-ec2`, представляющий собой метакоманду для выполнения других команд. Чтобы создать кластер Hadoop в среде EC2, введите такую команду:

```
hadoop-ec2 launch-cluster <cluster-name> <number-of-slaves>
```

Сначала она создает главный экземпляр EC2, затем загружает запрошенное количество подчиненных узлов, каждый из которых указывает на главный узел. По завершении эта команда выводит публичное DNS-имя главного узла, которое мы обозначим `<master-host>`. В этот момент еще не все подчиненные узлы полностью загружены. Для наблюдения за состоянием кластера и подчиненных узлов можете зайти в веб-интерфейс JobTracker по адресу http://`<master-host>`:50030/.

Примечание. Новые пользователи EC2 не могут запускать более 20 одновременно работающих экземпляров. Для увеличения лимита следует заполнить форму запроса экземпляра Amazon EC2 Instance Request Form по адресу http://www.amazon.com/gp/html-forms-controller/ec2-request.

После создания кластера Hadoop вы можете зайти на главный узел и работать с кластером так, будто он организован на ваших машинах. Для входа выполните команду:

```
hadoop-ec2 login <cluster-name>
```

Переменная `$HADOOP_HOME` для экземпляра Hadoop EC2 содержит значение `/usr/local/hadoop-x.y.z`, где `x.y.z` – номер версии Hadoop. Быстро проверить, работает ли Hadoop в этом кластере, можно следующим образом:

```
# cd /usr/local/hadoop-*
# bin/hadoop jar hadoop-*-examples.jar pi 10 10000000
```

Далее в этой главе знак решетки (`#`) в начале командной строки будет означать, что команду следует выполнять на главном узле кластера Hadoop EC2, а не на локальной машине. Приведенные выше команды запускают программу для вычисления числа «пи» с помощью Hadoop. Следить за ее работой можно на странице по адресу http://<master-host>:50030.

9.4.1. Перенос своего кода в кластер Hadoop

Все приложения Hadoop состоят из двух частей: код и данные. Сначала перенесем в кластер код, а в следующем подразделе обсудим, как получить доступ к данным (для чего необязательно переносить их в кластер, хотя это и возможно).

Копирование кода приложения на главный узел кластера Hadoop EC2 производится командой scp. Выполните на своей локальной машине такие команды:

```
source hadoop-ec2-env.sh
scp $SSH_OPTS <local-filepath> root@$MASTER_HOST:<master-filepath>
```

где `<local-filepath>` – путь к каталогу с кодом приложения на локальной машине, а `<master-filepath>` – путь к целевому каталогу на главном узле.

9.4.2. Доступ к данным из кластера Hadoop

Поскольку кластер Hadoop EC2 арендованный, находящиеся там данные (в том числе и в файловой системе HDFS) не сохраняются между сеансами. Их необходимо сохранить где-то в другом месте и предоставить кластеру для обработки. Вариантов сохранения и загрузки данных множество, каждый со своими плюсами и минусами.

Перенос данных непосредственно в систему HDFS

Если размер данных невелик (<100 ГБ) и обрабатывается только один раз, то проще всего скопировать их на главный узел, а оттуда в систему HDFS кластера. Копирование данных на главный узел ничем не отличается от копирования кода приложения (см. раздел 9.4.1). После того как данные окажутся на главном узле, вы можете зайти на него и скопировать данные в HDFS с помощью стандартной команды Hadoop:

```
# bin/hadoop fs –put <master-filepath> <hdfs-filepath>
```

Поток данных графически представлен на рис. 9.6. Сразу стоит отметить несколько проблем. Во-первых, за сетевой трафик между AWS и внешним миром взимается плата (в дополнение к почасовой оплате использования каждого экземпляра), но трафик внутри AWS бесплатен. В данном случае нужно будет платить за копирование на главный узел, но не за копирование с главного узла в HDFS (не берется плата также за передачу данных в процессе работы MapReduce-программы и между ней и HDFS). Но какой бы способ загрузки данных в кластер Hadoop ни выбрать, по крайней мере один раз трафик оплачивать придется. В данном случае перенос данных на главный узел займет довольно много времени, потому что соединение между вашей машиной и AWS гораздо медленнее, чем внутри AWS. Но повторим еще раз: эти временные издержки неизбежны при любой организации потока данных. Проблема же описанной выше архитектуры в том, что время

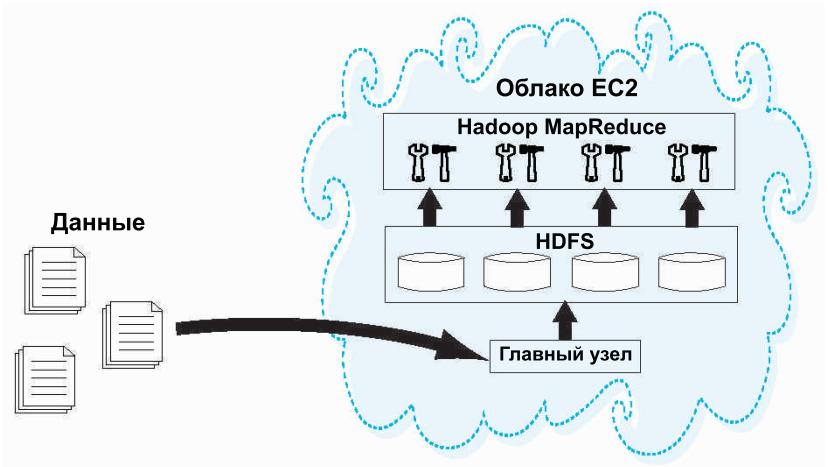


Рис. 9.6. Непосредственный перенос данных в облако Hadoop EC2

и деньги приходится тратить при *каждом* запуске кластера Hadoop. Если одни и те же входные данные будут по-разному обрабатываться в нескольких сеансах, то такой подход не рекомендуется.

Еще один недостаток этого варианта связан с ограничением на объем входных данных. Сначала все данные должны оказаться на главном узле, а в малом экземпляре EC имеется только один раздел диска емкостью 150 ГБ. Обойти это ограничение можно, если удастся разбить входные данные на несколько порций и копировать по одной порции за раз. Можно также выбрать больший экземпляр с несколькими разделами по 420 ГБ. Но прежде чем пробовать более сложные схемы, следует рассмотреть возможность включения в поток данных службы S3.

Перенос данных в HDFS через S3

S3 – это входящая в состав AWS облачная служба хранения. Вы уже знаете, что она используется для хранения образов EC2. Плата за хранение данных в S3 взимается за сетевой трафик с машинами, не входящими в AWS, плюс ежемесячная абонентская плата, зависящая от объема данных. Для многих приложений и, в частности, для кластеров Hadoop EC2, такая модель тарификации хранения является привлекательной альтернативой.

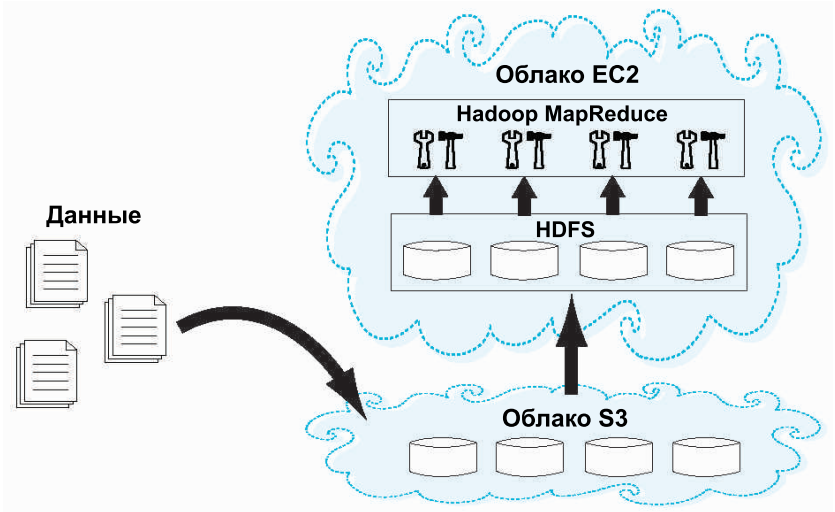


Рис. 9.7. Использование Hadoop в среде AWS в конфигурации S3 плюс HDFS

Такая модель потока данных изображена на рис. 9.7. Основное отличие от рис. 9.6 состоит в том, что данные сначала переносятся в облако S3, а не на главный узел. Отметим, что в отличие от главного узла данные в облачной системе хранения S3 сохраняются независимо от кластера Hadoop EC2. Вы можете много раз создавать и уничтожать кластеры Hadoop EC2, и все они будут читать из S3 одни и те же входные данные. Преимущество такой конфигурации в том, что временные и денежные затраты на копирование данных в AWS вы будете нести только один раз – при копировании в S3, тогда как в конфигурации на рис. 9.6 они возникают при каждом сеансе работы с кластером. После того как входные данные скопированы в S3, копирование их из S3 в систему HDFS кластера производится быстро и бесплатно, поскольку обе службы – S3 и EC2 – работают внутри системы AWS. Правда, теперь приходится дополнительно вносить ежемесячную плату за хранение данных в S3, но обычно она минимальна. Если вам необходима масштабируемая система для хранения архивных данных, то S3 может выступать и в этой роли, что служит еще одним аргументом в пользу такой организации потока данных.

В стандартном дистрибутиве Hadoop уже имеется поддержка S3. Существует специальная файловая система S3 Block FileSystem, построенная на базе S3 и обеспечивающая хранение больших файлов (в S3 размер файла ограничен 5 ГБ). Файловую систему S3 Block FileSystem не следует отождествлять с S3, точно так же как HDFS не следует отождествлять с лежащей в ее основе файловой системой Unix.

Для S3 Block FileSystem требуется отдельная корзина S3. Создав ее, вы сможете скопировать в S3 данные со своей локальной машины:

```
bin/hadoop fs –put <local-filepath>
➥ s3://<access-key-id>:<secret-access-key>@<s3-bucket>/<s3-filepath>
```

Напомним, что `<access-key-id>` и `<secret-access-key>` – параметры аутентификации, описанные в разделе 9.2.1, а `<s3-bucket>` – имя корзины S3, созданной для размещения файловой системы S3 Block FileSystem.

Перенеся данные в S3, вы сможете затем скопировать их в любой кластер Hadoop EC2. На главном узле кластера выполните команду

```
# bin/hadoop distcp s3://<access-key-id>:<secret-access-key>@<s3-bucket>/
➥ <s3-filepath> <hdfs-filepath>
```

Поместив данные в HDFS, вы можете запускать Hadoop-программы, как обычно.

Доступ к данным непосредственно из S3

До сих пор мы рассматривали случай, когда перед запуском приложения Hadoop данные копируются в HDFS. Это позволяет сохранить локальность данных между системой хранения и MapReduce-программой. Для очень маленьких задач можно отказаться от HDFS и пожертвовать локальностью данных в обмен на возможность пропустить шаг копирования данных из S3 в HDFS. Такой поток данных изображен на рис. 9.8.

В этом случае укажите S3 в качестве пути к входным данным при запуске приложения Hadoop:

```
# bin/hadoop jar <app-jar-filepath> s3://<access-key-id>:
➥ <secret-access-key>@<s3-bucket>/<s3-filepath> <hdfs-filepath>
```

Эта команда сохраняет выходной файл в HDFS, но можно и здесь указать S3.

Другие способы использования S3 вместе с Hadoop

В некоторых ситуациях могут быть полезны и другие способы использования S3.

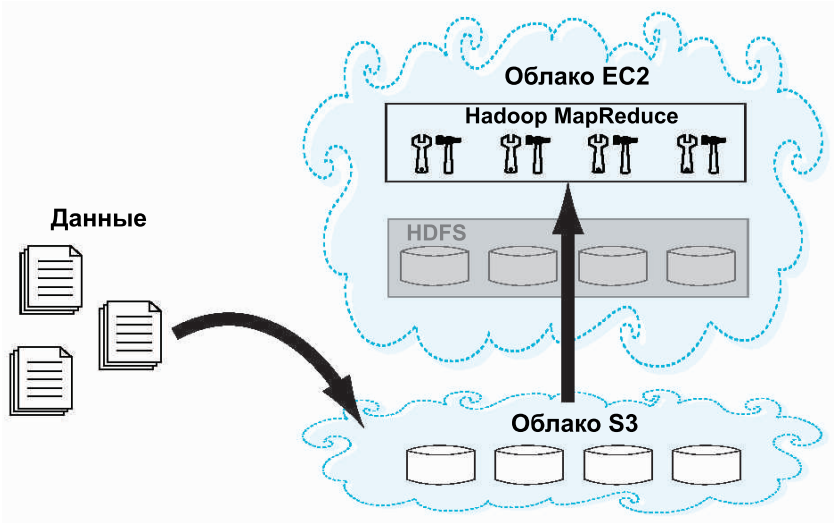


Рис. 9.8. Hadoop, работающий в среде EC2, может напрямую обращаться к данным, хранящимся в S3

До сих пор мы использовали специальную файловую систему (S3 Block FileSystem) для хранения данных в S3. Альтернатива –

работать с «родной» файловой системой S3. Ее основной недостаток – ограничение 5 ГБ на размер файла. Но если ваши входные файлы меньше этого предела, то лучше «родной» файловой системы не найти. Она совместима со всеми стандартными инструментами S3, тогда как формат файловой системы Hadoop S3 весьма специфичен. Стандартные инструменты S3 делают работу с «родной» файловой системой более прозрачной и понятной. Чтобы использовать «родную» файловую систему S3 вместо S3 Block FileSystem, указывайте в путях к файлам схему `s3`, а не `s3n`. Например, пишите

```
s3n://<access-key-id>:<secret-access-key>@<s3-bucket>/<s3-filepath>
```

вместо

```
s3://<access-key-id>:<secret-access-key>@<s3-bucket>/<s3-filepath>
```

Если вы пользуетесь службой S3 часто, то утомительно набирать длинный URI всякий раз, как нужно обратиться к какому-то файлу. Упростить себе жизнь можно, добавив в файл `conf/core-site.xml` такие строки:

```
<property>
  <name>fs.s3.awsAccessKeyId</name>
  <value>AWS_ACCESS_KEY_ID</value>
</property>

<property>
  <name>fs.s3.awsSecretAccessKey</name>
  <value>AWS_SECRET_ACCESS_KEY</value>
</property>
```

При этом вместо `AWS_ACCESS_KEY_ID` следует указать свой 20-значный идентификатор ключа доступа, а вместо `AWS_SECRET_ACCESS_KEY` – свой 40-символьный секретный ключ доступа. После этого URI-адреса файлов в S3 можно сократить до

```
s3://<s3-bucket>/<s3-filepath>
```

или

```
s3n://<s3-bucket>/<s3-filepath>
```

в случае «родной» файловой системы S3.

Примечание. Если вам не повезло и в секретном ключе доступе встречается символ /, то включить секретный ключ в URI не получится. В таком случае один из способов использовать учетную запись AWS/S3 состоит в том, чтобы указать секретный ключ в файле `core-site.xml`, как описано выше. (В некоторых документах рекомендуют экранировать символ /, заменив его в URI строкой `%2F`, но на практике такой способ, похоже, не работает.)

Другой способ сократить количество вводимых символов – сделать S3 файловой системой по умолчанию вместо HDFS. Для этого сначала добавьте в файл `conf/coresite.xml` два описанных выше свойства, а затем измените свойство `fs.default.name` следующим образом:

```
<property>
  <name>fs.default.name</name>
  <value>s3://S3_BUCKET</value>
</property>
```

где `S3_BUCKET` – корзина S3, выбранная для размещения файловой системы Hadoop S3 Block FileSystem (выше мы обозначали ее `<s3-bucket>`).

9.5. Очистка и останов экземпляров EC2

По умолчанию Hadoop хранит выходные данные задачи Hadoop в файловой системе HDFS на кластере, а вы бы, наверное, хотели сохранить их в более постоянном месте. Все рассмотренные выше способы загрузки данных относятся и к выгрузке, только применяются в обратном направлении. Основное отличие заключается в том, что размер выходных данных обычно на несколько порядков меньше, чем входных. С учетом этого обстоятельства копирование через главный узел может оказаться оптимальным вариантом.

Поскольку вы платите за аренду экземпляров EC2 на почасовой основе, то важно останавливать экземпляры по завершении работы, чтобы AWS прекратила начислять плату. А ведь так просто выйти из кластера, позабыв, что экземпляры продолжают работать, и отнюдь не бесплатно! Чтобы правильно остановить кластер, выполните следующую команду:

```
bin/hadoop-ec2 terminate-cluster <cluster-name>
```

После этого все экземпляры EC2 прекращают работу, и хранившиеся на них данные теряются. Больше никакой очистки не требуется.

9.6. Amazon Elastic MapReduce и другие службы AWS

Amazon Web Services постоянно пополняется новыми возможностями, многие из которых упрощают жизнь разработчикам, пишущим

для Hadoop. Из последних на момент написания книги новшеств стоит упомянуть службы Amazon Elastic MapReduce (EMR) и AWS Import/Export.

9.6.1. Amazon Elastic MapReduce

За небольшую дополнительную плату служба EMR создает заранее сконфигурированный кластер Hadoop, в котором вы сможете запускать свои MapReduce-программы. Основное упрощение состоит в том, что вам не нужно заботиться о настройке экземпляров EC2, а, значит, не придется иметь дело с сертификатами, командными утилитами и прочим. Вы будете взаимодействовать с EMR исключительно через веб-консоль по адресу https://console.aws.amazon.com/elasticmapreduce/home. На рис. 9.9 показана начальная страница этого веб-интерфейса.

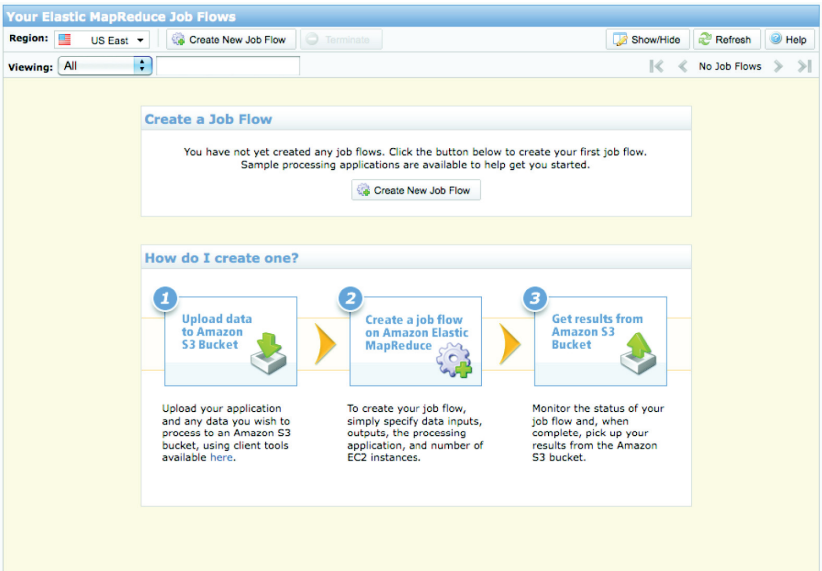


Рис. 9.9. Начальная страница веб-консоли службы Amazon Elastic MapReduce. Для определения потока выполнения задачи следуйте инструкциям на экране.

Ее дизайн ориентирован на работу с одиночными задачами. Вы передаете задачу MapReduce в виде скрипта (Streaming, Pig или Hive) либо JAR-файла, а EMR настраивает кластер для выполнения этой

задачи. По завершении задачи кластер по умолчанию останавливается. Входные данные читаются из S3, выходные записываются туда же. Активный пользователь Hadoop обычно запускает сразу много задач, обрабатывающих одни и те же данные, поэтому для него такая конфигурация не годится (см. раздел 9.4.2). Но человек, работающий с Hadoop лишь время от времени, согласится, что EMR здорово упрощает запуск MapReduce-программ в облаке. К тому же нетрудно предположить, что EMR будет развиваться и в конечном итоге станет основным способом эксплуатации Hadoop в среде AWS.

Дополнительные сведения о службе Amazon Elastic MapReduce можно найти на следующих сайтах:

- http://aws.amazon.com/elasticmapreduce/.
- http://docs.amazonwebservices.com/ElasticMapReduce/latest/GettingStartedGuide/.

9.6.2. AWS Import/Export

Одно из основных препятствий для обработки больших объемов данных в облаке – сложность перемещения туда наборов данных. Если вы уже разработали процедуру сохранения данных в S3, то запустить Hadoop в среде EC2 для их обработки несложно. С другой стороны, если требуется перенести в облако данные с единственной целью проанализировать их, то сам перенос может представлять трудности. Компания Amazon разработала службу AWS Import/Export, смысл которой в том, что вы передаете физический жесткий диск, а персонал компании загружает находящиеся на нем данные в S3 по высокоскоростной внутренней сети. Интересна вам эта служба или нет, зависит от скорости имеющегося у вас подключения к сети. В табл. 9.2 приведены грубые оценки, взятые с сайта AWS.

Таблица 9.2. Оценка размера набора данных, при котором служба AWS Import/Export выгоднее, чем загрузка через Интернет.

Скорость подключения к Интернету	Теоретически минимальное число дней, необходимое для загрузки 1 ТБ данных при использовании 80% пропускной способности сети	Когда имеет смысл прибегнуть к AWS Import/Export
T1 (1,544 Мб/с)	82 дня	свыше 100 ГБ
10 Мб/с	13 дней	свыше 600 ГБ

Таблица 9.2. (окончание).

Скорость подключения к Интернету	Теоретически минимальное число дней, необходимое для загрузки 1 ТБ данных при использовании 80% пропускной способности сети	Когда имеет смысл прибегнуть к AWS Import/Export
T3 (44,736 Мб/с)	3 дня	свыше 2 ТБ
100 Мб/с	от 1 до 2 дней	свыше 5 ТБ
1000 Мб/с	менее 1 дня	свыше 60 ТБ

Дополнительные сведения о службе AWS Import/Export можно найти на сайте http://aws.amazon.com/importexport/.

9.7. Резюме

Облачная инфраструктура – прекрасное место для запуска Hadoop, так как позволяет без труда масштабировать кластер до сотен узлов и позволяет избежать капиталовложений на раннем этапе. В Hadoop уже включена поддержка служб Amazon Web Services (AWS). В этой главе мы познакомились с тем, как завести себе учетную запись для аренды вычислительных мощностей в AWS. Решив арендовать узлы, вы найдете в Hadoop инструменты для автоматизации процедур организации и запуска кластера. В AWS имеется также служба облачного хранения данных (S3), которую можно использовать вместе с HDFS или вместо нее. Существуют разные конфигурации со своими плюсами и минусами. Наконец, не следует забывать об останове кластера Hadoop по завершении работы. Оплата за пользование облачной инфраструктурой почасовая и начисляется, пока виртуальные машины не будут явно остановлены.

ГЛАВА 10. Программирование с помощью Pig

В этой главе:

- Установка Pig и работа с оболочкой Grunt.
- Язык Pig Latin.
- Расширение языка Pig Latin за счет пользовательских функций.
- Интерфейс с реляционными базами данных.
- Эффективное вычисление схожих документов с применением простого скрипта на языке Pig Latin.

Часто приходится слышать, что для каркаса MapReduce трудно программировать. Впервые обдумывая некую проблему обработки данных, вы размышляете о ней в терминах таких операций управления потоком выполнения, как циклы и фильтры. Однако программу для MapReduce требуется реализовывать с помощью функций распределения и редукции и сцепления задач. Некоторые функции, считающиеся полноценными операциями в языках высокого уровня, оказывается совсем не тривиально реализовать в MapReduce; пример такого рода, относящийся к соединению данных, мы видели в главе 5. Pig – это расширение Hadoop, призванное упростить программирование за счет высокоуровневого языка обработки данных, сохранив при этом простоту масштабируемости и надежность Hadoop. В компании Yahoo, являющейся одним из самых активных пользователей Hadoop (и спонсором разработки ядра Hadoop и Pig), примерно 40 процентов всех задач для Hadoop написано на Pig. Еще один известный пользователь Pig – сайт Twitter[1].

[1] http://www.slideshare.net/kevinweil/hadoop-pig-and-twitter-nosql-east-2009

Pig состоит из двух основных компонентов.

1. Высокоуровневый язык обработки данных Pig Latin.
2. Компилятор, который транслирует и запускает скрипт на языке Pig Latin, выбирая *механизм вычисления*. Основным механизмом вычисления является Hadoop. Pig поддерживает также локальный режим для целей разработки.

Pig упрощает программирование за счет того, что на языке Pig Latin очень легко выразить назначение кода. Это освобождает от необходимости оптимизировать программу вручную. По мере совершенствования компилятора Pig быстродействие программы на Pig Latin автоматически повышается.

10.1. Научитесь думать по-свински[2]

В дизайн Pig заложена определенная идеология. От любого подпроекта Hadoop мы ожидаем простоты использования, высокой производительности и неограниченной масштабируемости. Но для понимания Pig важно знать, какие решения были приняты при проектировании языка программирования (языка описания потоков данных), какие типы данных он поддерживает и как трактует определенные пользователем функции (UDF), считающиеся полноценными конструкциями языка.

10.1.1. Язык описания потоков данных

Программа на языке Pig Latin представляет собой последовательность шагов, каждый из которых описывает одно высокоуровневое преобразование данных. Преобразования поддерживают такие реляционные операции, как фильтрация, объединение, группировка и соединение. Ниже приведен пример написанной на Pig Latin программы для обработки журнала поисковых запросов.

```
log = LOAD 'excite-small.log' AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE group, COUNT(log);
DUMP cntd;
```

Несмотря на то, что операции записываются в реляционном духе, Pig Latin остается языком описания потоков данных. Такой язык удобнее для программистов, мыслящих в терминах алгоритмов, которые более естественно выражаются с помощью потоков данных и

[2] Pig по-английски «свинья». *Прим. перев.*

управления. С другой стороны, декларативный язык, например SQL, иногда оказывается проще для аналитиков, предпочитающих формулировать цель в терминах результата, который должна выдать программа. В Hadoop имеется еще один подпроект, Hive, ориентированный на пользователей, предпочитающих модель SQL. Подробнее о Hive мы будем говорить в главе 11.

10.1.2. Типы данных

Идеологию Pig в отношении типов данных можно выразить фразой «свинья ест всё». Формат входных данных произволен. Популярные форматы, к примеру, текстовые поля, разделенные знаком табуляции, поддерживаются изначально. Но пользователь может добавить функции для обработки других форматов файлов. Pig не требует метаданных или схемы, но, если таковая предоставлена, то может ей воспользоваться.

Pig может работать с реляционными, иерархическими, полуструктурированными и неструктурированными данными. Для поддержки такого разнообразия в Pig определены составные типы данных, например неупорядоченные множества (bag) и кортежи, которые могут быть вложенными и тем самым образовывать весьма сложные структуры данных.

10.1.3. Определенные пользователем функции

Pig проектировался для использования в самых разных приложениях – обработка журналов, обработка текстов на естественном языке, анализ топологии сетей и т. д. Ожидается, что для многих вычислений будет необходима специализированная обработка. Поэтому в архитектуру Pig с самого начала было заложено применение пользовательских функций. Умение писать UDF составляет существенную часть освоения Pig.

10.2. Установка Pig

Последнюю версию Pig можно скачать со страницы http://hadoop.apache.org/pig/releases.html. Когда писалась эта книга, последними были версии Pig 0.4 и 0.5. Для обеих требуется Java 1.6. Основное различие между ними состоит в том, что Pig 0.4 ориентирована

на версию Hadoop 0.18, а Pig 0.5 – на Hadoop 0.20. Как обычно, не забудьте установить переменную окружения `JAVA_HOME`, так чтобы она указывала на корень инсталляции Java. Пользователям Windows придется установить Cygwin. Предполагается, что кластер Hadoop уже настроен. В идеале это должен быть реальный кластер, работающий в полностью распределенном режиме, но для изучения подойдет и псевдораспределенный режим.

Чтобы установить Pig на локальную машину, нужно для начала распаковать полученный дистрибутив. В самом кластере Hadoop модифицировать ничего не придется. Можете считать, что Pig – это компилятор и некий набор средств для разработки и развертывания. Он упрощает программирование для MapReduce, но в остальном слабо связан с производственным кластером Hadoop.

В каталоге, куда вы распаковали Pig, следует создать подкаталоги `logs` и `conf` (если их еще нет). Pig читает конфигурационные параметры из файлов, находящихся в каталоге `conf`. Если вы только что создали этот каталог, то, разумеется, никаких конфигурационных файлов в нем еще нет, поэтому необходимо создать файл с именем pig-env.sh. Этот скрипт исполняется при запуске Pig и может использоваться для настройки переменных окружения. Помимо `JAVA_HOME`, интерес представляют переменные `PIG_HADOOP_VERSION` и `PIG_CLASSPATH`. С их помощью вы сообщаете Pig о своем кластере Hadoop. Например, следующие строки в pig-env.sh означают, что используется версия Hadoop 0.18, а в путь к классам Pig добавлен конфигурационный каталог локальной инсталляции Hadoop:

```
export PIG_HADOOP_VERSION=18
export PIG_CLASSPATH=$HADOOP_HOME/conf/
```

Мы предполагаем, что переменная `HADOOP_HOME` указывает на инсталляционный каталог Hadoop на локальной машине. После добавления конфигурационного каталога Hadoop (`conf`) в путь к классам, Pig может узнать, где находятся узлы NameNode и JobTracker кластера.

Вместо использования пути к классам Pig можно описать структуру кластера Hadoop в файле pig.properties. Этот файл свойств должен находиться в созданном ранее каталоге `conf`. В нем следует определить свойства `fs.default.name` и `mapred.job.tracker`, то есть местоположение файловой системы (узел NameNode, на котором находится HDFS) и узла JobTracker. Ниже приведен пример файла pig.properties, описывающего Hadoop, работающий в псевдораспределенном режиме:

```
fs.default.name=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
```

Для удобства добавим в переменную окружения `PATH` путь к подкаталогу `bin` инсталляционного каталога Pig в предположении, что на него указывает переменная `PIG_HOME`:

```
export PATH=$PATH:$PIG_HOME/bin
```

После того как этот каталог включен в список путей, для запуска Pig достаточно просто ввести команду `pig`. Для начала посмотрите, какие флаги она поддерживает:

```
pig -help
```

Запустим интерактивную оболочку Pig и убедимся, что конфигурационные параметры прочитаны правильно:

```
pig
2009-07-11 22:33:04,797 [main] INFO
➥ org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
➥ Connecting to hadoop file system at: hdfs://localhost:9000
2009-07-11 22:33:09,533 [main] INFO
➥ org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
➥ Connecting to map-reduce job tracker at: localhost:9001
grunt>
```

Информация, которую Pig сообщает о файловой системе и узле JobTracker, должна совпадать с прописанной в конфигурации. Сейчас вы находитесь в интерактивной оболочке Pig, которая называется Grunt[3].

10.3. Запуск Pig

Выполнить команды на языке Pig Latin можно тремя способами: в интерактивной оболочке Grunt, поместив их в файл скрипта или в виде вложенных запросов в программе на Java. Любой из этих способов работает в двух режимах: локальном и Hadoop (режим Hadoop в документации по Pig иногда называется режимом Mapreduce). В конце предыдущего раздела мы запустили оболочку Grunt в режиме Hadoop.

Оболочка Grunt позволяет вводить команды Pig вручную. Обычно это бывает нужно для нерегулярного анализа данных или во время интерактивных циклов разработки программы. Большие или часто запускаемые программы на Pig оформляются в виде файлов скриптов.

[3] Grunt по-английски «хрюкать». *Прим. перев.*

Для входа в Grunt наберите команду `pig`. Для запуска Pig-скрипта наберите ту же команду `pig`, но с именем файла в качестве аргумента, например: `pig myscript.pig`. По принятому соглашению, файлы со скриптами на Pig имеют расширение `.pig`.

Программы на Pig в чем-то похожи на SQL-запросы, а в состав Pig входит класс `PigServer`, позволяющий исполнять Pig-запросы из программ на Java. Концептуально это аналог использования JDBC для исполнения SQL-запросов. Вложенные Pig-программы – довольно сложная тема, дополнительную информацию по этому поводу можно найти на странице http://wiki.apache.org/pig/EmbeddedPig.

При запуске Pig в локальном режиме Hadoop вообще не используется[4]. Команды Pig компилируются и исполняются в собственной JVM и работают с локальными файлами. Обычно этот режим применяется на этапе разработки, когда нужно получать результаты быстро на небольшом наборе данных. При запуске же Pig в режиме Hadoop откомпилированная программа исполняется в контексте Hadoop. Как правило, предполагается, что речь идет о полностью распределенном кластере Hadoop. (Псевдораспределенная конфигурация Hadoop, упомянутая в разделе 10.2, приведена только для демонстрации. Она используется редко, разве что для отладки конфигурации.) Режим выполнения задается с помощью аргумента `-x` или `-exectype` команды `pig`. Войти в оболочку Grunt в локальном режиме можно так:

```
pig -x local
```

А в режиме Hadoop – так:

```
pig -x mapreduce
```

либо ввести команду `pig` без аргументов, поскольку режим Hadoop выбирается по умолчанию.

10.3.1. Управление оболочкой Grunt

Помимо исполнения предложений на языке Pig Latin (который мы рассмотрим в следующем разделе), оболочка Grunt поддерживает ряд служебных команд[5]. Команда `help` печатает справку обо всех служебных командах. Команда `quit` завершает работу с Grunt. Чтобы снять

[4] Есть планы изменить Pig, так чтобы Hadoop использовался даже в локальном режиме, что позволит программировать более единообразно. Обсуждение этого вопроса ведется на странице https://issues.apache.org/jira/browse/PIG-1053.

[5] Технически все это тоже команды Pig Latin, но вряд ли вам придет в голову использовать их вне Grunt.

задачу Hadoop, введите команду `kill`, а затем идентификатор задачи. Команда `set` устанавливает некоторые параметры Pig, например:

```
grunt> set debug on
grunt> set job.name 'my job'
```

Параметр `debug` включает или выключает отладочное протоколирование. Параметр `job.name` принимает строку в одиночных кавычках, которая в дальнейшем используется как имя задачи Hadoop, реализуемой данной Pig-программой. Наличие осмысленного имени полезно для идентификации задачи Pig в веб-интерфейсе Hadoop.

Оболочка Grunt поддерживает также команды для работы с файлами, например, `ls` и `cp`. Полный список служебных и файловых команд приведен в табл. 10.1. Большая часть файловых команд – это подмножество команд для работы с файловой системой HDFS, и особых пояснений они не требуют.

Таблица 10.1. Служебные и файловые команды оболочки Grunt

Служебные команды	`help`
	`quit`
	`kill` ***`jobid`***
	`set debug [on|off]`
	`set job.name` ***`'jobname'`***

Файловые команды	`cat, cd, copyFromLocal, copyToLocal, cp, ls, mkdir, mv, pwd, rm, rmf, exec, run`

Но есть и две новых команды: `exec` и `run`. Они запускают Pig-скрипты, не покидая Grunt, и могут быть полезны при отладке скриптов. Команда `exec` исполняет Pig-скрипт в отдельном пространстве имен. Псевдонимы, определенные в скрипте, не видны оболочке, и наоборот. Команда `run` исполняет скрипт в пространстве имен самой оболочки (так называемый *интерактивный режим*). Эффект получается такой же, как если бы каждая строчка скрипта была набрана в оболочке.

10.4. Изучение языка Pig Latin с помощью Grunt

Прежде чем переходить к формальному описанию типов и операторов обработки данных в Pig, давайте выполним несколько команд в оболочке Grunt, чтобы получить представление о том, как

обрабатываются данные в Pig. Для обучения удобно запустить Grunt в локальном режиме:

```
pig -x local
```

Для начала попробуйте парочку файловых команд, скажем `pwd` и `ls`, позволяющих сориентироваться в файловой системе.

А теперь обратимся к данным. Позже мы еще раз воспользуемся данными о патентах из главы 4, а пока покопаемся в интересном наборе данных, взятом из журнала запросов поисковой системы Excite. Этот набор данных входит в состав дистрибутива Pig и находится в файле `tutorial/data/excite-small.log` в инсталляционном каталоге Pig. Данные представлены в виде трех столбцов, разделенных знаками табуляции. Первый столбец – идентификатор анонимного пользователя, второй – временная метка в формате Unix, а третий – собственно поисковый запрос. Ниже приведено несколько записей из 4500 содержащихся в файле (выборка специально взята не случайная):

```
3F8AAC2372F6941C   970916093724   minors in possession
C5460576B58BB1CC   970916194352   hacking telenet
9E1707EE57C96C1E   970916073214   buffalo mob crime family
06878125BE78B42C   970916183900   how to make ecstacy
```

Находясь в Grunt, введите следующее предложение для загрузки этих данных в «псевдоним» (то есть переменную) `log`:

```
grunt> log = LOAD 'tutorial/data/excite-small.log' AS (user, time, query);
```

Визуально ничего не происходит. В Grunt введенные предложения разбираются, но физически не исполняются, пока не будет введена команда `DUMP` или `STORE`. Команда `DUMP` распечатывает содержимое псевдонима, а команда `STORE` сохраняет его в файле. Почему Pig не исполняет команды, пока вы явно не запросите результат, станет ясно, если вспомнить, что обычно мы обрабатываем большой набор данных. В памяти просто не хватит места для «загрузки» данных, да и в любом случае мы хотим проверить логику плана выполнения перед тем, как тратить время и ресурсы на его физическое выполнение.

Команда `DUMP` обычно используется только для разработки. Чаще требуется сохранить полученные результаты в каком-нибудь каталоге командой `STORE`. (Как и Hadoop, Pig автоматически разбивает набор данных на файлы с именами `part-`*`nnnnn`*.) Выполняя для псевдонима команду `DUMP`, позаботьтесь о том, чтобы его содержимое не оказалось

слишком большим для осмысленного вывода на экран. Обычно для этого создается другой псевдоним командой `LIMIT` и команде `DUMP` передается этот, меньший, псевдоним. Команда `LIMIT` позволяет указать количество возвращаемых кортежей (строк). Например, чтобы вывести четыре кортежа псевдонима `log`, надо написать:

```
grunt> lmt = LIMIT log 4;
grunt> DUMP lmt;
(2A9EABFB35F5B954,970916105432L,+md foods +proteins)
(BED75271605EBD0C,970916001949L,yahoo chat)
(BED75271605EBD0C,970916001954L,yahoo chat)
(BED75271605EBD0C,970916003523L,yahoo chat)
```

В табл. 10.2 перечислены операторы чтения и записи в языке Pig Latin. `LIMIT`, строго говоря, не является ни тем, ни другим, но, поскольку часто используется вместе с ними, то мы решили включить его в эту таблицу.

Таблица 10.2. Операторы чтения/записи данных в языке Pig Latin

`LOAD`	`alias = LOAD 'file' [USING function] [AS schema];` Загружает данные из файла в отношение. По умолчанию использует функцию загрузки `PigStorage`, если противное не указано с помощью опции `USING`. С данными можно связать схему, задав ее в опции `AS`.
`LIMIT`	`alias = LIMIT alias n;` Ограничивает количество кортежей величиной *n*. Если используется сразу после того, как псевдоним `alias` был обработан оператором `ORDER`, то возвращает первые *n* кортежей. В противном случае порядок возвращенных кортежей не определен. Оператор `LIMIT` не подпадает ни под какую категорию, так как, безусловно, не является оператором чтения/записи, но и настоящим реляционным оператором его не назовешь. Мы включили его из чисто прагматических соображений, чтобы читатель, который ищет информацию об операторе `DUMP` (см. ниже), не забывал использовать перед ним оператор `LIMIT`.
`DUMP`	`DUMP alias;` Выводит содержимое отношения. Применяется преимущественно для отладки. Отношение должно быть невелико, иначе не поместится на экране. К псевдониму можно применить оператор `LIMIT`, чтобы ограничить объем выводимой на экран информации.
`STORE`	`STORE alias INTO 'directory' [USING function];` Сохраняет данные из отношения в каталоге. Перед выполнением этой команды указанного каталога не должно быть. Pig создаст каталог и сохранит отношение в файлах с именами `part-`*nnnnn*. По умолчанию использует функцию сохранения `PigStorage`, если противное не указано с помощью опции `USING`.

Надо полагать, загрузка и сохранение данных не поразили вас до глубины души. Но попробуем выполнить несколько команд обработки и посмотрим, как можно использовать Pig Latin в контексте Grunt.

```
grunt> log = LOAD 'tutorial/data/excite-small.log'
           ➥ AS (user:chararray, time:long, query:chararray);
grunt> grpd = GROUP log BY user;
grunt> cntd = FOREACH grpd GENERATE group, COUNT(log);
grunt> STORE cntd INTO 'output';
```

Показанные выше команды подсчитывают, сколько поисковых запросов поступило от каждого пользователя. В выходные файлы (просматривать файл следует вне Grunt) будет записана примерно такая информация:

```
002BB5A52580A8ED    18
005BD9CD3AC6BB38    18
00A08A54CD03EB95    3
011ACA65C2BF70B2    5
01500FAFE317B7C0    15
0158F8ACC570947D    3
018FBF6BFB213E68    1
```

Концептуально мы только что произвели операцию агрегирования, аналогичную следующему SQL-запросу:

```
SELECT user, COUNT(*) FROM excite-small.log GROUP BY user;
```

Но стоит сразу отметить два существенных различия между вариантами на Pig Latin и SQL. Ранее мы отмечали, что Pig Latin – язык обработки данных. Вы описываете последовательность шагов обработки, а не формулируете сложный SQL-запрос с различными фразами. Второе различие более тонкое – у любого отношения в SQL имеется фиксированная схема, которая определяется еще до того, как отношение заполняется данными. В Pig подход к схемам гораздо либеральнее. Если не хотите, можете вообще ими не пользоваться, и так часто бывает при обработке полуструктурированных и неструктурированных данных. В данном случае мы задали схему для отношения `log`, но только в команде загрузки; до загрузки данных эта схема не навязывается. Если во время загрузки обнаружено поле, не согласующееся со схемой, то вместо него записывается null. Таким образом, гарантируется, что во всех последующих операциях мы будем иметь дело с отношением `log`, удовлетворяющим заданной схеме.

Pig прилагает все усилия к тому, чтобы выявить схему отношения, исходя из операции, с помощью которой оно было создано. Узнать о том, какую схему вывел Pig, можно, выполнив для отношения

команду DESCRIBE. Это бывает полезно, когда хочется понять, что делает команда Pig. Например, можно поинтересоваться схемами grpd и cntd. Но предварительно посмотрим, как команда DESCRIBE описывает отношение log.

```
grunt> DESCRIBE log;
log: {user: chararray,time: long,query: chararray}
```

Как и следовало ожидать, команда LOAD ассоциировала с log ту схему, которую мы явно указали. Это отношение состоит из трех полей: user, time и query. Поля user и query строковые (chararray в терминологии Pig), а time – длинное целое (long).

Операция GROUP BY, примененная к отношению log, порождает отношение grpd. Зная эту операцию и схему log, Pig выводит схему grpd:

```
grunt> DESCRIBE grpd;
grpd: {group: chararray,log: {user: chararray,time: long,query: chararray}}
```

В этом отношение два поля: group и log. Поле log – *неупорядоченное множество* (bag) с подполями user, time и query. Мы еще не рассматривали систему типов в языке Pig и операцию GROUP BY, поэтому не ожидаем, что вы понимаете эту схему. Но суть в том, что схемы отношений в Pig могут быть довольно сложными, и команда DESCRIBE окажет неоценимую помощь в понимании того, с чем вы имеете дело.

```
grunt> DESCRIBE cntd;
cntd: {group: chararray,long}
```

Наконец, команда FOREACH применяется к отношению grpd и порождает отношение cntd. Взглянув на распечатку cntd, мы видим, что в нем два поля: идентификатор пользователя и количество запросов. DESCRIBE показывает, что в схеме, выведенной Pig для cntd, тоже два поля. Первое, group, взято из схемы grpd. Второе – безымянное типа long. Это поле генерируется функцией COUNT, а никакая функция не присваивает имя автоматически, хотя и сообщает Pig о типе (в данном случае long).

Если команда DESCRIBE рассказывает о схеме отношения, то ILLUSTRATE производит выборочный прогон, шаг за шагом показывая, как Pig будет вычислять отношение. Pig пытается имитировать выполнение команд, вычисляющих отношение, но берет только малую выборку из данных, чтобы ускорить процесс. Чтобы лучше понять, как работает ILLUSTRATE, применим ее к отношению cntd (формат вывода изменен, чтобы не выходить за пределы страницы).

```
grunt> ILLUSTRATE cntd;
-------------------------------------------------------------------
| log | user: bytearray  | time: bytearray | query: bytearray   |
-------------------------------------------------------------------
|     | 0567639EB8F3751C | 970916161410    | "conan o'brien"    |
|     | 0567639EB8F3751C | 970916161413    | "conan o'brien"    |
|     | 972F13CE9A8E2FA3 | 970916063540    | finger AND download |
-------------------------------------------------------------------
-----------------------------------------------------------------
| log | user: chararray  | time: long   | query: chararray    |
-----------------------------------------------------------------
|     | 0567639EB8F3751C | 970916161410 | "conan o'brien"     |
|     | 0567639EB8F3751C | 970916161413 | "conan o'brien"     |
|     | 972F13CE9A8E2FA3 | 970916063540 | finger AND download |
-----------------------------------------------------------------
-------------------------------------------------------------------
| grpd  | group: chararray | log: bag({user: chararray,time: long, |
|       |                  |           query: chararray})          |
-------------------------------------------------------------------
|       | 0567639EB8F3751C | {(0567639EB8F3751C, 970916161410,     |
|       |                  |   "conan o'brien"),                   |
|       |                  |  (0567639EB8F3751C,970916161413,      |
|       |                  |   "conan o'brien")}                   |
|       | 972F13CE9A8E2FA3 | {(972F13CE9A8E2FA3, 970916063540,     |
|       |                  |   finger AND download)}               |
-------------------------------------------------------------------
--------------------------------------
| cntd | group: chararray | long     |
--------------------------------------
|      | 0567639EB8F3751C | 2        |
|      | 972F13CE9A8E2FA3 | 1        |
--------------------------------------
```

`ILLUSTRATE` показывает, что для получения `cntd` нужно произвести четыре преобразования. Заголовок каждой таблицы описывает схему отношения, получающегося после преобразования, а в остальных строках показаны выборочные данные. Отношение `log` фигурирует в двух преобразованиях. Сами данные при этом не изменяются, зато изменяется схема – место обобщенного типа `bytearray` (так Pig представляет двоичные объекты) занимает заданная нами схема. Операция `GROUP` над отношением `log`, примененная к трем выборочным кортежам, порождает отношение `grpd`. Зная это, мы можем сделать вывод, что операция `GROUP` взяла поле `user` и преобразовала его в поле `group`. Кроме того, все принадлежащие `log` кортежи с одинаковым значением `user` объединены в множество в отношении `grpd`. Видя выборочные данные в имитированном командой `ILLUSTRATE` прогоне, можно лучше понять смысл различных операций. Наконец,

мы видим, что операция FOREACH, примененная к grpd, дает cntd. Взглянув на данные в grpd, показанные в предыдущей таблице, мы сразу же заключаем, что функция COUNT() вычисляет размер каждого множества.

Команды DESCRIBE и ILLUSTRATE играют основную роль в понимании предложений языка Pig Latin, но в Pig имеется также команда EXPLAIN, которая показывает подробный логический и физический план выполнения. Все диагностические операторы приведены в табл. 10.3.

Таблиц 10.3. Диагностические операторы в языке Pig Latin

DESCRIBE	`DESCRIBE alias;` Выводит схему отношения.
EXPLAIN	`EXPLAIN [-out path] [-brief] [-dot] [-param ...] [-param_file ...] alias;` Выводит план выполнения, приводящий к вычислению отношения. Если указать в качестве аргумента имя скрипта, например `EXPLAIN myscript.pig`, то будет показан план выполнения этого скрипта.

ILLUSTRATE	`ILLUSTRATE alias;`
	Выводит пошаговую последовательность преобразований, необходимых для получения результирующего отношения (начиная с команды загрузки). Чтобы обработка не занимала слишком много времени, а результаты были обозримыми, исполнение имитируется на небольшой (не совсем случайной) выборке из входных данных.
	Если случится так, что ни один элемент взятой первоначально выборки в результате работы скрипта не попадает в выходные данные, то Pig генерирует фиктивные начальные данные, так чтобы результирующий псевдоним оказался непустым. Рассмотрим, к примеру, такие операции:
	`A = LOAD 'student.data' as (name, age);`
	`B = FILTER A by age > 18;`
	`ILLUSTRATE B;`
	Если окажется, что в каждом кортеже, включенном Pig в выборку для A, возраст меньше или равен 18, то псевдоним B будет пуст, так что никакой «иллюстрации» не получится. Тогда Pig искусственно сконструирует в A несколько кортежей, в которых возраст больше 18. В этом случае B будет непустым отношением, и пользователь сможет увидеть, как скрипт работает.
	Чтобы команда ILLUSTRATE могла работать, команда LOAD на первом шаге должна включать схему. Последующие преобразования не должны содержать операторов LIMIT, SPLIT, вложенных операторов FOREACH или данных типа map (см. раздел 10.5.1).

10.5. Учимся говорить на Pig Latin

Итак, вы знаете, как с помощью Grunt выполнять команды Pig Latin и исследовать порядок их исполнения и результаты. Теперь можно вернуться назад и описать язык более формально. Не забывайте пользоваться Grunt для экспериментов с описываемым понятиями по ходу их изложения.

10.5.1. Типы данных и схемы

Сначала рассмотрим типы данных Pig, начиная с простейших. В языке Pig есть шесть простых атомарных типов и три составных. Они перечислены в таблицах 10.4 и 10.5 соответственно. К числу атомарных относятся числовые скалярные типы, а также строки и бинарные объекты. Если не указано противное, то любое поле по умолчанию имеет тип `bytearray`.

Таблица 10.4. Атомарные типы данных в языке Pig Latin

`int`	32-разрядное целое со знаком
`long`	64-разрядное целое со знаком
`float`	32-разрядное число с плавающей точкой
`double`	64-разрядное число с плавающей точкой
`chararray`	Массив символов (строка) в кодировке UTF-8
`bytearray`	Массив байтов (двоичный объект)

К составным типам относятся tuple (кортеж), bag (множество) и map (словарь).

Поле в кортеже или значение в словаре могут принимать значение null или принадлежать любому атомарному или составному типу. Хотя структуры данных могут быть произвольно сложными, некоторые определенно более полезны и встречаются чаще, а глубина вложенности редко превышает два уровня. В рассмотренном выше примере журнала Excite оператор `GROUP BY` породил отношение `grpd`, в котором каждый кортеж включает поле типа `bag`. Схема этого отношения покажется вам более естественной, если рассматривать каждую запись `grpd` как историю одного пользователя. Тогда кортеж будет представлять пользователя, а множество – совокупность запросов этого пользователя.

Таблица 10.5. Составные типы данных в языке Pig Latin

`Tuple`	`(12.5,hello world,-2)` Кортеж – это упорядоченное множество полей. Чаще всего он используется для описания строки отношения. Записывается в виде заключенной в круглые скобки последовательности полей, разделенных запятыми.
`Bag`	`{(12.5,hello world,-2),(2.87,bye world,10)}` `Bag` – это неупорядоченное множество кортежей. Отношение представляет собой частный случай множества, который иногда называют внешним множеством. Внутренним называют множество, являющееся полем некоторого объемлющего составного типа. Множество записывают в виде заключенной в фигурные скобки последовательности кортежей, разделенных запятыми. Кортежи, являющиеся элементами множествами, необязательно должны описываться одной и той же схемой или хотя бы содержать одно и то же количество полей. Однако это все же предпочтительно, если только вы не имеете дело с полуструктурированными или неструктурированными данными.
`Map`	`[key#value]` Словарь – это совокупность пар ключ/значение. Ключи должны быть уникальными и строковыми `(chararray)`. Значения могут иметь любой тип.

Можно также взглянуть на модель данных Pig сверху вниз. На верхнем уровне находятся команды Pig Latin для работы с отношениями, то есть с множествами кортежей. Если количество полей во всех кортежах одинаково и каждое поле имеет атомарный тип, то мы получаем реляционную модель данных, то есть отношение можно рассматривать как таблицу, кортежи – как строки (записи), а поля – как столбцы. Но модель данных в Pig более сложная и гибкая – она допускает *вложенные* типы данных. Поля сами могут быть кортежами, множествами или словарями. Словари особенно полезны при обработке полуструктурированных данных, например представленных в формате JSON или XML, а также разреженных реляционных данных. Кроме того, не требуется, чтобы все кортежи в множестве состояли из одного и того же числа полей. Это дает возможность представлять неструктурированные данные.

В схеме для полей можно задавать не только типы, но и имена, чтобы на них было проще ссылаться. Пользователь может определить схему отношения с помощью ключевого слова `AS` в операторах `LOAD`, `STREAM` и `FOREACH`. Например, в команде `LOAD`

для загрузки журнала запросов Excite мы определили типы данных всех полей в отношении `log`, а также назвали поля `user`, `time` и `query`.

```
grunt> log = LOAD 'tutorial/data/excite-small.log'
           ➥ AS (user:chararray, time:long, query:chararray);
```

Если в определении схемы опустить тип, то Pig по умолчанию возьмет тип `bytearray`, как наиболее общий. Имя также можно опускать, тогда поле останется неименованным, и сослаться на него можно будет только по позиции.

10.5.2. Выражения и функции

К полям данных можно применять выражения и функции для вычисления различных значений. Простейшее выражение – это константа. Вслед за ним идет значение поля. Значение именованного поля можно получить, просто сославшись на него по имени, а значение неименованного поля – с помощью обозначения `$n`, где *n* – позиция поля в кортеже (позиции нумеруются, начиная с 0). Например, следующая команда `LOAD` назначает полям отношения `log` имена с помощью схемы:

```
log = LOAD 'tutorial/data/excite-small.log'
    ➥ AS (user:chararray, time:long, query:chararray);
```

Здесь мы встречаем три именованных поля: user, time и query. К полю time можно обратиться двумя способами: `time` или `$1`, поскольку оно второе в отношении `log` (номер позиции равен 1). Предположим, что нужно поместить поле time в отдельное отношение; для этого можно написать такую команду:

```
projection = FOREACH log GENERATE time;
```

Тот же результат мы можем получить и таким способом:

```
projection = FOREACH log GENERATE $1;
```

Как правило, поля следует именовать. Один из случаев, когда приходится ссылаться на поля по номеру позиции, – обработка неструктурированных данных. При работе с составными типами для ссылки на поле, вложенное в кортеж или множество, применяется точка. Напомним, что выше мы группировали журнал Excite по идентификаторам пользователей и получили в результате отношение `grpd` с иерархической схемой.

```
-----------------------------------------------------------------
| grpd | group: chararray | log: bag({user: chararray,time: long,|
|      |                  |           query: chararray})        |
-----------------------------------------------------------------
|      | 0567639EB8F3751C | {(0567639EB8F3751C, 970916161410,    |
|      |                  |   "conan o'brien"),                  |
|      |                  | (0567639EB8F3751C,970916161413,      |
|      |                  |   "conan o'brien")}                  |
|      | 972F13CE9A8E2FA3 | {(972F13CE9A8E2FA3, 970916063540,    |
|      |                  |   finger AND download)}              |
-----------------------------------------------------------------
```

Второе поле в `grpd` называется `log` и имеет тип `bag`. Каждое множество состоит из кортежей с тремя именованными полями: user, time и query. В этом отношении выражение `log.query` для первого кортежа возвратило бы две копии строки «conan o'brien». По-другому тот же результат можно было бы получить, написав `log.$2`.

На поля внутри множества следует ссылаться с помощью оператора `#`, а не точка. Так, значение, ассоциированное с ключом `k` в множестве `m`, можно получить, написав `m#k`.

Но возможность ссылаться на значения – это только первый шаг. Pig поддерживает стандартные арифметические операции, сравнение, условные выражения, приведение типов и булевские операции – как в любом распространенном языке программирования (табл. 10.6).

Таблица 10.6. Выражения в языке Pig Latin

Константы	`12, 19.2, 'hello world'`	Постоянные значения, например, 19 или «hello world». Считается, что число без десятичной точки имеет тип `int`, если не оканчивается суффиксом `l` или `L`, означающим, что его тип `long`. Число с десятичной точкой имеет тип `double`, если не оканчивается суффиксом `f` или `F`, означающим, что его тип `float`.
Основные арифметические операции	`+,-,*,/`	Сложение, вычитание, умножение и деление.
Знак числа	`+x, -x`	Минус (`-`) изменяет знак числа на противоположный.
Приведение типа	`(t)x`	Преобразует значение `x` к типу `t`.
Деление по модулю	`x % y`	Возвращает остаток от деления `x` на `y`.
Условное выражение	`(x ? y : z)`	Возвращает `y`, если `x` истинно, иначе `z`. Выражение должно быть заключено в скобки.

Таблица 10.6. (окончание)

Операции сравнения	`==, !=, <, >, <=, >=`	Равно, не равно, меньше, больше, меньше или равно, больше или равно.
Сопоставление с образцом	`x matches regex`	Сопоставление строки `x` с регулярным выражением `regex`. Регулярные выражения записываются в синтаксисе Java (класс `java.util.regex.Pattern`).
Null	`x is null, x is not null`	Проверяет совпадение (или несовпадение) `x` с `null`.
Булевские операции	`x and y x or y not x`	И, или, не.

Pig поддерживает также функции. В табл. 10.7 приведены встроенные в Pig функции, большая часть которых не нуждается в пояснениях. Функции, определенные пользователем (UDF), мы будем рассматривать в разделе 10.6.

Таблица 10.7. Функции, встроенные в язык Pig Latin

`AVG`	Вычисляет среднее числовых значений по множеству с одним столбцом.
`CONCAT`	Конкатенирует две строки (`chararray`) или два двоичных объекта (`bytearray`).
`COUNT`	Вычисляет количество кортежей в множестве. Для других типов см. `SIZE`.
`DIFF`	Сравнивает два поля в кортеже. Если эти поля являются множествами, то возвращает кортежи, которые присутствуют в одном множестве, но отсутствуют в другом. Если поля являются значениями, возвращает кортежи, в которых они не совпадают.
`MAX`	Вычисляет максимальное значение в множестве с одним столбцом. Столбец должен быть числовым или типа `chararray`.
`MIN`	Вычисляет минимальное значение в множестве с одним столбцом. Столбец должен быть числовым или типа `chararray`.
`SIZE`	Вычисляет количество элементов. Для множества возвращается число кортежей в нем, для кортежа – количество элементов, для `chararray` – число символов, для `bytearray` – число байтов, для числовых скаляров – 1.
`SUM`	Вычисляет сумму числовых значений по множеству с одним столбцом.
`TOKENIZE`	Разбивает строку (chararray), возвращая множество слов (каждое слово представляет собой кортеж). Разделителями считаются пробел, двойная кавычка (`"`), запятая, скобка и звездочка (`*`).
`IsEmpty`	Проверяет, является ли множество или кортеж пустым.

Выражения и функции нельзя использовать изолированно. Они должны применяться в сочетании с реляционными операторами для преобразования данных.

10.5.3. Реляционные операторы

Самая интересная особенность языка Pig Latin – наличие в нем реляционных операторов. Именно они превращают его в язык обработки данных. Сначала бегло рассмотрим простые операторы, чтобы освоиться с их стилем и синтаксисом. А затем перейдем к более сложным операторам, таким, как `COGROUP` и `FOREACH`.

Оператор `UNION` объединяет несколько отношений, а `SPLIT` разбивает одно отношение на несколько. Поясним на примере:

```
grunt> a = load 'A' using PigStorage(',') as (a1:int, a2:int, a3:int);
grunt> b = load 'B' using PigStorage(',') as (b1:int, b2:int, b3:int);
grunt> DUMP a;
(0,1,2)
(1,3,4)
grunt> DUMP b;
(0,5,2)
(1,7,8)
grunt> c = UNION a, b;
grunt> DUMP c;
(0,1,2)
(0,5,2)
(1,3,4)
(1,7,8)
grunt> SPLIT c INTO d IF $0 == 0, e IF $0 == 1;
grunt> DUMP d;
(0,1,2)
(0,5,2)
grunt> DUMP e;
(1,3,4)
(1,7,8)
```

Оператор `UNION` не устраняет дубликаты. Если это необходимо, воспользуйтесь оператором `DISTINCT`. Операция `SPLIT`, примененная к `c`, отправляет кортеж в `d`, если его первое поле (`$0`) равно 0, и в `e` – если оно равно 1. Условия могут быть такими, что некоторые строки попадают и в `d`, и в `e`, а некоторые не попадают никуда. Поведение `SPLIT` можно имитировать с помощью нескольких операторов `FILTER`. Оператор `FILTER` отбирает из отношения только кортежи, удовлетворяющие некоторому условию:

```
grunt> f = FILTER c BY $1 > 3;
```

```
grunt> DUMP f;
(0,5,2)
(1,7,8)
```

Мы уже видели, что оператор LIMIT выбирает из отношения заданное число кортежей. Оператор SAMPLE производит случайную выборку кортежей из отношения в количестве, определяемом заданной процентной долей.

Рассмотренные до сих пор операторы были относительно простыми в том смысле, что кортеж в них рассматривался как атомарная единица. В более сложных случаях возникает необходимость работать с группами кортежей. Далее мы рассмотрим операторы группировки. В отличие от предыдущих, они создают на выходе новые схемы, в которых активно используются множества и вложенные типы данных. К работе со сгенерированными схемами нужно еще привыкнуть. Имейте в виду, что операторы группировки почти всегда применяются для порождения промежуточных данных. То, что они создают, – лишь временные объекты на пути к вычислению окончательных результатов.

Самым простым является оператор GROUP. Продолжим работать с теми же отношениями, что и выше.

```
grunt> g = GROUP c BY $2;
grunt> DUMP g;
(2,{(0,1,2),(0,5,2)})
(4,{(1,3,4)})
(8,{(1,7,8)})
grunt> DESCRIBE c;
c: {a1: int,a2: int,a3: int}
grunt> DESCRIBE g;
g: {group: int,c: {a1: int,a2: int,a3: int}}
```

Мы создали новое отношение g путем группировки кортежей из c с одинаковым значением в третьем столбце ($2, или a3). GROUP всегда порождает отношение с двумя полями. Первое поле – групповой ключ, в данном случае a3, а второе – множество, содержащее все кортежи с одним и тем же значением группового ключа. Из распечатки g видно, что в нем есть три кортежа, по числу уникальных значений в третьем столбце c. Множество в первом кортеже содержит все кортежи c, для которых в третьем столбце находится 2; множество во втором кортеже – все кортежи c, для которых в третьем столбце находится 4. И так далее. Поняв, как порождаются данные в отношении g, вы легко разберетесь с его схемой. Первое поле отношения, порождаемого оператором GROUP, всегда называется group. В данном случае было бы естественнее назвать первое поле a3, но Pig не позволяет задавать

вместо group другое имя. Придется вам привыкнуть к имени group. Второе поле порожденного GROUP отношения называется по имени отношения, к которому GROUP применялся, в данном случае c, и, как мы уже отмечали, его тип – множество. Поскольку это множество служит для хранения кортежей из c, то его схема в точности совпадает со схемой c – три целочисленных поля с именами a1, a2 и a3.

Перед тем как двигаться дальше, отметим, что группировку в операторе GROUP можно производить по любой функции или выражению. Например, если поле time содержит временную метку и существует функция DayOfWeek, то можно выполнить группировку так, что получится отношение с семью кортежами:

```
GROUP log BY DayOfWeek(time);
```

Наконец, можно поместить все кортежи отношения в одно большое множество. Это полезно для агрегирования отношений, так как функции работают с множествами, а не с отношениями. Например:

```
grunt> h = GROUP c ALL;
grunt> DUMP h;
(all,{(0,1,2),(0,5,2),(1,3,4),(1,7,8)})
grunt> i = FOREACH h GENERATE COUNT($1);
grunt> dump i;
(4L)
```

Это один из способов подсчитать число кортежей в c. Первое поле в множестве, которое выводит GROUP ALL, всегда содержит строку all.

Освоившись с оператором GROUP, мы можем перейти к оператору COGROUP, который группирует кортежи *из нескольких отношений*. Это очень похоже на операцию соединения. Например, выполним совместную группировку a и b по третьему столбцу.

```
grunt> j = COGROUP a BY $2, b BY $2;
grunt> DUMP j;
(2,{(0,1,2)},{(0,5,2)})
(4,{(1,3,4)},{})
(8,{},{(1,7,8)})
grunt> DESCRIBE j;
j: {group: int,a: {a1: int,a2: int,a3: int},b: {b1: int,b2: int,b3: int}}
```

Если множество, порождаемое оператором GROUP, всегда содержит два поля, то COGROUP порождает три поля (или больше, если применяется более чем к двум отношениям). Первое – групповой ключ, а второе и третье – множества, содержащие кортежи из совместно сгруппированных отношений с одинаковым групповым ключом. Если кортежи

с некоторым групповым ключом присутствуют в одном отношении, но отсутствуют в другом, то в поле, соответствующем отношению без этого ключа, будет пустое множество. Если требуется игнорировать групповые ключи, отсутствующие в одном из соединяемых отношений, то можно добавить ключевое слово INNER, например:

```
grunt> j = COGROUP a BY $2, b BY $2 INNER;
grunt> dump j;
(2,{(0,1,2)},{(0,5,2)})
(8,{},{(1,7,8)})
grunt> j = COGROUP a BY $2 INNER, b BY $2 INNER;
grunt> dump j;
(2,{(0,1,2)},{(0,5,2)})
```

Концептуально можно считать, что по умолчанию оператор COGROUP ведет себя как операция внешнего соединения, а ключевое слово INNER превращает его в левое внешнее соединение, правое внешнее соединение или внутреннее соединение. Другой способ получить внутреннее соединение в Pig – воспользоваться оператором JOIN. Основное различие между JOIN и COGROUP с модификатором INNER состоит в том, что JOIN создает плоское множество выходных записей, в чем можно убедиться, взглянув на схему:

```
grunt> j = JOIN a BY $2, b BY $2;
grunt> dump j;
(0,1,2,0,5,2)
grunt> DESCRIBE j;
j: {a::a1: int,a::a2: int,a::a3: int,b::b1: int,b::b2: int,b::b3: int}
```

И последний реляционный оператор, с которым мы познакомимся, – FOREACH. Он перебирает все кортежи входного отношения и порождает новые кортежи в выходном. Но за этой кажущейся простотой скрыта поразительная мощь, особенно когда оператор применяется к составным типам данных, порождаемым операторами группировки. Существует даже вложенная форма FOREACH, предназначенная для обработки составных типов. Рассмотрим разные способы применения FOREACH к простым отношениям.

```
grunt> k = FOREACH c GENERATE a2, a2 * a3;
grunt> DUMP k;
(1,2)
(5,10)
(3,12)
(7,56)
```

За FOREACH всегда следует псевдоним (имя, присвоенное отношению), а за ним – ключевое слово GENERATE. Выражение, находящееся

после GENERATE, управляет выводом. В простейшем случае FOREACH используется для проецирования отдельных столбцов входного отношения на выходное. Но можно указывать произвольные выражения; в примере выше это было умножение. Для отношений с вложенными множествами (например, порождаемыми операторами группировки) у FOREACH имеется специальный синтаксис проецирования и расширенный набор функций. Вот как выглядит вложенная проекция, сохраняющая только первое поле каждого множества:

```
grunt> k = FOREACH g GENERATE group, c.a1;
grunt> DUMP k;
(2,{(0),(0)})
(4,{(1)})
(8,{(1)})
```

А вот как можно получить два поля из каждого множества:

```
grunt> k = FOREACH g GENERATE group, c.(a1,a2);
grunt> DUMP k;
(2,{(0,1),(0,5)})
(4,{(1,3)})
(8,{(1,7)})
```

Большинство встроенных в Pig функций ориентированы на работу с множествами:

```
grunt> k = FOREACH g GENERATE group, COUNT(c);
grunt> DUMP k;
(2,2L)
(4,1L)
(8,1L)
```

Напомним, что g получилось группировкой c по третьему столбцу. Поэтому такая команда FOREACH порождает счетчики вхождений для каждого значения, встречающегося в третьем столбце c. Как уже было сказано, операторы группировки предназначены в основном для генерации промежуточных данных, которые впоследствии будут упрощены за счет применения других операторов, например FOREACH. COUNT – одна из агрегатных функций. Ниже мы увидим, что с помощью механизма UDF можно определить много других функций.

Функция FLATTEN предназначена для «разглаживания» вложенных типов данных. Синтаксически она выглядит как функция, например COUNT или AVG, но на самом деле является специальным оператором, поскольку может изменять структуру результата, созданного оператором FOREACH...GENERATE. Поведение зависит от того, как этот оператор применяется и к чему именно. Рассмотрим, к

примеру, отношение с кортежами вида (a, (b, c)). Команда `FOREACH... GENERATE $0, FLATTEN($1)` для каждого входного кортежа создаст один выходной вида (a, b, c).

В применении к множествам `FLATTEN` модифицирует команду `FOREACH...GENERATE`, так что она порождает новые кортежи. Он убирает один уровень вложенности, то есть по своему действию почти противоположен операциям группировки. Если множество содержит *N* кортежей, то после разглаживания само множество исключается, а вместо него образуется *N* кортежей.

```
grunt> k = FOREACH g GENERATE group, FLATTEN(c);
grunt> DUMP k;
(2,0,1,2)
(2,0,5,2)
(4,1,3,4)
(8,1,7,8)
grunt> DESCRIBE k;
k: {group: int,c::a1: int,c::a2: int,c::a3: int}
```

На операцию `FLATTEN` можно взглянуть и по-другому, заметив, что она порождает перекрестное произведение. Такая точка зрения полезна, когда мы используем `FLATTEN` несколько раз в одной команде `FOREACH`. Допустим, к примеру, что мы каким-то образом создали отношение `l`.

```
grunt> dump l;
(1,{(1,2)},{(3)})
(4,{(4,2),(4,3)},{(6),(9)})
(8,{(8,3),(8,4)},{(9)})
grunt> describe l;
d: {group: int,a: {a1: int,a2: int},b: {b1: int}}
```

Тогда следующая команда разглаживания двух множеств выводит все комбинации кортежей, взятых по одному из каждого множества:

```
grunt> m = FOREACH l GENERATE group, FLATTEN(a), FLATTEN(b);
grunt> dump m;
(1,1,2,3)
(4,4,2,6)
(4,4,2,9)
(4,4,3,6)
(4,4,3,9)
(8,8,3,9)
(8,8,4,9)
```

Мы видим, что кортеж с групповым ключом 4 в отношении `l` содержит множества размера 2 в обоих полях `a` и `b`. Поэтому резуль-

тирующее отношение m, являющее полным перекрестным произведением, содержит четыре строки.

Наконец, существует вложенная форма FOREACH для более сложной обработки множеств. Предположим, что имеется отношение (назовем его l), в котором одно из полей (a) является множеством, а оператор FOREACH с вложенным блоком имеет такой вид:

```
alias = FOREACH l {
            tmp1 = operation on a;
            [more operations...]
            GENERATE expr [, expr...]
        }
```

Команда GENERATE всегда должна находиться в конце вложенного блока. Она порождает какой-то результат для каждого кортежа в l. Операции внутри вложенного блока могут создавать новые отношения, основываясь на множестве a. Например, можно породить некоторое подмножество множества a в каждом элементе кортежа l.

```
grunt> m = FOREACH l {
        tmp1 = FILTER a BY a1 >= a2;
        GENERATE group, tmp1, b;
   }
grunt> DUMP m;
(1,{},{(3)})
(4,{(4,2),(4,3)},{(6),(9)})
(8,{(8,3),(8,4)},{(9)})
```

В одном вложенном блоке может быть несколько команд, причем они могут даже работать с разными множествами:

```
grunt> m = FOREACH l {
        tmp1 = FILTER a BY a1 >= a2;
        tmp2 = FILTER b by $0 < 7;
        GENERATE group, tmp1, tmp2;
   };
grunt> DUMP m;
(1,{},{(3)})
(4,{(4,2),(4,3)},{(6)})
(8,{(8,3),(8,4)},{})
```

Когда писалась эта книга, во вложенном блоке разрешалось употреблять только пять операторов: DISTINCT, FILTER, LIMIT, ORDER, SAMPLE. Ожидается, что в будущем будут поддержаны и другие.

Примечание. Иногда входные и выходные схемы FOREACH радикально различаются. В таких случаях пользователь может задать выходную схему с помощью ключевого слова AS после каждого поля. Этот синтаксис отличается от применяемого в команде LOAD, где схема задается в виде списка после ключевого слова AS, но в любом случае схема описывается с помощью AS.

В табл. 10.8 перечислены реляционные операторы, имеющиеся в языке Pig Latin. Во многих операторах встречается опция `PARALLEL n`. Число `n` – это желаемая степень параллелизма оператора. На практике `n` равно числу заданий редукции в задаче Hadoop, которую создает Pig. Если `n` не задано явно, то по умолчанию берется параметр кластера Hadoop. В документации по Pig рекомендуется вычислять `n` по следующей формуле:

```
n = (#nodes - 1) * 0.45 * RAM
```

где `#nodes` – количество узлов, а `RAM` – объем памяти каждого узла в гигабайтах.

Таблица 10.8. Реляционные операторы в языке Pig Latin

`SPLIT`	`SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression ...];` Разбивает одно отношение на два или более в соответствии с заданными булевскими выражениями. Отметим, что кортеж может попасть в несколько результирующих отношений или ни в одно из них.
`UNION`	`alias = UNION alias, alias, [, alias ...]` Создает объединение двух или более отношений. Отметим, что: ❑ как и для любого отношения, порядок кортежей не определен; ❑ не требуется, чтобы отношения имели одинаковую схему или хотя бы одно и то же количество полей; ❑ кортежи-дубликаты не удаляются.
`FILTER`	`alias = FILTER alias BY expression;` Выбирает кортежи, соответствующие булевскому выражению. Используется как для выборки интересующих кортежей, так и для исключения не интересующих.
`DISTINCT`	`alias = DISTINCT alias [PARALLEL n];` Удаляет кортежи-дубликаты.

`SAMPLE`	`alias = SAMPLE alias factor;`
	Строит случайную выборку из отношения. Коэффициент выборки задается параметром `factor`. Например, чтобы выбрать 1% записей из отношения `large_data`, следует написать
	`small_data = SAMPLE large_data 0.01;`
	Операция недетерминированная, то есть размер `small_data` может оказаться не в точности равен 1% размера `large_data`, и не гарантируется, что каждый раз будет порождаться одно и то же количество кортежей.

Таблица 10.8. (продолжение)

FOREACH	`alias = FOREACH alias GENERATE expression [,expression ...] [AS schema];` Перебирает все кортежи исходного отношения и порождает новые. Обычно применяется для преобразования столбцов данных, например, добавления или удаления полей. Для выходного отношения можно задать необязательную схему, например, для присваивания имен новым полям.

FOREACH (вложенный)	`alias = FOREACH nested_alias {`
	`  alias = nested_op;`
	`  [alias = nested_op; ...]`
	`  GENERATE expression [, expression ...];`
	`};`
	Перебирает все кортежи отношения `nested_alias` и порождает новые.
	Хотя бы одно из полей `nested_alias` должно быть множеством. К внутренним множествам в `nested_op` разрешается применять операторы `DISTINCT`, `FILTER`, `LIMIT`, `ORDER` и `SAMPLE`.

JOIN	`alias = JOIN alias BY field_alias, alias BY field_alias [, alias BY field_alias …] [USING "replicated"] [PARALLEL n];` Вычисляет внутреннее соединение двух или более отношений по значениям общих полей. При использовании опции репликации Pig сохраняет все отношения, кроме первого, в памяти для ускорения обработки. Вы должны следить за тем, чтобы совокупный размер меньших отношений был достаточно мал для размещения в памяти. Если входные отношения, являющиеся операндами `JOIN`, плоские, то и выходное будет плоским. Кроме того, число полей в выходном отношении равно сумме чисел полей во входных отношениях, а схема выходного отношения представляет собой конкатенацию схем входных отношений.
GROUP	`alias = GROUP alias { [ALL] | [BY {[field_alias [, field_alias]] | * | [expression]] } [PARALLEL n];` В одном отношении группирует кортежи с одинаковым групповым ключом. Обычно групповой ключ состоит из одного или нескольких полей, но это может быть также весь кортеж (`*`) или выражение. Конструкция `GROUP alias ALL` позволяет поместить все кортежи в одну группу. Выходное отношение состоит из двух полей с автоматически генерируемыми именами. Первое поле всегда называется «group», а его тип совпадает с типом группового ключа. Второе поле называется так же, как входное отношение, а его тип – множество. Схема этого множества совпадает со схемой входного отношения.

Таблица 10.8. (окончание)

COGROUP	`alias = COGROUP alias BY field_alias [INNER | OUTER] , alias BY field_alias [INNER | OUTER] [PARALLEL n];` Группирует кортежи из двух или более отношений по общим групповым значениям. В выходном отношении будет присутствовать кортеж для каждого уникального группового значения. В каждом кортеже первым полем будет групповое значение. Второе поле – множество, содержащее кортежи из первого входного отношения с соответствующим групповым значением. То же относится к третьему и последующим полям выходного кортежа. По умолчанию подразумевается семантика внешнего соединения, когда в выходном отношении присутствуют все групповые значения, встречающиеся хотя бы в одном входном отношении. Если в каком-то входном отношении нет кортежа с некоторым групповым значением, то в соответствующем поле выходного кортежа появится пустое множество. Если для некоторого отношения задана опция `INNER`, то в выходном отношении будут присутствовать только кортежи, для которых в этом входном отношении соответствующее групповое значение представлено. В поле выходного кортежа, соответствующем этому отношению, не может быть пустого множества. Разрешается группировать по нескольким полям. Для этого `field_alias` должно быть заключенным в скобки списком имен полей через запятую. Операторы `COGROUP` (с опцией `INNER`) и `JOIN` аналогичны, но `COGROUP` порождает вложенные выходные кортежи.
CROSS	`alias = CROSS alias, alias [, alias …] [PARALLEL n];` Вычисляет (плоское) перекрестное произведение двух и более отношений. Это крайне дорогостоящая операция, которой всеми силами следует избегать.
ORDER	`alias = ORDER alias BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] } [PARALLEL n];` Сортирует отношение по одному или нескольким полям. Если извлечь отношение сразу после выполнения `ORDER` (с помощью `DUMP` или `STORE`), то оно гарантированно будет отсортировано в нужном порядке. В результате последующих операций обработки (`FILTER`, `DISTINCT` и т. д.) сортировка может быть нарушена.
STREAM	`alias = STREAM alias [, alias …] THROUGH {'command' | cmd_alias } [AS schema] ;` Обрабатывает отношение внешним скриптом.

Итак, мы познакомились с различными аспектами языка Pig Latin – типами данных, выражениями, функциями и реляционными операторами. Язык можно расширять путем определения пользова-

тельских функций. Но перед тем как переходить к обсуждению этой темы, сделаем несколько замечаний о компиляции и оптимизации программ на Pig Latin.

10.5.4. Оптимизация исполнения

Как и многие современные компиляторы, компилятор Pig может изменять последовательность исполнения ради оптимизации при условии, что получившийся план выполнения логически эквивалентен исходной программе. Например, представим себе программу, которая применяет дорогостоящую функцию (скажем, шифрование) к некоторому полю (скажем, номеру социального страхования) каждой записи, а затем функцию фильтрации для выборки записей по какому-то другому полю (например, чтобы оставить только людей, проживающих в определенном регионе). Тогда компилятор может поменять местами эти две операции; на конечный результат это не повлияет, а производительность существенно возрастет. После фильтрации количество подлежащих обработке записей сильно уменьшится, поэтому и шифровать придется меньше.

По мере развития Pig в компилятор будут добавляться новые способы оптимизации. Поэтому так важно всегда работать с последней версией. Однако возможности компилятора по оптимизации произвольного кода небезграничны. В онлайновой документации по Pig приведены советы, как повысить производительность. Перечень таких советов для Pig версии 0.3 можно найти по адресу http://hadoop.apache.org/pig/docs/r0.3.0/cookbook.html.

10.6. Определяемые пользователем функции

Неотъемлемой частью идеологии Pig Latin является расширяемость за счет определяемых пользователем функций (UDF). Для написания таких функций имеется набор четко определенных API. Но это не означает, что каждый должен писать все необходимые функции самостоятельно. Частью экосистемы языка Pig[6] является PiggyBank[7], онлайновый репозиторий, в который авторы могут помещать напи-

[6] Я думал назвать ее «Pig pen» (загон для свиней), но название PigPen уже зарезервировано для подключаемого к Eclipse модуля для редактирования скриптов на Pig Latin. См. http://wiki.apache.org/pig/PigPen.

[7] http://wiki.apache.org/pig/PiggyBank.

санные ими функции для общего пользования. Перед тем как разрабатывать свою функцию, проверьте, нет ли чего-нибудь похожего в PiggyBank. А написав UDF самостоятельно, вы, возможно, захотите поместить ее в PiggyBank и тем самым поделиться с сообществом Pig.

10.6.1. Использование UDF

В настоящее время UDF пишутся на Java и упаковываются в jar-файлы. Чтобы воспользоваться UDF, необходим jar-файл, содержащий ее класс (или классы). Например, из репозитория PiggyBank вы, скорее всего, получите файл piggybank.jar.

Прежде чем использовать UDF, ее jar-файл необходимо зарегистрировать в Pig командой `REGISTER`. Затем для обращения к UDF указывается полное имя ее Java-класса. Так, в PiggyBank есть функция `UPPER`, преобразующая строку в верхний регистр:

```
REGISTER piggybank/java/piggybank.jar;
b = FOREACH a GENERATE
      ➥ org.apache.pig.piggybank.evaluation.string.UPPER($0);
```

Если необходимо вызывать функцию в нескольких местах, то каждый раз выписывать полное имя класса утомительно. Для этой цели Pig предлагает команду `DEFINE`, позволяющую присвоить UDF имя. Следовательно, приведенный выше фрагмент можно переписать и так:

```
REGISTER piggybank/java/piggybank.jar;
DEFINE Upper org.apache.pig.piggybank.evaluation.string.UPPER();
b = FOREACH a GENERATE Upper($0);
```

В табл. 10.9 приведены команды, относящиеся к UDF.

Таблица 10.9. Относящиеся к UDF команды языка Pig Latin

`DEFINE`	`DEFINE alias { function | 'command' [...] };` Назначает псевдоним функции или команде.
`REGISTER`	`REGISTER alias;` Регистрирует UDF в Pig. В настоящее время UDF пишутся только на java, а *alias* – путь к JAR-файлу. Любую UDF необходимо зарегистрировать до первого обращения.

Если вы собираетесь только пользоваться UDF, написанными другими людьми, то больше ничего знать не нужно. Но если подходящей UDF найти не удалось, придется писать собственную.

10.6.2. Создание UDF

Pig поддерживает две категории UDF: вычислительные (eval)[8] и загрузки/сохранения. Функции загрузки/сохранения используются только в командах `LOAD` и `STORE`, чтобы помочь Pig читать и записывать данные в специальных форматах. Большинство UDF относятся к категории вычислительных функций, то есть принимают одно и возвращают другое значение.

В настоящее время писать UDF можно только с помощью предлагаемого Pig Java API[9]. Для создания вычислительной UDF необходимо написать Java-класс, расширяющий `EvalFunc<T>`. В нем имеется единственный абстрактный метод, который вам надлежит реализовать:

```
abstract public T exec(Tuple input) throws IOException;
```

Этот метод вызывается для каждого кортежа отношения, представленного объектом тип `Tuple`. Метод `exec()` обрабатывает кортеж и возвращает объект типа `T`, соответствующего одному из типов языка Pig Latin. Соответствие между типами Java и Pig Latin описано в табл. 10.10. Некоторые типы встроены в Java, другие являются расширениями, реализованными в Pig.

Таблица 10.10. Типы Pig Latin и эквивалентные им типы Java

Тип Pig Latin	Тип Java
`Bytearray`	`DataByteArray`
`Chararray`	`String`
`Int`	`Integer`
`Long`	`Long`
`Float`	`Float`
`Double`	`Double`
`Tuple`	`Tuple`
`Bag`	`DataBag`
`Map`	`Map<Object, Object>`

[8] Некоторые вычислительные функции настолько распространены, что заслуживают отдельного рассмотрения. Иногда их помещают в специальные категории. Это функции фильтрации (возвращающие булевское значение) и агрегатные функции (принимающие множество и возвращающие скалярное значение).

[9] Документация по API в формате Javadoc приведена на странице http://hadoop.apache.org/pig/javadoc/docs/api/.

Научиться писать UDF проще всего на примере какой-нибудь функции из репозитория PiggyBank. Часто при создании собственной функции полезно взять какую-нибудь готовую UDF, функционально похожую на то, что вам требуется, а затем модифицировать логику обработки. Мы изучим уже упоминавшуюся выше функцию `UPPER`. Метод `exec()` в ней выглядит следующим образом:

```
public class UPPER extends EvalFunc<String>
{
   public String exec(Tuple input) throws IOException {
      if (input == null || input.size() == 0)
         return null;

      try {
         String str = (String)input.get(0);
            return str.toUpperCase();
      } catch(Exception e){
         System.err.println("Failed to process input; error - " +
                           ➥ e.getMessage());
         return null;
      }
   }
}
```

Объект `input` имеет тип класса `Tuple`, в котором есть два метода для извлечения содержимого:

```
List<Object> getAll();
Object get(int fieldNum) throws ExecException;
```

Метод `getAll()` возвращает все поля кортежа в виде упорядоченного списка. Но в функции `UPPER` используется метод `get()`, возвращающий конкретное поле (в позиции с индексом 0). Если запрошенный индекс больше или равен количеству полей в кортеже, то возбуждается исключение `ExecException`. Полученное поле приводится к типу Java String. Обычно эта операция завершается нормально, но может возбудить исключение при попытке приведения несовместимых типов данных. Ниже мы увидим, как Pig гарантирует успешность приведения. Как бы то ни было, блок try/catch перехватит и обработает любое исключение. Если все хорошо, то метод `exec()` вернет строку String, в которой все буквы преобразованы в верхний регистр. Кроме того, большинство UDF должны реализовывать поведение по умолчанию, заключающееся в том, что на выходе получается `null`, если на вход был передан `null`.

Помимо реализации метода `exec()`, функция `UPPER` переопределяет еще два метода из класса `EvalFunc`. Первый называется `getArgToFuncMapping`:

```
@Override
public List<FuncSpec> getArgToFuncMapping() throws
FrontendException {
   List<FuncSpec> funcList = new ArrayList<FuncSpec>();
   funcList.add(new FuncSpec(this.getClass().getName(),
      ➥ new Schema(new Schema.FieldSchema(null, DataType.CHARARRAY))));
   return funcList;
}
```

Метод `getArgToFuncMapping()` возвращает список `List` объектов типа `FuncSpec`, представляющий схему всех полей в кортеже `input`. Pig обрабатывает пытается привести типы полей кортежа в соответствии с этой схемой до передачи методу `exec()`. Вместо полей, которые не удается привести к нужному типу, передается `null`.

Функцию `UPPER` интересует только тип первого поля, поэтому она включает в список только один объект `FuncSpec`, который говорит, что поле должно принадлежать типу `chararray`, представленному классом `DataType.CHARARRAY`. Создание `FuncSpec` выглядит довольно запутанно, что объясняется встроенной в Pig возможностью обрабатывать составные вложенные типы. К счастью, если типы, с которыми вы работаете, не особенно сложны, то код создания нужного вам объекта `FuncSpec`, скорее всего, уже есть в какой-нибудь функции из PiggyBank, так что его вы сможете им воспользоваться. Можно даже взять весь код метода `getArgToFuncMapping()` целиком, если схема кортежа совпадает с той, что подразумевается в существующей UDF.

Помимо входной схемы, вы должны сообщить Pig о схеме результата. Если результатом UDF является скаляр, то делать это необязательно, потому что в Pig используется механизм отражения Java для автоматического вывода схемы. Но если ваша UDF возвращает кортеж или множество, то этот механизм не сумеет полностью определить схему, и тогда вам придется описать ее явно, чтобы Pig мог распространить ее далее.

В случае `UPPER` выводится простая строка String, поэтому можно было бы обойтись и без определения выходной схемы. Но `UPPER` все же делает это, переопределяя метод `outputSchema()` и сообщая Pig, что собирается вернуть строку (`DataType.CHARARRAY`).

```
@Override
public Schema outputSchema(Schema input) {
   return new Schema(
      new Schema.FieldSchema(
          getSchemaName(this.getClass().getName().toLowerCase(), input),
          DataType.CHARARRAY
```

```
        )
    );
}
```

И на этот раз в том, как конструируется объект `Schema`, разберешься не сразу – все из-за обработки составных вложенных типов в Pig. Упомянем один частный случай – когда выходная схема возвращаемого UDF значения совпадает с входной схемой. Тогда можно просто вернуть копию входной схемы:

```
public Schema outputSchema(Schema input) {
   return new Schema(input);
}
```

Как всегда при конструировании `FuncSpec`, велики шансы найти в PiggyBank готовую UDF с нужной выходной схемой.

Некоторые типы UDF заслуживают специального упоминания. Функции фильтрации – это вычислительные функции, возвращающие значение типа Boolean, они используются в командах `FILTER` и `SPLIT`. Классы таких UDF должны расширять `FilterFunc`, а не `EvalFunc`. Агрегатными называются вычислительные функции, которые принимают множество и возвращают скаляр. Обычно они используются для вычисления таких агрегированных показателей, как `COUNT`, и иногда в Hadoop их удается оптимизировать за счет применения комбинатора. Мы не стали рассматривать UDF загрузки/сохранения, которые читают и записывают наборы данных. Это более сложная тема, с ней можно познакомиться в документации по Pig на странице http://hadoop.apache.org/pig/docs/r0.3.0/udf.html.

10.7. Работа со скриптами

Создание скриптов на языке Pig Latin сводится к собиранию вместе команд, успешно протестированных в Grunt. Однако же есть и ряд уникальных для скриптов особенностей: комментарии, подстановка параметров и режим многозапросного исполнения.

10.7.1. Комментарии

В предположении, что скрипт будет использоваться неоднократно, очевидно, имеет смысл прокомментировать его, чтобы другие люди (или вы сами) впоследствии могли понять, что к чему. В Pig Latin поддерживаются два вида комментариев: однострочные и многострочные. Однострочный комментарий начинается двумя

дефисами и продолжается до конца строки. Многострочный комментарий обрамляется маркерами `/*` и `*/`, как в Java. Вот, например, как выглядит скрипт на Pig Latin с комментариями:

```
/*
 * Myscript.pig
 * Еще одна строка комментария
 */
log = LOAD 'excite-small.log' AS (user, time, query);
lmt = LIMIT log 4; -- Показывать только 4 кортежа
DUMP lmt;
-- Конец программы
```

10.7.2. Подстановка параметров

Когда пишется повторно используемый скрипт, его обычно стараются параметризовать, так чтобы можно было настраивать работу при каждом запуске. Например, скрипт может принимать пути к входному и выходному файлу. Pig поддерживает подстановку параметров, чтобы такого рода информацию можно было задавать на этапе выполнения. Внутри скрипта параметры обозначаются переменными с префиксом `$`. Так, следующий скрипт выводит заданное число кортежей из указанного файла журнала:

```
log = LOAD '$input' AS (user, time, query);
lmt = LIMIT log $size;
DUMP lmt;
```

Параметрами скрипта являются `$input` и `$size`. При запуске скрипта командой `pig` параметры задаются в аргументе вида `-param name=value`.

```
pig -param input=excite-small.log -param size=4 Myscript.pig
```

Отметим, что в аргументах указывать префикс `$` не нужно. Если значение параметра состоит из нескольких слов, то его следует заключить в одиночные или двойные кавычки. Бывает полезно использовать команды Unix для генерации значений параметров, особенно дат. Для этого применяется имеющийся в Unix механизм подстановки результатов выполнения команды путем заключения ее в обратные апострофы (`), например:

```
pig -param input=web-`date +%y-%m-%d`.log -param size=4 Myscript.pig
```

В результате имя входного файла для скрипта `Myscript.pig` оказывается зависящим от даты запуска скрипта: если скрипт запущен

29 июля 2009 года, то входной файл будет называться `web-09-07-29.log`.

Если требуется задать много параметров, то удобнее поместить их в файл и сообщить Pig имя этого файла. Например, можно создать файл Myparams.txt с таким содержимым:

```
# Комментарии в файле параметров должны начинаться со знака решетки
input=excite-small.log
size=4
```

Имя файла параметров передается команде `pig` в аргументе `-param_file filename`.

```
pig -param_file Myparams.txt Myscript.pig
```

Можно задать несколько файлов параметров, можно также задавать одни параметры в файлах, а другие прямо в командной строке с помощью аргумента `-param`. Если некоторый параметр определен несколько раз, то во внимание принимается только последнее значение. Если сомневаетесь, с какими параметрами будет в конечном итоге запущен скрипт, можете вызвать команду `pig` с флагом `-debug`. Он означает, что Pig должен выполнить скрипт и создать файл `original_script_name.substituted`, в котором вместо всех параметров подставлены их значения. Если задать еще флаг `-dryrun`, то этот файл будет создан, но выполнять скрипт Pig не станет.

Команды `exec` и `run` позволяют запускать скрипты на языке Pig Latin из оболочки Grunt и поддерживают подстановку параметров с помощью всё тех же аргументов `-param` и `-param_file`, например:

```
grunt> exec -param input=excite-small.log -param size=4 Myscript.pig
```

Однако при запуске с помощью `exec` или `run` подстановка результатов выполнения команд Unix не производится и флаги `debug` и `dryrun` не поддерживаются.

10.7.3. Режим многозапросного исполнения

При работе внутри оболочки Grunt команды `DUMP` и `STORE` обрабатывают все предыдущие команды, необходимые для получения результата. С другой стороны, скрипт оптимизируется и исполняется как единое целое. И не было бы никакой разницы, если бы скрипт содержал всего одну команду `DUMP` или `STORE`, причем в самом

конце. Если же в скрипте есть несколько команд DUMP или STORE, то включается *режим многозапросного исполнения* (multiquery execution), в котором эффективность повышается за счет устранения избыточных вычислений. Предположим, к примеру, что имеется такой скрипт, сохраняющий промежуточные данные:

```
a = LOAD ...
b = какое-то преобразование a
STORE b ...
c = еще какое-то преобразование b
STORE c ...
```

Если ввести эти команды в оболочке Grunt, где нет режима многозапросного исполнения, то в точке STORE b будет создана цепочка задач для вычисления b. А в точке STORE c Grunt создаст новую цепочку задач для вычисления c, но теперь будут еще раз вычислены a и b! Можно избежать повторного вычисления вручную, вставив после STORE b команду b = LOAD ..., чтобы заставить оболочку использовать сохраненное значение b при вычислении c. Такой способ будет работать в предположении, что сохраненное значение b не модифицировалось, поскольку Grunt сама по себе узнать это не в состоянии.

С другой стороны, если поместить все эти команды в скрипт и запустить его, то режим многозапросного исполнения может произвести оптимизацию за счет интеллектуальной обработки промежуточных данных. Pig компилирует все команды сразу и может определить, где имеются зависимости и избыточность. По умолчанию этот режим включен и обычно не оказывает влияния на результаты вычислений. Однако режим многозапросного исполнения может давать сбои, если между данными имеются зависимости, о которых Pig не знает. Редко, но такое случается, например, при использовании UDF. Рассмотрим следующий скрипт:

```
STORE a INTO 'out1';
b = LOAD ...
c = FOREACH b GENERATE MYUDF($0,'out1');
STORE c INTO 'out2';
```

Если пользовательская функция MYUDF обращается к a через файл out1, то Pig никогда об этом не узнает. Не видя зависимости, компилятор Pig может ошибочно решить, что допустимо вычислять b и c до вычисления a. Чтобы отключить режим многозапросного исполнения, вызывайте команду pig с флагом -M или -no_multiquery.

10.8. Pig в действии: отыскание похожих патентов

С учетом дополнительных возможностей, которые дает Pig, мы можем взяться за более амбициозные задачи обработки данных. Например, интересно было бы найти похожие патенты, основываясь на данных о цитировании. Патенты, которые часто цитируются вместе, должны быть в каком-то смысле похожи (или, по крайней мере, взаимосвязаны). Это приложение по сути своей напоминает механизм коллаборативной фильтрации, применяемый на сайте Amazon.com («Посетители, купившие *это*, купили также и *то*»), и отыскание похожих документов (путем поиска документов с похожим набором слов). Будем считать, что нам нужно найти патенты, которые цитировались вместе более `N` раз, где `N` – заданная нами величина[10].

Если приложение должно производить вычисления над парами элементов (например, вычислять количество совместных ссылок для каждой пары патентов), то сразу приходит на ум реализация с двумя вложенными циклами, в которых перебираются все попарные комбинации и для каждой пары производится некое вычисление. Но хотя Hadoop и допускает масштабирование путем простого добавления оборудования, не стоит забывать о фундаментальных понятиях теории вычислительной сложности. Квадратичная сложность легко поставит линейную масштабируемость на колени. Даже сравнительно небольшой набор из 3 миллионов патентов порождает 9 триллионов пар. Нет, нам определенно нужен более разумный алгоритм.

Главное, на что нужно обратить внимание, – это *разреженность* результирующих данных. У большинства пар схожесть нулевая, поскольку, как правило, патенты не цитируются вместе. Вычисление схожести станет куда более обозримым, если нам удастся переформулировать алгоритм так, чтобы он работал только с парами патентов, которые заведомо цитировались вместе. И при нашей организации данных такой подход оказывается вполне естественным. Реализация подразумевает выполнение следующих действий для каждого патента:

[10] Можно вместо этого использовать более сложные оценочные функции, например, нормализацию для часто встречающихся значений или вычисление оценки схожести, а не просто отсечение. Простой критерий отсечения выбран нами для того, чтобы было легче реализовать и проиллюстрировать принципы определения подобия.

1. Получить список патентов, на которые в нем есть ссылки.
2. Сгенерировать попарные комбинации элементов из полученного списка и записать каждую пару.
3. Подсчитать, сколько раз встречается каждая пара.

Если в каждом патенте есть ссылки на одно и то же число патентов, например 10, то для каждого патента будет сгенерировано 45 пар (45 – число попарных комбинаций 10 элементов, легко доказать, что оно равно $10 \times 9 / 2$). При 3 миллионах патентов мы получим 135 миллионов пар, что на несколько порядков меньше, чем при прямом переборе.

Чем больше размер набора данных о патентах, тем нагляднее выигрыш. Но хотя мы и нашли подходящий алгоритм, реализовать его в духе MapReduce довольно утомительно. Потребуется сцеплять несколько задач, и для каждой задачи нужно будет написать отдельный класс. А на языке Pig Latin программа из трех описанных выше шагов занимает всего десяток строчек (листинг 10.1), причем возможна оптимизация, которая еще уменьшит число строк и повысит эффективность.

Листинг 10.1.
Написанный на Pig Latin скрипт для нахождения патентов, которые часто цитируются вместе

```
          cite = LOAD 'input/cite75_99.txt' USING PigStorage(',')
                 ➥ AS (citing:int, cited:int);
     cite_grpd = GROUP cite BY citing;
 cite_grpd_dbl = FOREACH cite_grpd GENERATE group, cite.cited AS cited1,
                 ➥ cite.cited AS cited2;
        cocite = FOREACH cite_grpd_dbl
                 ➥ GENERATE FLATTEN(cited1), FLATTEN(cited2);
  cocite_fltrd = FILTER cocite BY cited1 != cited2;
   cocite_grpd = GROUP cocite_fltrd BY *;
    cocite_cnt = FOREACH cocite_grpd
                 ➥ GENERATE group, COUNT(cocite_fltrd) as cnt;
   cocite_flat = FOREACH cocite_cnt GENERATE FLATTEN(group), cnt;
cocite_cnt_grpd = GROUP cocite_flat BY cited1;
    cocite_bag = FOREACH cocite_cnt_grpd
                 ➥ GENERATE group, cocite_flat.(cited2, cnt);
cocite_final = FOREACH cocite_cnt_grpd {
    similar = FILTER cocite_flat BY cnt > 5;
    GENERATE group, similar;
}
STORE cocite_final INTO 'output';
```

Скрипты на Pig Latin да и, наверное, любые программы обработки сложных данных, читать тяжело. К счастью, мы можем применить

команду Grunt `ILLUSTRATE` к `cocite_bag`, получить имитацию выполнения этих предложений и посмотреть, что порождает каждая операция (мы изменили формат вывода, чтобы не выходить за пределы печатной страницы).

```
-------------------------------------------------
| cite | citing: bytearray | cited: bytearray     |
-------------------------------------------------
|      | 3858554           | 3601095              |
|      | 3858554           | 3685034              |
|      | 3859004           | 1730866              |
|      | 3859004           | 3022581              |
|      | 3859572           | 3206651              |
-------------------------------------------------
---------------------------------------
| cite | citing: int | cited: int     |
---------------------------------------
|      | 3858554     | 3601095        |
|      | 3858554     | 3685034        |
|      | 3859004     | 1730866        |
|      | 3859004     | 3022581        |
|      | 3859572     | 3206651        |
---------------------------------------
-------------------------------------------------------------------------
| cite_grpd | group: int | cite: bag({citing: int,cited: int})      |
-------------------------------------------------------------------------
|           | 3858554    | {(3858554, 3601095), (3858554, 85034)}   |
|           | 3859004    | {(3859004, 1730866), (3859004, 3022581)} |
|           | 3859572    | {(3859572, 3206651)}                     |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| cite_grpd_dbl | group:  | cited1:               | cited2:            |
|               | int     | bag({cited: int})     | bag({cited: int})  |
-------------------------------------------------------------------------
|               | 3858554 | {(3601095), (3685034)} | {(3601095),(3685034)} |
|               | 3859004 | {(1730866), (3022581)} | {(1730866),(3022581)} |
|               | 3859572 | {(3206651)}           | {(3206651)}          |
-------------------------------------------------------------------------
```

Отношение `cite_grpd` содержит по одному множеству для каждого патента, и в этом множестве находятся цитированные патенты. Из этого отношения мы видим (по результатам выборочного прогона), что на патенты 3601095 и 3685034 есть ссылки из патента 3858554. Группировка совместно цитируемых патентов была произведена операцией `GROUP` в ходе создания `cite_grpd`. Отношение `cite_grpd_dbl` всего лишь удаляет избыточный «цитирующий» патент и создает столбец-дубликат. Значения в столбцах `cited1` и `cited2` одинаковы. Дублирование позволяет с

помощью операции перекрестного произведения создать все попарные комбинации.

```
-----------------------------------------------------
| cocite | cited1::cited: int | cited2::cited: int     |
-----------------------------------------------------
|        | 3601095            | 3601095                |
|        | 3601095            | 3685034                |
|        | 3685034            | 3601095                |
|        | 3685034            | 3685034                |
|        | 1730866            | 1730866                |
|        | 1730866            | 3022581                |
|        | 3022581            | 1730866                |
|        | 3022581            | 3022581                |
|        | 3206651            | 3206651                |
-----------------------------------------------------
```

Операция перекрестного произведения «разглаживает» каждую строку `cite_grpd_dbl` и создает отношение `cocite`[11]. Это перечисление всех пар патентов, на которые есть совместные ссылки, и является главным местом в нашем алгоритме. Мы знаем, что `cocite` – объемное отношение, пусть даже наша схема более эффективна, чем прямой перебор. Но есть три способа еще уменьшить `cocite`. Мы обсудим все три, но реализуем только один.

Для начала заметим, что на каждый цитированный патент имеется ссылка из него самого. Но в нашем приложении бессмысленно считать, что патент похож на себя, поэтому все такие пары можно игнорировать. Отметим, что если оставлять такие «тождественные» пары при вычислениях, то счетчик совместных ссылок для них будет в точности равен счетчику ссылок. Эти числа могли бы быть полезны, если бы нас интересовала процентная доля случаев совместного цитирования патентов. Но мы вычисляем не проценты, так что это соображение не играет роли.

Поскольку совместное цитирование симметрично, то каждая пара встречается дважды, с перестановкой элементов. Например, мы видим обе пары (3601095,3685034) и (3685034,3601095). Учитывая, что приложение ищет пары патентов, совместно цитируемых более *N* раз, мы можем оставить только одну из двух пар и тем самым уменьшить размер отношения `cocite` вдвое. Для определенности будем применять следующее правило: оставлять пару, в которой первое поле меньше второго. Впрочем, в некоторых приложениях наличие

[11] Отметим, что `cocite` можно вычислить по `cite_grpd` непосредственно, воспользовавшись более сложной формой `FOREACH`, и, научившись читать скрипты на Pig Latin, вы, наверное, так и будете поступать.

избыточных пар может оказаться полезным для последующих действий. Например, можно найти все патенты, цитируемые вместе с X, взяв пары, в которых X находится на первом месте.

Наконец, можно применить эвристические оценки, чтобы удалить пары, которые нам кажутся не существенными. При этом мы жертвуем точностью результата ради эффективности. Существуют ли такие эвристики и насколько они полезны, зависит от семантики приложения и распределения данных. В нашем случае патент, в котором есть много ссылок на другие патенты, порождает квадратичное количество строк в `cocite`. Если мы считаем, что такие «изобильные ссылками» патенты мало что дают для понимания схожести патентов, то их удаление мало повлияет на конечный результат, зато позволит существенно сократить объем данных. Выигрыш от этой эвристики будет гораздо больше, если мы исследуем обратное цитирование или текстовые документы, где распределение частот крайне неравномерно, а квадратичная зависимость для немногих часто встречающихся элементов может привести к резкому увеличению объема обрабатываемых данных. На самом деле, в таких ситуациях приближенные эвристические соображения почти всегда необходимы.

Важно отметить, что при обсуждении процедуры мы сосредоточились на верхнем уровне обработки, а низкоуровневые детали MapReduce опустили.

```
-------------------------------------------------------------
| cocite_fltrd | cited1::cited: int | cited2::cited: int     |
-------------------------------------------------------------
|              | 3601095            | 3685034                |
|              | 3685034            | 3601095                |
|              | 1730866            | 3022581                |
|              | 3022581            | 1730866                |
-------------------------------------------------------------
```

Мы решили ограничиться фильтрацией «тождественных» пар патентов:

```
-------------------------------------------------------------------
| cocite_grpd | group:                      | cocite_fltrd:    |
|             | tuple({cited1::cited: int,  | bag({cited1::cited: int, |
|             |        cited2::cited: int}) |      cited2::cited: int})|
-------------------------------------------------------------------
|             | (1730866, 3022581)          | {(1730866, 3022581)}     |
|             | (3022581, 1730866)          | {(3022581, 1730866)}     |
|             | (3601095, 3685034)          | {(3601095, 3685034)}     |
|             | (3685034, 3601095)          | {(3685034, 3601095)}     |
-------------------------------------------------------------------
```

```
-----------------------------------------------------------------
| cocite_cnt | group:                                           | cnt: |
|            | tuple({cited1::cited: int,cited2::cited: int})   | long |
-----------------------------------------------------------------
|            | (1730866, 3022581)                               | 1    |
|            | (3022581, 1730866)                               | 1    |
|            | (3601095, 3685034)                               | 1    |
|            | (3685034, 3601095)                               | 1    |
-----------------------------------------------------------------
-----------------------------------------------------------------
| cocite_flat | group::cited1::cited: | group::cited2::cited: | cnt: |
|             | int                   | int                   | long |
-----------------------------------------------------------------
|             | 1730866               | 3022581               | 1    |
|             | 3022581               | 1730866               | 1    |
|             | 3601095               | 3685034               | 1    |
|             | 3685034               | 3601095               | 1    |
-----------------------------------------------------------------
```

Мы сгруппировали пары, содержащие совместные ссылки на патенты, подсчитали их и «разгладили» отношение. К сожалению, команда `ILLUSTRATE` порождает выборку, в которой все счетчики совместных ссылок равны 1. Однако мы все же видим, что делается именно то, что мы хотели. Если не отступать от исходной постановки задачи, согласно которой ищутся пары патентов, которые совместно цитировались более *N* раз, то останется только применить фильтр к отношению `cocite_flat`. Но мы хотим показать, как можно продолжить группировку кортежей, что может пригодиться для других типов фильтрации. Например, что если требуется для каждого патента найти *K* самых цитируемых вместе с ним патентов? Посмотрим на продолжение распечатки:

```
-----------------------------------------------------------------
| cocite_cnt_grpd | group: int | cocite_flat: bag({group::cited1::cited:  |
|                 |            | int,group::cited2::cited: int,cnt: long})|
-----------------------------------------------------------------
|                 | 1730866    | {(1730866, 3022581, 1)}                  |
|                 | 3022581    | {(3022581, 1730866, 1)}                  |
|                 | 3601095    | {(3601095, 3685034, 1)}                  |
|                 | 3685034    | {(3685034, 3601095, 1)}                  |
-----------------------------------------------------------------
-----------------------------------------------------------------
| cocite_bag | group: int | cocite_flat:                               |
|            |            | bag({group::cited2::cited: int,cnt: long})  |
-----------------------------------------------------------------
|            | 1730866    | {(3022581, 1)}                             |
|            | 3022581    | {(1730866, 1)}                             |
|            | 3601095    | {(3685034, 1)}                             |
|            | 3685034    | {(3601095, 1)}                             |
-----------------------------------------------------------------
```

Если бы нужно было найти для каждого патента *K* самых цитируемых вместе с ним патентов, то следовало бы воспользоваться оператором `FOREACH` для обработки каждого кортежа отношения `cocite_bag` и написать свою UDF-функцию, которая принимает множество (`cocite_flat`) и возвращает не более *K* кортежей (с наибольшим количеством совместных ссылок). Этот последний шаг оставляем в качестве самостоятельного упражнения. И посмотрим, как выглядит вложенный оператор `FOREACH`, который отфильтровывает из множеств кортежи, для которых счетчик не превышает 5.

```
cocite_final = FOREACH cocite_cnt_grpd {
    similar = FILTER cocite_flat BY cnt > 5;
    GENERATE group, similar;
}
```

Как видите, Pig позволил здорово упростить реализацию этого приложения для обработки данных. Известно, что «поиск похожих» полезен во многих разных приложениях, но реализовать его нелегко. Однако с помощью Pig и Hadoop для решения задачи хватило бы остатка рабочего дня после обеда. Более того, простота разработки дает возможность быстро создавать прототипы других программ. В качестве упражнения попробуйте найти не патенты, которые часто цитируются совместно, а патенты, на которые имеются схожие ссылки.

10.9. Резюме

Pig – высокоуровневый программный слой для обработки данных, надстроенный над Hadoop. Язык Pig Latin предлагает программистам интуитивно более понятный способ описания потоков данных. Он поддерживает схемы для обработки структурированных данных, но при этом обладает достаточной гибкостью для обработки неструктурированного текста или полуструктурированных XML-документов. Язык допускает расширение за счет функций, определяемых пользователем. Pig существенно упрощает соединение данных и сцепление задач – именно те два аспекта программирования для MapReduce, которые вызывают наибольшие сложности у программистов. Для демонстрации полезности мы разработали довольно сложную программу нахождения совместно цитируемых патентов, которая на Pig Latin записывается в несколько строчек.

ГЛАВА 11.
Hive и другие

В этой главе:

- Что такое Hive.
- Настройка Hive.
- Использование Hive для организации хранилищ данных.
- Другие программные пакеты, связанные с Hadoop.

Каким бы мощным инструментом ни был Hadoop, он не может устроить всех и каждого. От Hadoop отпочковалось много специализированных проектов. Самые значительные и хорошо поддерживаемые получили статус официальных подпроектов Apache Hadoop[1]. К их числу относятся:

- *Pig* – высокоуровневый язык описания потоков данных;
- *Hive* –SQL-подобная инфраструктура для организации хранилищ данных;
- *HBase* – распределенная СУБД с хранением по столбцам, устроенная по образцу Google Bigtable;
- *ZooKeeper* – надежная система координации для управления состоянием, общим для нескольких распределенных приложений;
- *Chukwa* – система сбора данных для управления большими распределенными системами.

В главе 10 мы подробно рассмотрели Pig, а теперь поговорим о Hive. В разделе 11.2 мы также кратко опишем другие связанные

[1] То, что мы называем в этой книге «Hadoop» (то есть HDFS и MapReduce), строго говоря, называется подпроектом «Hadoop Core» проекта Apache Hadoop, хотя в разговорной речи все говорят просто «Hadoop».

с Hadoop проекты. Часть из них не имеет отношения к Apache (например, Cascading, CloudBase). Некоторые пока находятся в стадии становления (Hama, Mahout). О том, как эти инструменты применяются на практике, мы расскажем в главе 12.

11.1. Hive

Hive[2] – это пакет для организации хранилищ данных, построенный на базе Hadoop. Он был создан в компании Facebook, которой необходимо было обрабатывать гигантские объемы пользовательских данных и журналов. Теперь это подпроект Hadoop, над которым работает много людей. Ориентирован он на аналитиков, которые уверенно владеют SQL и нуждаются в средстве для выполнения произвольных запросов, агрегирования и анализа данных, объем которых таков, что требуется Hadoop[3]. Для взаимодействия с Hive применяется SQL-подобный язык запросов, называемый HiveQL. Например, вот как выглядит запрос для получения всех активных пользователей в таблице `user`:

```
INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;
```

Дизайн Hive отражает его предназначение – служить в качестве системы для управления и предъявления запросов к *структурированным* данным. Будучи ориентирован на структурированные данные, Hive может позволить себе некоторые виды оптимизации и удобства для пользователей, отсутствующие в более общем каркасе MapReduce. Построенный по образцу SQL язык изолирует пользователя от сложностей программирования для MapReduce. В нем используются знакомые понятия из области баз данных: таблицы, строки, столбцы, схема, что облегчает изучение. Кроме того, если Hadoop естественно работает с плоскими файлами, то Hive способен использовать структуры каталогов для «секционирования» данных, это повышает производительность некоторых запросов. Для поддержки дополнительных возможностей, в Hive включен новый важный компонент – метахранилище, в котором хранится информация о схеме. Обычно метахранилище размещается в реляционной базе данных.

[2] http://hadoop.apache.org/hive/

[3] Отметим, что поскольку Hive построен на базе Hadoop, он также ориентирован на пакетную обработку с малой задержкой. Поэтому его нельзя рассматривать как прямую замену традиционным SQL-хранилищам данных, в частности, тем, что предлагает Oracle.

Для взаимодействия с Hive есть несколько способов, в том числе веб-интерфейс (GUI) и интерфейс Java Database Connectivity (JDBC). Но чаще для этой цели используется командный интерфейс (command line interface – CLI), который мы и будем рассматривать. На рис. 11.1 изображена общая архитектура Hive.

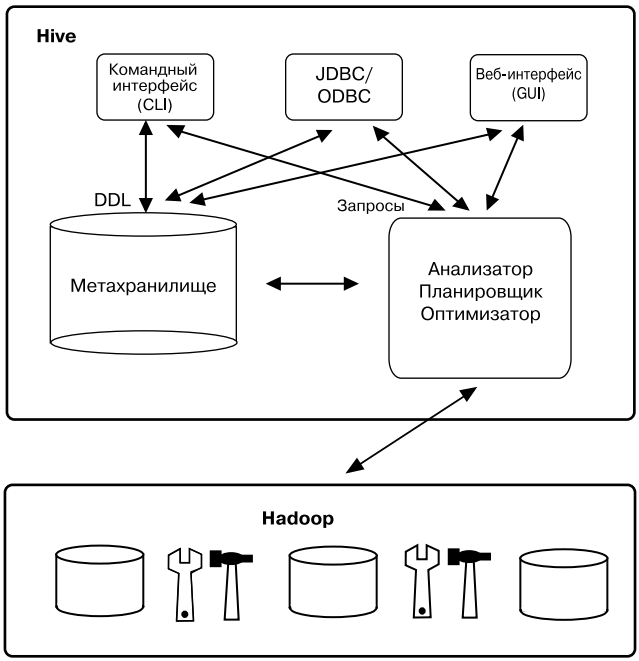


Рис. 11.1. Архитектура Hive.
Запросы разбираются и исполняются с помощью Hadoop.
Важным компонентом является метахранилище,
позволяющее решить, как следует исполнять запрос.

11.1.1. Установка и настройка Hive

Для работы Hive необходимы Java 1.6 и Hadoop версии 0.17 или выше. Последнюю версию Hive можно найти по адресу http://hadoop.apache.org/hive/releases.html. Скачайте и распакуйте tgz-архив в какой-нибудь каталог, на который будет указывать переменная окружения `HIVE_HOME`. Предполагается, что Hadoop уже установлен и запущен. Кроме того, необходимо создать в системе HDFS два каталога специально для Hive.

```
bin/hadoop fs -mkdir /tmp
bin/hadoop fs -mkdir /user/hive/warehouse
bin/hadoop fs -chmod g+w /tmp
bin/hadoop fs -chmod g+w /user/hive/warehouse
```

Если вы разрешите Hive взять на себя все управление вашими данными, то Hive будет хранить их в каталоге /user/hive/warehouse. Hive может автоматически сжимать данные и создавать специальные структуры каталогов (аналог секций) для ускорения выполнения запросов.

Поручить управление данными Hive имеет смысл, если вы планируете использовать Hive для предъявления к ним запросов. Но если данные уже хранятся в других каталогах HDFS, и вы хотите, чтобы они там и оставались, то и тогда Hive сможет с ними работать. Но в таком случае не ждите никакой оптимизации хранилища для обработки запросов. Неопытные пользователи не всегда понимают это различие и думают, что Hive требует хранить данные в каком-то особом формате. Это безусловно не так.

Hive хранит метаданные в стандартной реляционной базе данных. В комплект поставки Hive входит облегченная встраиваемая СУБД с открытым исходным кодом на основе SQL под названием Derby[4], которая устанавливается на клиентской машине вместе с Hive. Если вы единственный пользователь Hive, то такой конфигурации по умолчанию вполне достаточно. Но, скорее всего, после начальных экспериментов и оценки пригодности, вы захотите развернуть Hive в многопользовательской среде, и вряд ли вам понравится, если у каждого пользователя будет собственная версия метаданных. Потребуется некое централизованное хранилище. Обычно для этой цели используют многопользовательскую СУБД типа MySQL, но подойдет любая совместимая с JDBC СУБД. Вам понадобится сервер баз данных и отдельная база, играющая роль метахранилища Hive. Эту базу принято называть `metastore_db`. Создав базу, сконфигурируйте все инсталляции Hive, так чтобы они использовали ее в качестве метахранилища. Для этого нужно модифицировать файлы hive-site.xml и jpox.properties, находящиеся в каталоге `$HIVE_HOME/conf`. На этапе установки файл hive-site.xml не создается, вам придется создать его вручную. Заданные в этом файле свойства переопределяют свойства в файле hive-default.xml точно так же, как файл hadoop-site.xml переопределяет hadoop-default.xml. В файле hive-site.xml следует переопределить три свойства, после чего он будет выглядеть примерно так:

[4] http://db.apache.org/derby/.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>
  <name>hive.metastore.local</name>
  <value>false</value>
</property>

<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:mysql://<hostname>/metastore_db</value>
</property>

<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>com.mysql.jdbc.Driver</value>
</property>

</configuration>
```

Что означает каждое свойство, поясняется в табл. 11.1. Свойства `javax.jdo.option.ConnectionURL` и `javax.jdo.option.ConnectionDriverName` следует еще раз задать в файле jpox.properties. Там же задаются имя пользователя и пароль для подключения к базе данных. Итого в файле jpox.properties должны быть такие четыре строки:

```
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL=jdbc:mysql://<hostname>/metastore_db
javax.jdo.option.ConnectionUserName=<username>
javax.jdo.option.ConnectionPassword=<password>
```

Таблица 11.1. Конфигурационные параметры для использования базы данных MySQL в качестве хранилища метаданных в многопользовательском режиме

Свойство	Описание
`hive.metastore.local`	Надо ли создавать и использовать локальный сервер метахранилища на клиентской машине. Присвойте значение false, если хотите работать с удаленным сервером метахранилища.
`javax.jdo.option.ConnectionURL`	URL JDBC-соединения, определяющий, как подключиться к базе данных, содержащей метахранилище[1]. Например, `jdbc:mysql://<hostname>/metastore_db`.

[1] Полностью формат URL для подключения к MySQL через JDBC описан на странице http://dev.mysql.com/ doc/refman/5.0/en/connector-j-reference-configuration-properties.html.

Таблица 11.1. (окончание)

Свойство	Описание
`javax.jdo.option.ConnectionDriverName`	Имя класса драйвера JDBC, например `com.mysql.jdbc.Driver`.
`javax.jdo.option.ConnectionUserName`	Имя пользователя для подключения к базе данных.
`javax.jdo.option.ConnectionPassword`	Пароль для подключения к базе данных.

После настройки базы данных (или если вы оставили подразумеваемый по умолчанию однопользовательский режим на время оценки пригодности Hive) можно зайти в командный интерфейс. Находясь в каталоге `$HIVE_HOME`, введите команду `bin/hive`. Появится приглашение Hive, означающее, что можно вводить команды.

```
bin/hive
Hive history file=/tmp/root/hive_job_log_root_200908240830_797162695.txt
hive>
```

11.1.2. Примеры запросов

Перед тем как приступать к формальному описанию языка HiveQL, полезно выполнить несколько команд. Это поможет вам составить представление о языке и перейти к самостоятельным экспериментам.

Предположим, что на локальной машине имеется файл с данными о цитировании патентов cite75_99.txt. Напомним, что поля данных в нем разделены запятыми. Сначала определим в Hive таблицу для хранения данных:

```
hive> CREATE TABLE cite (citing INT, cited INT)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY ','
    > STORED AS TEXTFILE;
OK
Time taken: 0.246 seconds
```

Каждое предложение в языке HiveQL должно завершаться точкой с запятой. Предложение может занимать несколько строк при условии, что точка с запятой ставится в самом конце, – мы так и поступили.

Основное действие этой команды из четырех строчек описывается в первой строке. Здесь определяется таблица *cite* с двумя столбцами. Первый столбец называется *citing* и имеет тип `INT`, второй – *cited* и тоже имеет тип `INT`. В остальных строчках мы сообщаем Hive, как

хранятся данные (в виде текстового файла) и как их следует разбирать (разбивая по точке с запятой).

Посмотреть, какие в данный момент определены таблицы, позволяет команда `SHOW TABLES`:

```
hive> SHOW TABLES;
OK
cite
Time taken: 0.053 seconds
```

Между сообщениями «OK» и «Time taken» мы видим таблицу cite. Чтобы посмотреть ее схему, выполните команду `DESCRIBE`:

```
hive> DESCRIBE cite;
OK
citing int
cited int
Time taken: 0.13 seconds
```

Как и следовало ожидать, таблица содержит те самые два столбца, что мы определили ранее. Средства определения и управления таблицами в HiveQL очень похожи на применяемые в стандартных реляционных базах данных. Загрузим в эту таблицу данные о патентах.

```
hive> LOAD DATA LOCAL INPATH 'cite75_99.txt'
    > OVERWRITE INTO TABLE cite;
Copying data from file:/root/cite75_99.txt
Loading data to table cite
OK
Time taken: 9.51 seconds
```

Здесь мы просим Hive загрузить данные из файла cite75_99.txt, находящегося в локальной файловой системе, в нашу таблицу *cite*. В результате этот файл будет скопирован в некий каталог HDFS, управляемый Hive. (Если вы не изменяли конфигурацию, то этот каталог будет расположен внутри /user/hive/warehouse.)

Hive не станет помещать в таблицу данные, не соответствующие схеме, а подставит вместо них значение null. Чтобы просмотреть данные в таблице cite, воспользуемся простой командой `SELECT`:

```
hive> SELECT * FROM cite LIMIT 10;
OK
NULL NULL
3858241 956203
3858241 1324234
3858241 3398406
3858241 3557384
3858241 3634889
3858242 1515701
```

```
3858242 3319261
3858242 3668705
3858242 3707004
Time taken: 0.17 seconds
```

В нашей схеме определены два столбца, содержащие целые числа. Как видим, первая строка содержит null в обоих столбцах, то есть запись оказалась противоречащей схеме. Объясняется это тем, что в первой строке файла cite75_99.txt находятся имена столбцов, а не номера патентов. Ничего страшного в этом нет.

Теперь, когда Hive прочитал данные и управляет ими, мы можем предъявлять к данным произвольные запросы. Для начала подсчитаем количество строк в таблице. В SQL это делается с помощью команды `SELECT COUNT(*)`. В HiveQL синтаксис немного отличается:

```
hive> SELECT COUNT(1) FROM cite;
Total MapReduce jobs = 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_200908250716_0001, Tracking URL = http://ip-10-244-199-
143.ec2.internal:50030/jobdetails.jsp?jobid=job_200908250716_0001
Kill Command = /usr/lib/hadoop/bin/hadoop job -Dmapred.job.tracker=ip-10-
244-199-143.ec2.internal:9001 -kill job_200908250716_0001
 map = 0%, reduce =0%
 map = 12%, reduce =0%
 map = 25%, reduce =0%
 map = 30%, reduce =0%
 map = 34%, reduce =0%
 map = 43%, reduce =0%
 map = 53%, reduce =0%
 map = 62%, reduce =0%
 map = 71%, reduce =0%
 map = 75%, reduce =0%
 map = 79%, reduce =0%
 map = 88%, reduce =0%
 map = 97%, reduce =0%
 map = 99%, reduce =0%
 map = 100%, reduce =0%
 map = 100%, reduce =67%
 map = 100%, reduce =100%
Ended Job = job_200908250716_0001
OK
16522439
Time taken: 85.153 seconds
```

Из напечатанных сообщений следует, что в результате выполнения запроса была создана задача MapReduce. Элегантность Hive заключается в том, что пользователю нет нужды знать что-либо о MapReduce. С его точки зрения, производится обычный запрос к базе данных на языке, похожем на SQL.

Результат запроса выведен прямо на экран. В большинстве случае результат нужно сохранять на диске, обычно в какой-то другой таблице Hive. В следующем запросе мы найдем частоту цитирования каждого патента. Но предварительно создадим таблицу для хранения результата:

```
hive> CREATE TABLE cite_count (cited INT, count INT);
OK
Time taken: 0.027 seconds
```

Мы можем выполнить запрос для нахождения частоты цитирования. При этом используются функция `COUNT` и фраза `GROUP BY` — практически так же, как в SQL. Существует дополнительная фраза `INSERT OVERWRITE TABLE`, которая говорит Hive о том, что результат следует записать в таблицу:

```
hive> INSERT OVERWRITE TABLE cite_count
    > SELECT cited, COUNT(citing)
    > FROM cite
    > GROUP BY cited;
...
 map = 100%, reduce =89%
 map = 100%, reduce =90%
 map = 100%, reduce =100%
Ended Job = job_200908250716_0002
Loading data to table cite_count
3258984 Rows loaded to cite_count
OK
Time taken: 103.331 seconds
```

По завершении запроса печатается полезная информация о том, что в таблицу частот цитирования загружено 3 258 984 строк. Для просмотра этой таблицы можно выполнить еще одну команду HiveQL:

```
hive> SELECT * FROM cite_count WHERE count > 10 LIMIT 10;
Total MapReduce jobs = 1
Number of reduce tasks is set to 0 as there's no reduce operator
...
 map = 80%, reduce =0%
 map = 100%, reduce =100%
Ended Job = job_200908250716_0003
```

```
OK
163404 13
164184 16
217584 13
246144 14
288134 11
347644 11
366494 11
443764 11
459844 13
490484 13
```

Интересно отметить, что Hive оказался настолько «умным», что распознал отсутствие оператора редукции и потому установил количество заданий редукции в 0 (Number of reduce tasks is set to 0 as there's no reduce operator). Закончив работать с таблицей, можете удалить ее командой `DROP TABLE`:

```
hive> DROP TABLE cite_count;
OK
Time taken: 0.024 seconds
```

Но будьте осторожны, это команда не просит подтвердить, что вы действительно хотите удалить таблицу. А восстановить случайно удаленную таблицу будет нелегко.

Наконец, для завершения сеанса работы с Hive предназначена команда `exit`.

11.1.3. Детали языка HiveQL

Мы видели Hive в действии и теперь готовы познакомиться с формальным описанием различных аспектов языка HiveQL.

Модель данных

Как вы уже поняли, в основе модели данных Hive лежат таблицы. Физически таблицы хранятся в подкаталогах каталога /user/hive/warehouse. Например, созданная нами таблица *cite* представлена файлами в каталоге /user/hive/warehouse/cite. Выходная таблица *cite_count* будет находиться в каталоге /user/hive/warehouse/cite_count. В простейшем случае иерархия каталогов плоская, то есть все данные таблицы хранятся в нескольких файлах в одном каталоге.

В реляционных базах данных для ускорения запросов применяются индексы, построенные по столбцам таблицы. В Hive вместо этого используется идея разбивающих столбцов. Так называются столбцы, значения в которых разбивают таблицу на несколько сек-

ций. Например, столбец *state* разбил бы таблицу на 50 секций, по числу штатов[5]. Для данных из журналов часто в качестве разбивающего берут столбец *date*; тогда данные за каждый день будут находиться в отдельной секции. Hive рассматривает разбивающие столбцы иначе, чем обычные, и запросы, в которых участвуют такие столбцы, выполняются гораздо эффективнее. Причина в том, что Hive физически хранит разные секции в разных каталогах. Пусть, например, имеется таблица *users* с двумя разбивающими столбцами *date* и *state* (не считая обычных столбцов с данными). Тогда Hive создаст для нее примерно такую структуру каталогов:

```
/user/hive/warehouse/users/date=20090901/state=CA
/user/hive/warehouse/users/date=20090901/state=NY
/user/hive/warehouse/users/date=20090901/state=TX
...
/user/hive/warehouse/users/date=20090902/state=CA
/user/hive/warehouse/users/date=20090902/state=NY
/user/hive/warehouse/users/date=20090902/state=TX
...
/user/hive/warehouse/users/date=20090903/state=CA
/user/hive/warehouse/users/date=20090903/state=NY
/user/hive/warehouse/users/date=20090903/state=TX
...
```

Все данные о пользователях, проживавших в Калифорнии (`state=CA`) 1 сентября 2009 года (`date=20090901`), находятся в одном каталоге, а данные из разных секций – в разных каталогах. Если в запросе требуется найти информацию о пользователях из Калифорнии на 1 сентября 2009 года, то Hive должен будет обработать данные в одном каталоге, не обращая внимания на данные таблицы *users*, принадлежащие другим секциям. Если в запросе участвуют диапазоны значений в разбивающих столбцах, то придется обработать несколько каталогов, но все равно Hive сможет избежать полного сканирования таблицы *users*. В некотором смысле секционирование в Hive дает такие же преимущества, как индексирование в традиционных реляционных базах данных, хотя избирательность секционирования, конечно, гораздо ниже. Желательно, чтобы каждая секция была достаточно велика для эффективной работы задачи MapReduce.

Помимо секций, в модели данных Hive есть понятие *сегмента* (*bucket*). Сегменты обеспечивают эффективное выполнение запросов, которые можно без ущерба для результата выполнять на случайной выборке данных (например, при вычислении среднего по столбцу

[5] На практике пришлось бы отдельно рассматривать округ Колумбия и находящиеся в нем территории.

случайная выборка из множества значений этого столбца может дать хорошее приближение). Под сегментированием понимается разбиение на заданное количество файлов, исходя из значения хеша сегментирующего столбца. Если бы мы задали разбиение столбца таблицы *users*, содержащего идентификатор пользователя, на 32 сегмента, то полная структура таблицы в Hive выглядела бы так:

```
/user/hive/warehouse/users/date=20090901/state=CA/part-00000
...
/user/hive/warehouse/users/date=20090901/state=CA/part-00031
/user/hive/warehouse/users/date=20090901/state=NY/part-00000
...
/user/hive/warehouse/users/date=20090901/state=NY/part-00031
/user/hive/warehouse/users/date=20090901/state=TX/part-00000
...
```

В каждой секции было бы 32 сегмента. Hive знает, что в каждом файле `part-00000` … `part-00031` хранится случайное число пользователей. Точность агрегированной статистики останется достаточно высокой и при вычислении по выборке из набора данных. Для такого рода запросов сегментирование особенно полезно, ведь Hive может выполнить запрос на 1/32 части всех пользователей в некоторой секции, обработав лишь данные из файла `part-00000`, а в остальные даже не заглядывая. Hive может производить выборочную обработку и без сегментов (а также по столбцам, отличным от сегментирующих), но в таком случае ему придется сканировать все данные и игнорировать большую их часть. Поэтому все преимущества выборочной обработки теряются.

Управление таблицами

Мы уже видели, как создать простую таблицу для набора данных о цитировании патентов. Теперь рассмотрим по частям более сложную команду создания таблицы. Следующая команда создает таблицу с именем `page_view`.

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY (dt STRING, country STRING)
CLUSTERED BY (userid) INTO 32 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS SEQUENCEFILE;
```

Первая часть выглядит так же, как эквивалентная команда SQL:

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User')
```

Задается имя таблицы (page_view) и ее схема, содержащая имена и типы столбцов:

- *TINYINT* – 1-байтовое целое;
- *SMALLINT* – 2-байтовое целое;
- *INT* – 4-байтовое целое;
- *BIGINT* – 8-байтовое целое;
- *DOUBLE* – число с плавающей точкой двойной точности;
- *STRING* – последовательность символов.

Бросается в глаза отсутствие типа Boolean, вместо которого обычно используется TINYINT. В Hive имеются и составные типы: структуры, словари и массивы, причем они могут быть вложенными. Но в настоящее время для них нет языковой поддержки, так что эта тема считается более сложной. К каждому столбцу можно присоединить комментарий, как мы сделали для столбца `ip`. Мы добавили также комментарий ко всей таблице:

```
COMMENT 'This is the page view table'
```

В следующей части команды CREATE TABLE описываются разбивающие столбцы:

```
PARTITIONED BY (dt STRING, country STRING)
```

Выше уже отмечалось, что запросы к разбивающим столбцам оптимизируются. Они отличаются от обычных столбцов данных `viewTime`, `userid`, `page_url`, `referrer_url` и `ip`. Значение разбивающего столбца не хранится в строке явно, а берется из пути к каталогу. Но синтаксически запросы к разбивающим и обычным столбцам ничем не отличаются.

```
CLUSTERED BY (userid) INTO 32 BUCKETS
```

Фраза `CLUSTERED BY (...) INTO ... BUCKETS` описывает сегментирование, в том числе столбец, из которого берется случайная выборка, и количество сегментов. При задании количества сегментов нужно учитывать следующее:

1. Размер данных в каждой секции.
2. Желаемый размер выборки.

Первая характеристика важна потому, что размер каждого сегмента секции не должен быть слишком мал, иначе Hadoop будет работать неэффективно. С другой стороны, размеры всех сегментов должны быть не больше желательного размера выборки. Разбиение на 32 сегмента по идентификатору пользователя приемлемо, если в выборку должно попасть примерно 3 процента (~1/32) всех пользователей.

Примечание. В отличие от секционирования, Hive автоматически не обеспечивает сегментирование при записи данных в таблицу. Задание параметров сегментирования всего лишь говорит Hive, что вы сами позаботились о записи данных в нужные сегменты, и Hive может воспользоваться плодами ваших трудов при обработке запросов. Для поддержки критерия сегментирования следует правильно задать количество редукторов при заполнении таблицы. Дополнительные сведения можно найти по адресу http://wiki.apache.org/hadoop/Hive/LanguageManual/DDL/ BucketedTables.

Фраза `ROW FORMAT` говорит Hive, как хранить данные строки. Без нее Hive по умолчанию считает, что разделителем строк является символ новой строки, а разделителем полей – символ с ASCII-кодом 001 (Ctrl-A). В примере выше мы задали в качестве разделителя полей знак табуляции. Мы также указали, что разделителем строк будет символ новой строки, но это и так подразумевается по умолчанию, так что фраза включена лишь для иллюстрации:

```
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
```

Наконец, последняя фраза сообщает Hive о формате файла для хранения данных таблицы:

```
STORED AS SEQUENCEFILE;
```

В настоящее время Hive поддерживает два формата: `SEQUENCEFILE` и `TEXTFILE`. Первый – файл последовательности – это сжатый формат, который обычно обеспечивает более высокую производительность.

В команду `CREATE TABLE` можно включить модификатор `EXTERNAL`, тогда таблица будет создана в существующем каталоге, путь к которому следует указать:

```
CREATE EXTERNAL TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING, ip STRING)
LOCATION '/path/to/existing/table/in/HDFS';
```

Создав таблицу, вы можете запросить у Hive ее схему командой `DESCRIBE`:

```
hive> DESCRIBE page_view;
```

Для изменения структуры таблицы предназначена команда ALTER. Можно изменить имя:

```
hive> ALTER TABLE page_view RENAME TO pv;
```

добавить новые столбцы:

```
hive> ALTER TABLE page_view ADD COLUMNS (newcol STRING);
```

или удалить раздел:

```
hive> ALTER TABLE page_view DROP PARTITION (dt='2009-09-01');
```

Для удаления таблицы целиком служит команда DROP TABLE:

```
hive> DROP TABLE page_view;
```

Можно вывести перечень таблиц, которыми управляет Hive:

```
hive> SHOW TABLES ;
```

Если таблиц так много, что полный список оказывается слишком длинным, его можно сократить, указав регулярное выражение в синтаксисе Java:

```
hive> SHOW TABLES 'page_.*';
```

Загрузка данных

Загрузить данные в таблицу Hive можно разными способами, но в любом случае основу составляет команда LOAD DATA:

```
hive> LOAD DATA LOCAL INPATH 'page_view.txt'
    > OVERWRITE INTO TABLE page_view;
```

Эта команда загружает содержимое локального файла page_view.txt в таблицу page_view. Если опустить модификатор OVERWRITE, новые данные будут добавлены в таблицу без стирания уже существующих в ней данных. Если отсутствует модификатор LOCAL, то файл берется из HDFS, а не из локальной файловой системы. Команда LOAD DATA позволяет также указать имя конкретной секции, в которую следует загружать данные:

```
hive> LOAD DATA LOCAL INPATH 'page_view.txt'
    > OVERWRITE INTO TABLE page_view
    > PARTITION (dt='2009-09-01', country='US');
```

При работе с данными из локальной файловой системы имеется возможность выполнять команды Unix, не покидая командный интерфейс Hive. Для этого достаточно перед именем команды ввести восклицательный знак (!) и в самом конце команды поставить точку с запятой (;). Вот, например, как можно получить список файлов:

```
hive> ! ls ;
```

А так – вывести первые несколько строк файла:

```
hive> ! head hive_result ;
```

Отметим, что пробелы, отделяющие ! и ; от команды, необязательны. Мы добавили их только для наглядности.

Выполнение запросов

В целом запросы на HiveQL удивительно напоминают SQL-запросы. Одно из отличий состоит в том, что результаты HiveQL-запросов относительно велики. Почти всегда следует включать фразу INSERT, чтобы Hive где-то сохранил результат запроса. Часто таким местом является другая таблица:

```
INSERT OVERWRITE TABLE query_result
```

Но это может быть также каталог в HDFS:

```
INSERT OVERWRITE DIRECTORY '/hdfs_dir/query_result'
```

или локальный каталог:

```
INSERT OVERWRITE LOCAL DIRECTORY '/local_dir/query_result'
```

Простые запросы выглядят почти так же, как в SQL:

```
INSERT OVERWRITE TABLE query_result
SELECT *
FROM page_view
WHERE country='US';
```

Обратите внимание, что этот запрос по разбивающему столбцу (country) синтаксически ничем не отличается от запроса по обычному столбцу данных. Следует отметить одну существенную синтаксическую особенность: в HiveQL пишут COUNT(1) в тех местах, где в SQL обычно употребляется COUNT(*). Например, чтобы найти, сколько раз страница просматривалась посетителями из США, нужно написать такой запрос на HiveQL:

```
SELECT COUNT(1)
FROM page_view
WHERE country='US';
```

Как и в SQL, фраза GROUP BY позволяет выполнять агрегатные запросы на группах. Следующий запрос выведет счетчики просмотра страниц по странам:

```
SELECT country, COUNT(1)
FROM page_view
GROUP BY country;
```

А этот запрос выводит счетчики уникальных посетителей из каждой страны:

```
SELECT country, COUNT(DISTINCT userid)
FROM page_view
GROUP BY country;
```

В табл. 11.2 приведены все операторы, поддерживаемые в HiveQL. Они хорошо знакомы по SQL и языкам программирования, поэтому подробно объяснять их мы не будем. Исключение составляет сопоставление с регулярным выражением. В HiveQL есть два оператора для этой цели: `LIKE` и `REGEXP` (`RLIKE` – то же самое, что `REGEXP`). Оператор `LIKE` выполняет стандартное для SQL сопоставление: знак подчеркивания (`_`) в `B` сопоставляется с любым одиночным символом в `A`, а знак процента (`%`) – с любым числом произвольных символов. Оператор `REGEXP` трактует `B` как полноценное регулярное выражение в синтаксисе Java[6].

Таблица 11.2. Стандартные операторы в HiveQL.

Тип оператора	Операторы
Сравнение	`A = B, A <> B, A < B, A <= B, A > B, A >= B, A IS NULL, A IS NOT NULL, A LIKE B, NOT A LIKE B, A RLIKE B, A REGEXP B`
Арифметические	`A + B, A - B, A * B, A / B, A % B`
Поразрядные	`A & B, A | B, A ^ B, ~A`
Логические	`A AND B, A && B, A OR B, A || B, NOT A, !A`

Одна из основных причин, по которым пользователи ищут язык высокого уровня, например Pig Latin или HiveQL, – поддержка операции соединения. В настоящее время HiveQL поддерживает только эквисоединение (соединение по равенству). В качестве примера приведем такой запрос:

```
INSERT OVERWRITE TABLE query_result
SELECT pv.*, u.gender, u.age
FROM page_view pv JOIN user u ON (pv.userid = u.id);
```

С точки зрения синтаксиса, во фразу `FROM` добавляется ключевое слово `JOIN`, размещаемое между именами таблиц, а затем после ключевого слова `ON` перечисляются столбцы, по которым производится соединение. Для соединения нескольких таблиц эта последовательность повторяется:

```
INSERT OVERWRITE TABLE pv_friends
```

[6] Синтаксис регулярных выражений в Java документирован (в формате Javadoc) на странице http://java.sun.com/j2se/1.4.2/ docs/api/java/util/regex/Pattern.html.

```
SELECT pv.*, u.gender, u.age, f.friends
FROM page_view pv JOIN user u ON (pv.userid = u.id)
JOIN friend_list f ON (u.id = f.uid);
```

Путем модификации фразы FROM в любом запросе можно задать выборочную обработку. Следующий запрос пытается вычислить среднее время просмотра страниц по выборке, хранящейся в первом из 32 сегментов:

```
SELECT avg(viewTime)
FROM page_view TABLESAMPLE(BUCKET 1 OUT OF 32);
```

Общий синтаксис фразы TABLESAMPLE таков:

```
TABLESAMPLE(BUCKET x OUT OF y)
```

Размер выборки для запроса составляет примерно 1/y. Кроме того, y должно кратным или делителем числа сегментов, заданных в момент создания таблицы. Например, если изменить y на 16, то запрос примет вид:

```
SELECT avg(viewTime)
FROM page_view TABLESAMPLE(BUCKET 1 OUT OF 16);
```

Тогда в выборку будет включаться примерно 1 из 16 посетителей (поскольку сегментирование производится по столбцу userid). Таблица по-прежнему разбита на 32 сегмента, но Hive пытается выполнить запрос, обрабатывая только два сегмента: 1 и 17. С другой стороны, если y равно 64, то при выполнении запроса Hive будет обрабатывать только половину данных в одном сегменте. Параметр x служит для указания номера используемого сегмента. Если выборка действительно случайна, то его значение несущественно.

Помимо avg, в Hive встроено много других функций. Некоторые наиболее употребительные перечислены в таблицах 11.3 и 11.4. Если встроенных функций не хватает, то программист может определить свои собственные. Краткое руководство по написанию пользовательских функций имеется на странице http://wiki.apache.org/hadoop/Hive/AdminManual/Plugins.

Таблица 11.3. Встроенные функции.

Функция	Описание
concat(string *a*, string *b*)	Возвращает результат конкатенации строк *a* и *b*.
substr(string *str*, int *start*) substr(string *str*, int *start*, int *length*)	Возвращает подстроку *str*, начинающуюся в позиции *start*. Если не задан необязательный аргумент *length*, то подстрока продолжается до конца строки *str*.

Таблица 11.3. (продолжение)

Функция	Описание
`round(double num)`	Возвращает ближайшее целое (типа BIGINT).
`floor(double num)`	Возвращает наибольшее целое (типа BIGINT), меньшее или равное *num*.
`ceil(double num)` `ceiling(double num)`	Возвращает наименьшее целое (типа BIGINT), большее или равное *num*.
`sqrt(double num)`	Возвращает квадратный корень из *num*.
`rand()` `rand(int seed)`	Возвращает случайное число (для каждой строки свое). Необязательное значение seed позволяет получить детерминированную последовательность случайных чисел.
`Ln(double num)`	Возвращает натуральный логарифм *num*.
`log2(double num)`	Возвращает логарифм *num* по основанию 2.
`log10(double num)`	Возвращает десятичный логарифм *num*.
`log(double num)`	Возвращает натуральный логарифм *num*..
`log(double base, double num)`	Возвращает логарифм *num*. по основанию *base*.
`exp(double a)`	Возводит *e* (основание натуральных логарифмов) в степень *a*.
`power(double a, double b)` `pow(double a, double b)`	Возводит *a* в степень *b*.
`upper(string s)` `ucase(string s)`	Переводит строку *s* в верхний регистр.
`lower(string s)` `lcase(string s)`	Переводит строку *s* в нижний регистр.
`trim(string s)`	Отсекает пробелы с обоих концов строки *s*.
`ltrim(string s)`	Отсекает пробелы с левого конца строки *s*.
`rtrim(string s)`	Отсекает пробелы с правого конца строки *s*.
`regexp(string s, string regex)`	Определяет, соответствует ли строка *s* регулярному выражению `regexp`.
`regexp_replace(string s, string regex, string replacement)`	Возвращает строку, в которой все части *s*, соответствующие регулярному выражению `regexp`, заменены строкой *replacement*.
`day(string date)` `dayofmonth(string date)`	Возвращает день даты или временной метки.

Таблица 11.3. (окончание)

Функция	Описание
`month(string date)`	Возвращает месяц даты или временной метки.
`year(string date)`	Возвращает месяц даты или временной метки.
`To_date(string timestamp)`	Возвращает дату (год-месяц-день), соответствующую временной метке.
`unix_timestamp(string timestamp)`	Преобразует временную метку в формат времени, принятый в Unix.
`from_unixtime(int unixtime)`	Преобразует целое число, описывающее время в Unix, во временную метку.
`date_add(string date, int days)`	Прибавляет указанное количество дней к дате.
`date_sub(string date, int days)`	Вычитает указанное количество дней из даты.
`datediff(string date1, string date2)`	Вычисляет количество дней между двумя датами. Если `date1` раньше, чем `date2`, то результат будет отрицателен.

Таблица 11.4. Встроенные агрегатные функции

Функция	Описание
`count(1)` `count(DISTINCT col)`	Возвращает число членов группы или число различных значений в столбце.
`sum(col)` `sum(DISTINCT col)`	Возвращает сумму всех или только различных значений в столбце.
`avg(col)` `avg(DISTINCT col)`	Возвращает среднее, вычисленное по всем или только по различным значениям в столбце.
`max(col)`	Возвращает максимальное значение в столбце.
`min(col)`	Возвращает минимальное значение в столбце.

11.1.4. Hive: подводя итоги

Hive – это программный слой для организации хранилищ данных, построенный на базе массивно масштабируемой архитектуры Hadoop. Делая упор на структурированные данные, Hive добавляет целый ряд механизмов для повышения производительности (например, секционирование таблиц) и удобства пользования (язык, аналогичный SQL). В результате решать многие часто выполняемые задачи, напри-

мер соединение таблиц, становится проще. Hive сделал технологию Hadoop доступной широкой аудитории аналитиков и вообще не-программистов. В августе 2009 года компания Facebook подсчитала, что 29 процентов ее работников пользуются Hive, причем половина из них никак не связана с разработкой ПО[7].

11.2. Другие проекты, связанные с Hadoop

Экосистема Hadoop постоянно растет. Ниже перечислены так или иначе связанные с Hadoop проекты и поставщики, которые, на наш взгляд, полезны или обладают большим потенциалом. Все они, кроме Aster Data и Greenplum, поставляются с открытым исходным кодом на условиях той или иной лицензии.

11.2.1. HBase

http://hadoop.apache.org/hbase/. HBase – масштабируемое хранилище данных, ориентированное на доступ к структурированным (в какой-то степени) данным для произвольного чтения и записи. Построено по образцу Google Bigtable[8] и предназначено для поддержки больших таблиц с миллиардами строк и миллионами столбцов. В качестве файловой системы используется HDFS. В проект заложена полная распределенность и высокая доступность. В версии 0.20 производительность существенно повышена.

11.2.2. ZooKeeper

http://hadoop.apache.org/zookeeper/. ZooKeeper – служба координации для построения больших распределенных приложений. Может использоваться независимо от каркаса Hadoop Core. Реализует многие типичные для крупных распределенных приложений службы, в

[7] Этот факт приведен на странице http://www.facebook.com/note.php?note_id=114588058858. Почему Facebook решила положить Hadoop в основу своей инфраструктуры, объясняется на странице http://www.facebook.com/note.php?note_id=16121578919. В презентации, размещенной по адресу http://www.slideshare.net/zshao/hive-data-warehousinganalytics-on-hadoop-presentation, дается подробное описание проекта хранилища данных и аналитической подсистемы, которые Facebook построила на базе Hive.

[8] «Bigtable: A Distributed Storage System for Structured Data» by Chang et al., OSDI '06 – Seventh Symposium on Operating System Design and Implementation. http://labs.google.com/papers/bigtable.html.

частности, управление конфигурацией, регистрацию имен, синхронизацию и групповые службы. Исторически сложилось так, что разработчики вынуждены заново изобретать эти службы для каждого распределенного приложения, что отнимает время и чревато ошибками, поскольку правильно реализовать их трудно. Абстрагируя сложность, ZooKeeper упрощает реализацию алгоритмов достижения консенсуса, выбора лидера, протоколов обмена информацией о присутствии и других примитивов и дает разработчику возможность сосредоточиться на семантике приложения. ZooKeeper является важным компонентом других связанных с Hadoop проектов, в частности, HBase и Katta.

11.2.3. Cascading

http://www.cascading.org/. Cascading – это API для сборки и исполнения сложных задач обработки данных с помощью Hadoop. Он абстрагирует модель MapReduce, представляя ее в виде модели обработки данных, состоящей из кортежей, «труб» и «кранов» (источников и стоков). Трубы оперируют потоками кортежей, а в число операций входят `Each`, `Every`, `GroupBy` и `CoGroup`. Трубы можно собирать и вкладывать друг в друга, получая «сборки». Исполняемый «поток» получается, когда мы присоединяем трубную сборку к крану-источнику (данным) и к крану-стоку (данным).

У Cascading и Pig много общих черт, да и проектные цели схожи. Но есть и существенное отличие – оболочка Grunt позволяет легко выполнять произвольные запросы. Другое отличие заключается в том, что программы для Pig пишутся на языке Pig Latin, тогда как Cascading больше напоминает Java-каркас, в котором поток обработки данных создается путем конструирования объектов различных Java-классов (`Each`, `Every` и прочих). Для использования Cascading не потребуется изучать новый язык, а созданный поток обработки данных может оказаться более эффективным, так как построен вручную.

11.2.4. Cloudera

http://www.cloudera.com/. Компания Cloudera пытается сделать для Hadoop то, что RedHat сделала для Linux. Она поддерживает Hadoop и создает его дистрибутивы, стремясь сделать работу с ним простой и приятной для корпоративных пользователей. Компания проводит семинары в крупных городах и публикует учебные видеоролики на

своем сайте. Собранный ей бесплатный дистрибутив в формате пакетов RPM или Ubuntu/Debian упрощает развертывание Hadoop. Дистрибутив основан на последней стабильной версии и включает полезные (и протестированные) фрагменты из будущих версий, а также дополнительные инструменты, в частности, Pig и Hive. Cloudera также предлагает консультационные услуги и техническую поддержку компаниям, желающим использовать Hadoop.

11.2.5. Katta

http://katta.sourceforge.net/. Поскольку истоки Hadoop лежат в поисковых системах, неудивительно, что он применяется для распределенного индексирования и поиска. На базе Hadoop построена поисковая система Nutch для веб[9]. Однако Nutch предъявляет много уникальных требований и для конкретных поисковых приложений может не подойти.

Katta представляет собой масштабируемую, отказоустойчивую распределенную систему индексирования, менее громоздкую, чем Nutch, и вместе с тем более гибкую. В каком-то смысле она наделяет Lucene дополнительными возможностями (репликация, избыточность, отказоустойчивость и масштабируемость), сохраняя при этом основную прикладную семантику.

11.2.6. CloudBase

http://cloudbase.sourceforge.net/. CloudBase – это программный слой для организации удовлетворяющих стандарту ANSI SQL хранилищ данных, построенный на основе Hadoop. В отличие от Hive, CloudBase работает непосредственно с плоскими файлами, не нуждаясь в хранилище метаданных. Перед продуктом была поставлена цель обеспечить более строгое соблюдение стандарта ANSI SQL, и взаимодействие происходит в основном через драйвер JDBC, что упрощает подключение к инструментам бизнес-анализа и формирования отчетов. По сути своей, CloudBase – это компилятор, который транслирует SQL-запросы в MapReduce-программы. На момент написания этой книги сообщество активных разработчиков CloudBase было меньше, чем у Pig или Hive. К тому же продукт распространяется на условиях лицензии GPL, более ограничительной, чем лицензия Apache.

[9] Точнее говоря, система Nutch послужила стимулом для создания Hadoop. Исторический обзор см. в главе 1.

11.2.7. Aster Data и Greenplum

http://www.asterdata.com/; http://www.greenplum.com/. Компании Aster Data Systems и Greenplum предлагают коммерческие решения для организации высокопроизводительных масштабируемых хранилищ данных, в которых SQL тесно переплетен с MapReduce. Хотя оба решения поддерживают модель программирования MapReduce, они создавались независимо от Hadoop и в основу дизайна положены другие принципы. В отличие от Hadoop, архитектура этих продуктов в гораздо большей степени ориентирована на корпоративных заказчиков, которым нужны высокопроизводительные хранилища данных на базе SQL. Поскольку эти компании пришли к парадигме MapReduce с другого конца, то изучение созданного ими программного обеспечения помогает лучше понять природу некоторых архитектурных компромиссов, принятых в Hadoop.

11.2.8. Hama и Mahout

http://incubator.apache.org/hama/; http://lucene.apache.org/mahout/. Hama и Mahout предназначены для обработки научных данных с помощью Hadoop. Hama – это пакет для выполнения операций над матрицами: умножения, обращения, нахождения собственных значений и собственных векторов и т. д. Mahout в большей степени ориентирован на реализацию алгоритмов машинного обучения (более подробные сведения можно найти в книге «Mahout in Action» издательства Manning Publications). Версия Mahout 0.1 была выпущена в апреле 2009 и включала такие алгоритмы, как наивная байесовская классификация, кластеризация методом K средних и коллаборативная фильтрация.

На момент написания этой книги оба проекта еще сравнительно молоды и находятся в инкубаторе Apache. Интересующиеся читатели могут принять участие в разработке.

11.2.9. search-hadoop.com

Любому программисту для Hadoop часто бывает нужна та или иная документация по Hadoop или его подпроектам. Компания Sematext, специализирующаяся на поиске и аналитических исследованиях, организовала сайт http://search-hadoop.com/, который содержит средства для поиска по всем подпроектам и источникам данных о Hadoop – архивам списков рассылки, вики-сайтам, системам

отслеживания неполадок, исходному коду и т. д. Поисковая система постоянно обновляется. Результаты поиска можно фильтровать по проекту, по источнику данных и по автору, а также сортировать по дате, релевантности или комбинации того и другого.

11.3. Резюме

В этой главе мы рассмотрели ряд дополнительных инструментов, используемых в сочетании с Hadoop. Мы уделили особое внимание пакету Hive для организации хранилищ данных, поскольку он позволяет обрабатывать данные средствами Hadoop, но на SQL-подобном языке. Вокруг Hadoop выросла богатая экосистема вспомогательного программного обеспечения, и в следующей главе мы покажем, как оно применяется на практике.

ГЛАВА 12.
Примеры применения

В этой главе:

- New York Times.
- China Mobile.
- StumbleUpon.
- IBM.

В этой книге было приведено немало упражнений и примеров программ. Следующий шаг – применить все, что вы узнали о Hadoop, к разработке собственных реальных приложений. Чтобы помочь вам в этом начинании, мы собрали в этой главе несколько примеров того, как различные корпорации используют Hadoop для решения стоящих перед ними задач обработки данных.

У рассмотрения практических примеров двоякая цель. Во-первых, отступить на шаг и взглянуть на большие системы, важнейшей частью которых является Hadoop и вспомогательный инструментарий – Cascading, HBase, Jaql. Во-вторых, продемонстрировать, насколько разнообразны организации, применяющие Hadoop для разрешения рабочих проблем. Мы рассмотрим такие отрасли промышленности, как средства массовой информации (газета New York Times), телекоммуникации (China Mobile), Интернет (сайт StumbleUpon) и корпоративное программное обеспечение (IBM).

12.1. Преобразование 11 миллионов изображений из архива газеты New York Times

В 2007 году газета New York Times решила открыть на своем сайте бесплатный доступ ко всем материалам, опубликованным в период

между 1851 и 1922 годом. Для этого понадобилась масштабируемая система преобразования изображений. Все старые статьи хранились в виде отсканированных TIFF-файлов, а нужно было объединить различные части каждой статьи в единый файл в формате PDF. Раньше доступ к этим статьям был платным, и желающих ознакомиться с ними находилось немного. Times могла бы конвертировать TIFF-изображения – масштабировать, склеить и преобразовать в другой формат – в реальном масштабе времени. При небольшом количестве запросов такой подход мог бы оказаться работоспособным, но не масштабировался на случай ожидаемого роста трафика, связанного с переводом статей в бесплатный доступ. Необходимо было найти более подходящую архитектуру.

Было принято решение заранее сгенерировать файлы всех статей в формате PDF и выложить их как обычное статическое содержимое. New York Times уже располагала кодом для преобразования TIFF-изображений в PDF-файлы. Таким образом, нужно было всего лишь организовать массовую пакетную обработку всех статей, не дожидаясь, пока поступит запрос на конкретную статью. Беда была в том, что в архиве содержалось 11 миллионов статей общим объемом 4 ТБ.

Дерек Готфрид, работавший программистом в Times, решил, что это отличная возможность воспользоваться службами Amazon Web Services (AWS) и Hadoop. Сохранение окончательного набора PDF-файлов в облаке Simple Storage Service (S3) , откуда они могли бы выдаваться посетителю по запросу, к тому времени уже было признано экономически более выгодным, чем расширение системы хранения, обслуживающей веб-сайт. Так почему бы не организовать и преобразование в формат PDF в том же облаке AWS?

Дерек скопировал 4 ТБ TIFF-изображений в S3. Он «приступил к написанию программы, которая извлекала бы из S3 все части одной статьи, генерировала PDF-файл и сохраняла его в S3. Это оказалось не так уж сложно благодаря JetS3t – набору инструментов для работы с S3 с открытым исходным кодом на Java, библиотеке iText для работы с PDF и расширению Java Advanced Image Extension»[1]. Модифицировав свою программу с учетом особенностей Hadoop, Дерек развернул ее в кластере Hadoop из 100 узлов в облаке Amazon Elastic Compute Cloud (EC2). Задача работала 24 часа и сгенерировала еще 1,5 ТБ данных, также сохраненных в S3.

При цене 10 центов за аренду одного экземпляра в час стоимость вычислительной части работы составила всего 240 долларов (100 эк-

[1] http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.

земпляров × 24 часа × $0,10). Плата за хранение данных в S3 сюда не входит, но, поскольку Times все равно приняла решение хранить архив файлов в S3, то эти расходы уже амортизировались. Так как передача данных между S3 и EC2 бесплатна, то сетевой трафик, связанный с выполнением задачи Hadoop, обошелся даром.

Все программирование потребовало усилий только одного человека. Благодаря работе Дерека людям стало гораздо проще знакомиться с освещением исторических событий на страницах New York Times.

12.2. Добыча данных в компании China Mobile

Материал любезно предоставлен Чжи Гуо Лю (ZHIGUO LUO), Мень Ху (MENG XU) и Шаолинь Сун (SHAOLING SUN) из Научно-исследовательского института корпорации China Mobile Communication.

China Mobile Communication Corporation (CMCC) – крупнейший оператор мобильной связи в мире. На Нью-Йоркской фондовой бирже его акциям присвоен символ CHL. В рейтинге мировых брендов за 2009 год China Mobile уступает McDonald's и Apple, но опережает General Electric. Контролируя более двух третей китайского рынка мобильной телефонии, CMCC обсуживает нужды 500 миллионов абонентов. И даже при таком размере China Mobile продолжает быстро расти. Например, в 2006 году, когда абонентская база насчитывала всего 300 миллионов, годовой прирост абонентов составил 22 процента, потребление услуг голосовой связи выросло на 30 процентов, а количество SMS-сообщений – на 41 процент.

Как и любой оператор связи, China Mobile порождает гигантский объем данных в ходе нормальной работы сети. Например, при каждом звонке генерируется запись о вызове (call data record – CDR), в которой хранится номер вызывающего абонента, номер вызываемого абонента, время начала и продолжительность вызова, информация о маршрутизации вызова и т. д. Помимо записей о вызове, телефонная сеть генерирует данные о сигналах, которыми обмениваются различные коммутаторы, узлы и оконечные устройства. Все эти данные нужны, как минимум, для завершения вызовов и выставления счетов клиентам. Но они используются также для маркетинговых исследований, оптимизации сети и других целей.

Естественно, что из-за размера сети China Mobile объем порождаемых данных очень велик. Каждый день генерируется от 5 до 8 ТБ только в виде записей о вызовах. В филиалах China Mobile может насчитываться более 20 миллионов абонентов, что дает более 100 ГБ CDR для голосовых вызовов и от 100 до 200 ГБ CDR для SMS-сообщений ежедневно. Кроме того, типичный филиал каждый день генерирует около 48 ГБ данных о передаче сигналов в системе GPRS (General Packet Radio Service) и 300 ГБ данных о передаче сигналов в системе 3G.

China Mobile ищет решения для организации хранилища этих данных и их анализа с целью извлечения полезной информации для улучшения маркетинга, оптимизации сети и предоставления услуг. Вот некоторые типичные приложения:

- анализ поведения пользователей;
- предсказание оттока клиентов;
- анализ подключения услуг;
- анализ качества обслуживания в сети;
- анализ сигнальной информации;
- фильтрация спама.

China Mobile имела опыт работы с коммерческими средствами добычи данных от известных производителей. Но их архитектура построена так, что налагает ограничения на имеющуюся у China Mobile систему анализа, поскольку требует, чтобы все данные обрабатывались на одном сервере. В этом случае узким местом становится пропускная способность оборудования. В настоящее время в одном из филиалов развернуто коммерческое решение, включающее сервер под управлением Unix с 8 процессорными ядрами, 32 ГБ оперативной памяти и дисковым массивом хранения. Оно способно анализировать поведение лишь 1,4 миллиона клиентов, то есть около 10 процентов данных, генерируемых этим филиалом. Но даже при таких ограничениях на объем обрабатываемых данных быстродействие текущей системы недостаточно для многих приложений. К тому же, высокопроизводительные Unix-серверы и системы хранения данных дороги, а коммерческий продукт плохо поддерживает специализированные алгоритмы.

Принимая во внимание ограничения существующей системы, China Mobile решила запустить экспериментальный проект по разработке параллельного инструмента добычи данных на основе Hadoop

и сравнить его с имеющейся системой. Проект получил название Big Cloud-based Parallel Data Mining (BC-PDM – параллельная добыча данных на базе большого облака), перед ним было поставлено четыре цели:

- *массивная масштабируемость* – использование Hadoop для масштабирования по горизонтали;
- *низкая стоимость* – построение кластера из стандартного оборудования с использованием бесплатного ПО;
- *настраиваемость* – приложения должны создаваться с учетом конкретных требований бизнеса;
- *простота использования* – наличие графических интерфейсов пользователя, аналогичных тем, что применяются в коммерческих программах.

В BC-PDM реализовано много стандартных операций извлечения-преобразования-загрузки (ETL) и алгоритмов добычи данных в контексте MapReduce. К числу ETL-операций относятся вычисление агрегированной статистики, обработка атрибутов, выборка данных, устранение избыточности и прочие. Реализовано девять алгоритмов добычи данных из трех категорий: кластеризация (методом K средних), классификация (в частности, алгоритм C4.5), и поиск ассоциативных правил (например, Apriori). MapReduce-программы исполнялись в кластере Hadoop, состоящем из 256 узлов, подключенных к одному гигабитному коммутатору с 264 портами. В узлах было установлено следующее оборудование:

- *Datanode/TaskTracker* – один четырехъядерный процессор Xeon с тактовой частотой 2,5 ГГц, объемом оперативной памяти 8 ГБ и 4 дисками SATA II по 250 ГБ;
- *Namenode/JobTracker* – 2 двухъядерных процессора AMD Opteron с тактовой частотой 2,6 ГГц , объемом оперативной памяти 16 GB и 4 дисками SAS по 146 ГБ.

China Mobile провела сравнение BC-PDM с существующей системой на реальных данных из системы поддержки бизнес-аналитики (Business Analysis Support System –BASS). Всего было использовано три разных набора данных, все довольно большие. Для некоторых задач были необходимы меньшие выборочные поднаборы. Размеры исходных наборов, а также выборочных поднаборов приведены в табл. 12.1 в столбцах «Большой», «Средний» и «Малый».

Таблица 12.1. Размеры наборов, использованных для сравнения систем

Данные	Большой	Средний	Малый
Поведение пользователей	12 ТБ	120 ГБ	12 ГБ
Доступ пользователей	16 ТБ	160 ГБ	16 ГБ
Подключение новых услуг	120 ГБ	12 ГБ	1,2 ГБ

China Mobile оценивала BC-PDM по четырем направлениям: корректность, производительность, стоимость и масштабируемость. Разумеется, некорректно работающая система бесполезна. Для доказательства корректности результаты параллельного выполнения ETL-операций в BC-PDM сравнивались с результатами последовательного выполнения тех же операций в существующей системе. С другой стороны, не следовало ожидать, что алгоритмы добычи данных дадут *в точности те же* результаты, что существующая система. Различия обусловлены несущественными особенностями реализации и исполнения; в частности, на результат могли повлиять начальные условия и упорядоченность входных данных. Полученные результаты оценивались с точки зрения общей согласованности. Кроме того, для проверки параллельных алгоритмов добычи данных использовались наборы данных из репозитория UCI. Эти наборы широко применяются исследователями в области машинного обучения и хорошо изучены. Поэтому China Mobile имела возможность сравнить результаты работы BC-PDM с известными моделями.

После того как корректность реализаций MapReduce была установлена, проводилось сравнение производительности BC-PDM с текущей системой. Для этого использовался только кластер Hadoop с 16 узлами. Как мы увидим, такой небольшой кластер обходится дешевле монолитного сервера, обслуживающего текущую систему. На рис. 12.1 показано время работы обеих систем для ETL-операций (левая диаграмма) и добычи данных (правая диаграмма).

Отметим, что BC-PDM была настроена на обработку в *10 раз больше* данных, чем текущая система. BC-PDM оказалась быстрее на всех ETL-операциях, причем общая производительность была выше в 12–16 раз. На задачах добычи данных BC-PDM была подвергнута нагрузочному тестированию с объемом данных в 100 раз больше, чем

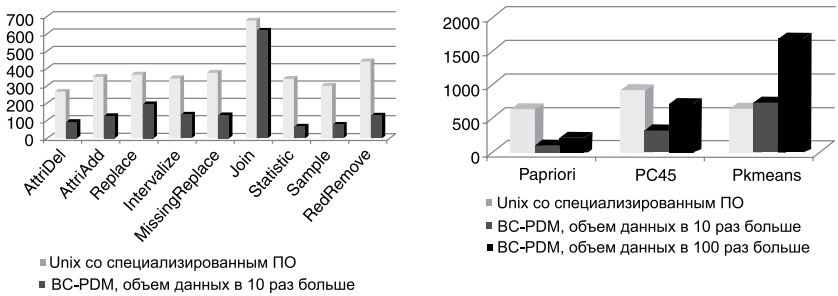


Рис. 12.1. Сравнение производительности кластера Hadoop и существующей коммерческой системы на базе Unix-сервера. Левая диаграмма относится к ETL-операциям, правая — к алгоритмам добычи данных.

для текущей системы. И даже при увеличении объема на два порядка BC-PDM работала быстрее текущей системы при выполнении алгоритмов Apriori (поиск ассоциативных правил) и C4.5 (классификация). Алгоритм кластеризации по методу K средних работал чуть дольше, чем в текущей системе, при десятикратном объеме данных. Полное тестирование трех приложений, эксплуатируемых в Шанхайском филиале, продемонстрировало увеличение производительности от 3 до 7 раз. Эти реальные приложения предназначены для моделирования предпочтительного выбора каналов, подключения к новым услугам и разбиения абонентов на группы. Напомним, что во всех тестах BC-PDM работала на относительно небольшом кластере Hadoop с 16 узлами. Ниже мы увидим, что BC-PDM и Hadoop прекрасно масштабируются при добавлении новых узлов. Ожидается, что на производственном кластере с 256 узлами BC-PDM сможет сохранять, обрабатывать и анализировать данные объемом порядка 100 ТБ.

Мало того что кластер BC-PDM с 16 узлами показал более высокую производительность, он еще и дешевле текущей системы. В табл. 12.2 показано, как сократились затраты (на момент написания этой статьи 1 доллар США стоил чуть меньше 7 юаней). Стоимость кластера Hadoop/BC-PDM с 16 узлами составляет примерно одну пятую от стоимости текущей коммерческой системы. Основная экономия обусловлена использованием дешевых стандартных серверов. Кластер с 16 узлами обошелся более чем в шестнадцать раз дешевле совокупной стоимости оборудования, необходимого для текущего решения.

Таблица 12.2. Сравнение цены и конфигурации существующего решения и кластера Hadoop с 16 узлами (на момент написания этой статьи 1 доллар США стоил чуть меньше 7 юаней)

		BC-PDM (16 узлов)	Существующее коммерческое решение на базе Unix-сервера
Стоимость оборудования	Вычислительная мощность	ЦП: 64 ядра Память: 128 ГБ	ЦП: 8 ядер Память: 32 ГБ
	Емкость системы хранения	16 ТБ (4 диска SATA II по 256 ГБ в каждом узле)	Массив хранения
	Стоимость	240 000 юаней	4 000 000 юаней
Стоимость ПО	СУБД	500 000 юаней	1 000 000 юаней
	Прикладное ПО	300 000 юаней	500 000 юаней
	Стоимость сопровождения	200 000 юаней	500 000 юаней
Итого		1 240 000 юаней	6 000 000 юаней

До сих пор мы рассматривали корректность, производительность и стоимость новой системы BC-PDM. Теперь перейдем к масштабируемости системы при добавлении узлов в кластер. Мы тестировали ETL-операции и алгоритмы добычи данных в кластерах с 32, 64 и 128 узлами. Измерялось увеличение быстродействия, причем за эталон было взято время выполнения в кластере с 32 узлами. Результаты представлены на рис. 12.2, где левая диаграмма относится к ETL-операциям, а правая – к добыче данных. Масштаб на горизонтальной оси (по которой откладывается размер кластера) логарифмический, равноотстоящие деления шкалы соответствуют величинам 32, 64 и 128. Лучшее, на что можно было рассчитывать, – линейная масштабируемость, поэтому в идеале график должен был бы показать рост быстродействия в два и в четыре раза. Как видим, многие ETL-операции близки к этому идеалу. На самом деле, при увеличении размера кластера в четыре раза (с 32 до 128 узлов) скорость всех операций, кроме двух, возросла более чем в 2,5 раза[2]. Алгоритмы добычи дан-

[2] Исключение составили операции соединения и удаления дубликатов. Время их выполнения было почти постоянным вне зависимости от размера кластера. В настоящее время мы пытаемся понять, в чем причина такой аномалии. Одно из возможных объяснений заключается в том, что данные задачи распределены неравномерно, поэтому какое-то одно задание оказывается узким местом, препятствующим завершению задачи в целом.

ных сложнее, но все же демонстрируют вполне приемлемую масштабируемость. В предыдущей серии тестов (рис. 12.1) использовался небольшой кластер с 16 узлами, который мы даже не рассматриваем как эталон для оценки масштабируемости. И тем не менее даже при 16 узлах кластер сумел обработать на порядок больше данных, чем существующая коммерческая система. В сочетании с результатами тестирования масштабируемости это доказывает, что кластер BC-PDM сможет обработать данные объемом 100 ТБ и выше.

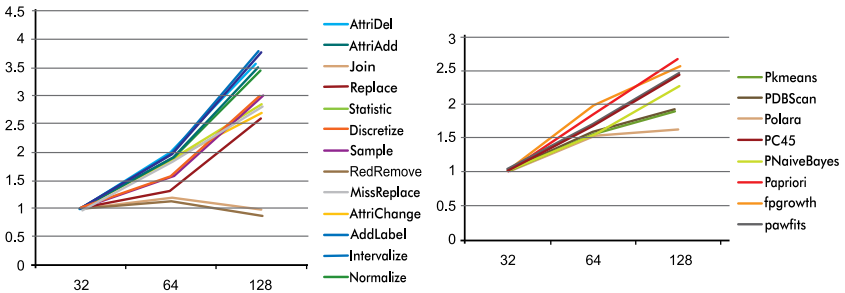


Рис. 12.2. Масштабируемость ETL-операций (слева) и алгоритмов добычи данных (справа) в кластере Hadoop при добавлении узлов. По горизонтальной оси откладывается количество узлов в кластере BC-PDM, а по вертикальной – коэффициент увеличения быстродействия относительно времени выполнения в кластере с 32 узлами.

После тщательного тестирования системы BC-PDM мы вместе с Шанхайским филиалом приступили к ее внедрению для удовлетворения некоторых потребностей бизнеса. В частности, требовалось охарактеризовать абонентскую базу для принятия более точных маркетинговых решений. Конкретно, нужно было выяснить, как сегментированы пользователи, каковы характеристики каждого сегмента и чем они отличаются и как классифицировать пользователей для целевого маркетинга. Для кластеризации абонентской базы мы использовали алгоритм K средних и получили диаграмму сегментации, показанную на рис. 12.3. Последующий анализ позволил охарактеризовать каждый сегмент в соответствии со средней суммой счета и типами подключенных услуг. BC-PDM выполнила этот анализ в 3 раза быстрее, чем существующая система на базе Unix.

Подведем итоги. China Mobile – крупный оператор мобильной связи, которому необходимо анализировать очень большие и постоянно растущие наборы данных. Эксплуатируемое на данный момент коммерческое решение дорого и не годится для анализа данных

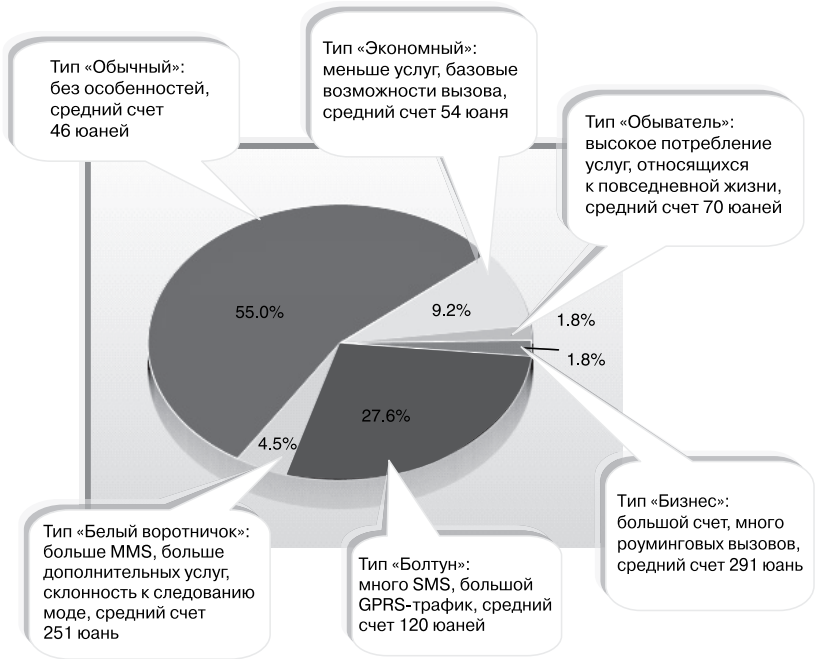


Рис. 12.3. Кластерный анализ абонентской базы Шанхайского филиала China Mobile с использованием алгоритма K средних. Результаты могут найти применение в маркетинговых кампаниях.

о пользователях. Мы изучили возможность применения Hadoop и построили систему добычи данных BC-PDM на базе MapReduce и HDFS. Обнаружилось, что она дает правильные результаты, работает быстро, стоит недорого и хорошо масштабируется. В будущем мы планируем еще повысить эффективность BC-PDM, а также расширить сферу ее применимости, реализовав дополнительные ETL-операции и алгоритмы добычи данных. Более того, мы собираемся использовать BC-PDM как основную платформу для анализа данных во всех филиалах China Mobile.

12.3. Рекомендование лучших веб-сайтов на StumbleUpon

Материал любезно предоставили Кен Макиннис (Ken Macinnis) и Райан Роусон (Ryan Rawson).

Сочетая мнения людей с машинным обучением для быстрого поиска релевантного содержимого, сайт StumbleUpon[3] предлагает только веб-сайты, которые рекомендовали другие посетители с аналогичным складом ума. При нажатии кнопки Stumble вам показывают высококачественный сайт, выбранный на основе коллективного мнения похожих на вас посетителей.

Для формирования коллективной оценки качества сайта StumbleUpon применяет рейтинги «нравится» и «не нравится». Нажав кнопку Stumble, вы увидите только страницы, рекомендованные вашими друзьями и другими посетителями Stumble, думающими так же, как вы. Это позволяет находить материалы, которые было бы трудно отыскать с помощью традиционных поисковых машин.

12.3.1. Как мы пришли к распределенной обработке в StumbleUpon

Для сбора и анализа данных о сайтах, StumbleUpon нуждается в высокодоступной серверной платформе, обрабатывающей миллионы рейтингов ежедневно. В настоящее время на StumbleUpon зарегистрировано около 10 миллионов пользователей, так что возможности традиционной платформы LAMP (Linux, Apache, MySQL, PHP) довольно быстро были исчерпаны. Тогда мы приступили к построению распределенной платформы, и вот почему.

- *Масштабируемость* – во многих случаях стандартное оборудование хорошо масштабируется. Кластер Hadoop с 20 узлами стоит столько же, сколько один сервер с резевированием для размещения базы данных.
- *Свобода разработки* – возможности разработчиков не так ограничены, как при проектировании для РСУБД с тщательно продуманной и подчас хрупкой архитектурой.
- *Эксплуатационные соображения* – для бесперебойной работы сервиса мирового класса важно избавиться от как можно большего количества компонентов, являющихся точками общего отказа системы.
- *Скорость обработки данных* – многие вычисления попросту невозможно выполнить в монолитной системе.

---

[3] Stumble upon означает «наткнуться», «набрести». *Прим. перев.*

12.3.2. HBase и StumbleUpon

HBase играет важнейшую роль в распределенной платформе StumbleUpon. HBase – это распределенная база данных с хранением по столбцам, в основе которой лежит вся мощь Hadoop и HDFS. Но в любой сложной системе приходится идти на компромиссы: HBase отказывается от таких присущих традиционным базам данных концепций, как соединения, внешние ключи и триггеры, ради построения масштабируемой системы, позволяющей хранить гигантские массивы разреженных данных с использованием стандартного оборудования.

Введение в HBASE

HBase устроена по образцу распределенной системы хранения Google Bigtable[4]. Напомним основные факты о Bigtable и подобных ей системах.

- Сочетает идеи, характерные для баз данных с хранением по строкам и по столбцам. Как пишут авторы, Bigtable – это «разреженный распределенный многомерный отсортированный словарь». Основная единица хранения, таблица, разбита на много *таблетов* (tablet) (в терминологии HBase *регионов*).
- Операции записи буферизуются в памяти, а спустя некоторое время сбрасываются в файлы, доступные только для чтения.
- Чтобы число файлов оставалось обозримым, периодически запускается процедура *уплотнения*, которая объединяет *N* файлов в один.
- Для отслеживания местонахождения данных используются специальные таблеты, или регионы.
- Благодаря организации хранения данных по столбцам, накладные расходы на хранение *разреженных* таблиц, в которых много ячеек содержат null, оказываются практически нулевыми, так как значение null явно не хранится.
- Для группировки столбцов из разных строк применяются семейства столбцов. Все столбцы, принадлежащие одному семейству, хранятся вместе (для обеспечения локальности) и описываются одними и теми же параметрами хранения и конфигурации.
- Для каждой ячейки таблицы хранится несколько версий, старые данные не перезаписываются.

[4] Bigtable: A Distributed Storage System for Structured Data. Chang, et al. http://labs.google.com/papers/bigtable.html.

- Емкость (как в плане размера системы хранения, так и в плане скорости обработки) можно увеличить, просто добавив в кластер дополнительные компьютеры; никакого специального конфигурирования не требуется.

HBase предлагает ряд дополнительных возможностей:

- Шлюзы к REST и Thrift[5] обеспечивают простой доступ для сред разработки на языках, отличных от Java.
- Простая интеграция с Hadoop MapReduce для обработки данных.
- Пользуется всеми преимуществами практически проверенной надежности и масштабируемости Hadoop и HDFS.
- Имеются веб-интерфейсы для управления главным и региональными серверами.
- Подкреплена активным сообществом.

На рис. 12.4 представлена упрощенная схема записи данных на *регионном сервере* HBase. Операция записи добавляется в конец журнала записи на сервере.

1. Затем данные помещаются в хранилище MemStore в памяти.
2. Когда размер MemStore превышает пороговую величину, его содержимое сбрасывается в новый файл на диске.
3. Когда на диске оказывается слишком много файлов с данными, производится их уплотнение, в результате чего файлов становится меньше.

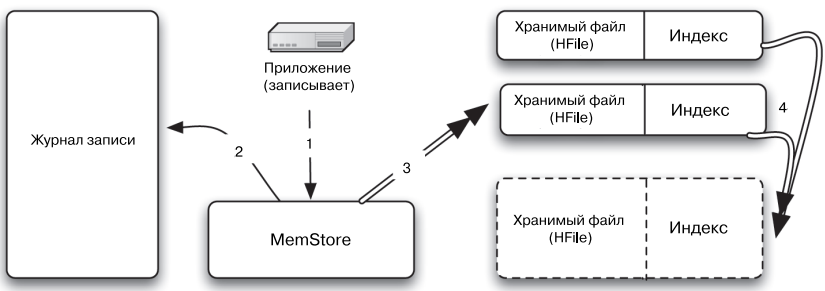


Рис. 12.4. Операции записи в HBase

[5] Thrift – это библиотека для вызова удаленных процедур, первоначально разработанная в Facebook. Теперь она имеет статус инкубаторного проекта Apache и размещается по адресу http://incubator.apache.org/thrift/.

Дополнительную информацию о том, как получить, запустить и улучшить HBase, см. на сайте проекта[6].

Использование HBASE на сайте StumbleUpon

При разработке StumbleUpon база данных HBase в результате тщательного анализа была выбрана из нескольких систем хранения и извлечения информации. Нам важна полная согласованность, когда в результатах запроса, следующего за некоторой операцией записи, эта операция гарантированно учтена. Кроме того, StumbleUpon привержен модели ПО с открытым исходным кодом, позволяющей нам вносить свой вклад в общее дело, а активное сообщество разработчиков, сложившееся вокруг HBase, разделяет эту приверженность и предлагает ценный ресурс, с помощью которого продукт можно совершенствовать.

Первое серьезное испытание, которому мы подвергли HBase, состояло в импорте данных из наших прежних систем на базе MySQL. Раньше мы выполняли эту процедуру только в случае острой необходимости (например, для миграции таблиц или хостов), она занимала от нескольких дней до нескольких недель.

Хранение по строкам и по столбцам

Необходимость в организации хранения данных по столбцам естественно возникает, например, в случае, когда требуется сохранить произвольный набор атрибутов пользователя, разбив их на логически связанные группы. Предположим, что у пользователя есть следующие атрибуты.

- *Контактные данные* – адреса электронной почты и веб-сайта, имена в системе обмена мгновенными сообщениями, URL-адреса фотографий в профиле.
- *Статистические данные* – дата регистрации, дата последнего входа, версия клиента при последнем посещении.
- *Атрибуты* – учетные данные для удаленной аутентификации в сторонней службе;
- *Разрешения* – права доступа к функциям сайта и размещенным на нем данным.

В традиционной реляционной СУБД мы могли сопоставить каждой группе отдельную таблицу. Атрибуты пользователя можно было бы извлекать вместе с основной записью, воспользовавшись внешними ключами и соединением таблиц. При тщательном проектирова-

[6] http://hadoop.apache.org/hbase/.

нии[7] и сравнительно небольшом объеме данных получается гибкая, легко сопровождаемая система. Но если требования изменяются, например, возникает необходимость хранить несколько комплектов учетных данных для каждого пользователя, хотя изначально планировался только один, то переделать проект бывает трудно.

Но настоящие проблемы начинаются, когда объем данных превышает некоторый порог, и схему приходится перерабатывать. Сама мысль о том, чтобы выполнить команду `ALTER TABLE` для таблицы в производственной базе данных, содержащей миллионы или миллиарды строк, вызывает ужас. И это не говоря о трудностях проверки корректности и полноты системных изменений, вносимых в схемы. Даже в случае идеальной, статической таблицы анализ данных затрудняется из-за непрекращающихся операций выборки, ввода и вывода записей.

Взгляните на листинг 12.1. Это простой пример, в котором типичный пользователь имеет лишь идентификатор и одну запись для каждой «находки»:

Листинг 12.1.
Вычисление количества «находок» для каждого пользователя и для каждого URL

```
public class CountUserUrlStumbles {
   public static class Map extends MapReduceBase
         implements Mapper<ImmutableBytesWritable, RowResult,
         Text, Text> {
      @Override
      public void map(ImmutableBytesWritable key,
                      RowResult value,
                      OutputCollector<Text, Text> output,
                      Reporter reporter) throws IOException {
        byte [] row = value.getRow();
        int userid = StumbleUtils.UserIndex.getUserId(row);
        int urlid = StumbleUtils.UserIndex.getUrlId(row);

        Text one = new Text("1");
        output.collect(new Text("U:" + Integer.toString(userid)), one);
        output.collect(new Text("Url:" + Integer.toString(urlid)), one);
      }
   }

   public static class Reduce extends MapReduceBase
         implements Reducer<Text,Text,Text,Text> {
      @Override
```

[7] А правильно спроектировать с первого раза получается редко, потому что окончательные схемы априори обычно неизвестны!

```
    public void reduce(Text key,
                       Iterator<Text> values,
                       OutputCollector<Text, Text> output,
                       Reporter reporter) throws IOException {
       int count = 0;
       while (values.hasNext()) {
          values.next();
          count++;
       }
       output.collect(key, new Text(Integer.toString(count)));
    }
 }

 public static void main(String []args) throws IOException {
    if (args.length < 2) {
      System.out.println("Give the name of the by-userid stumble table");
      return;
    }
    JobConf job = new JobConf(CountUserUrlStumbles.class);
    job.setInputFormat(TableInputFormat.class);
    FileInputFormat.setInputPaths(job, args[0]);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setOutputFormat(TextOutputFormat.class);
    TextOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setNumMapTasks(5000);
    JobClient jc = new JobClient(job);
    jc.submitJob(job);
 }
}
```

В этом примере представлена рутинная для сайта StumbleUpon задача: подсчитать количество «находок» для каждого пользователя и для каждого URL. Ничего особо сложного или поучительного в ней нет, это просто пример конкретной аналитической задачи, которые мы решаем ежедневно. Самое интересное, что эта тривиальная программа всего за час (на кластере из 20 узлов со стандартными компьютерами) обрабатывает порядка десятка миллиардов ключей. Аналогичная программа, работающая с базой данных MySQL, за сколько-нибудь приемлемое время не успевает завершиться – по крайней мере, без специальных мер для выгрузки данных из базы, разбиения всего множества строк на порции разумного размера и последующего объединения результатов.

Эта последовательность операций вам, конечно, знакома: распределение, затем редукция! Применяя обобщенные возможности HBase и Hadoop, мы можем проводить подобные статистические исследования по мере необходимости, без специальной подготовки и мороки с

настройкой среды исполнения. А если перейти к практическим реалиям, то это означает, что все задачи анализа удается завершить в тот день, когда поступил запрос. У нас появилась возможность выполнять произвольные запросы со скоростью, о которой мы и мечтать не могли до внедрения Hadoop и HBase. Бизнес процветает или хиреет в зависимости от того, какие данные он способен анализировать. Ускорение обработки данных, поступающих с сайта, позволило инженерам мгновенно производить анализ поступающей информации на предмет спама.

Трудно даже представить, с какими трудностями мы столкнулись бы при попытке переделать традиционный конвейер обработки в случае не столь тривиальной схемы, не будь в нашем распоряжении распределенной платформы для извлечения, преобразования и анализа данных.

Выходим за пределы одной машины

Мы уже сказали, что одно из важнейших достоинств масштабируемости HBase – возможность преодолеть (в конечном итоге) ограничения по записи на единственной машине.

Обычно масштабирование базы данных сводится к добавлению подчиненных узлов, выполняющих чтение, и механизма кэширования. Но читающие узлы могут помочь лишь в том случае, когда приложение ориентировано преимущественно на чтение, а не на запись. А кэширование дает эффект лишь в случае, если данные редко изменяются. И даже при таких условиях подобные архитектурные решения часто чрезмерно усложняют прикладную часть системы.

HBase может хранить регион на любой машине кластера (каждая машина является *региональным сервером*). Запись затрагивает тот региональный сервер, на котором находится данный регион, а региональный сервер HBase пишет на три (по умолчанию) узла данных HDFS[8]. При наличии большой таблицы и кластера соответствующего размера операции записи распределяются по многим машинам, что в корне устраняет проблему записи на единственной машине, свойственную системам с архитектурой главный-подчиненный.

Это позволяет осуществлять масштабирование, недоступное традиционным реляционным системам, при несравнимо меньших затратах. Поскольку стоимость оборудования растет быстрее, чем достигаемое таким образом увеличение производительности, описанная особенность становится чрезвычайно важной. При такой интенсив-

[8] HDFS записывает данные на несколько узлов, чтобы обеспечить их долговечность, с одной стороны, и повысить производительность за счет локальности, с другой.

ной нагрузке, как на сайте StumbleUpon, экономия может составить миллионы долларов. А некоторые задачи в системе с одним сервером вообще не решаются!

В случае динамично изменяющихся наборов данных, когда часто приходится читать только что записанные данные, кэширование, например с помощью memcached, мало помогает. HBase хранит недавно записанные данные в буфере записи. Чтение таких данных производится непосредственно из оперативной памяти. Тем самым необходимость в механизме кэширования зачастую вообще отпадает.

Примером динамично изменяющихся данных могут служить счетчики событий. Это трудная задача, поскольку большинство высокопроизводительных решений ради достижения максимального быстродействия хранят данные только в оперативной памяти (например, memcached). А ведь необходимо также обеспечить долговечность. И тут на помощь приходит HBase со своей функцией `incrementColumnValue()`. Эта операция рассматривается системой как любое другое изменение, то есть одновременно производится запись в журнал на диске и в буфер записи. Запросы на чтение удовлетворяются из буфера записи, что позволяет добиться как высокой производительности, так и долговечности. StumbleUpon использует изначально встроенную в HBase поддержку счетчиков едва ли не в каждом уголке сайта – показы страниц, переходы по ссылкам, демонстрации рекламных объявлений и т. д.

Более того, HBase предлагает замечательную альтернативу типичным решениям, касающимся секционирования. В большинстве традиционных подходов предполагается наличие априорной информации о пространстве ключей. Если функция хеширования распределяет ключи неравномерно или ключи не удовлетворяют первоначальным допущениям о схеме секционирования, то последствия для производительности могут оказаться очень печальными.

HBase подходит к задаче разбиения таблиц на регионы, опираясь исключительно на размер данных: как только размер региона достигает заранее заданной величины (сейчас по умолчанию 256 МБ), выбирается некий средний ключ и по нему регион расщепляется на две примерно равные части. Каждая часть становится новым регионом, который может продолжать расти. Если эта процедура повторяется 1000 раз, то в результате получится таблица, содержащая 2000 регионов приблизительно одинакового размера. На рис. 12.5 показана работа упрощенного кластера HBase, в котором на пространстве из

256 ключей (0x00-0xFF) одновременно выполняются три операции записи и одна операция чтения.

- Клиент в момент загрузки получает местоположение таблицы ROOT от сервера ZooKeeper.
- Таблица ROOT содержит указатели на другие таблицы META, в которых хранятся сведения о местонахождении пользовательских таблиц.
- Клиент находит регион для выбранной операции. Его местоположение кэшируется на клиенте.
- Запрос посылается выбранному региону для выполнения.

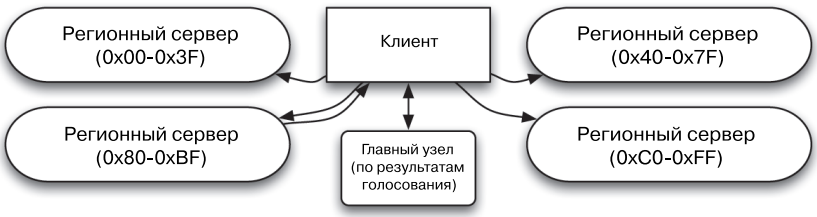


Рис. 12.5. HBase без всякого труда и автоматически осуществляет секционирование

Объем данных на сайте StumbleUpon продолжает расти, но их распределение неравномерно; однако мы не сталкиваемся с проблемой несбалансированных секций или регионов, требующей ручного вмешательства. В большинстве схем ручного секционирования на основе РСУБД эта проблема неизменно возникает.

О быстродействии HBase

Несмотря на все достоинства HBase, на первых порах мы обнаружили, что ее производительность не соответствует требованиям, предъявляемым к системам оперативной обработки данных. Чтобы исправить ситуацию, Райан предложил включить в проект HBase целый ряд усовершенствований, направленных на повышение производительности и надежности. Первым и главным было изменение формата файла HFile. В старом формате были неэффективно реализованы стратегия индексирования, пути чтения и внутренние API. Были выявлены следующие проблемы:

- ориентированность на потоковые операции затрудняла кэширование;
- эффективность индекса зависела от размера данных;

- производилось много операций копирования байтовых массивов;
- частота создания объектов была слишком высока.

HFile – неизменяемый формат. Один раз записанные данные уже нельзя изменить. Код чтения и записи разделен, и не существует никаких методов обновления. Поскольку файлы формата HFile обычно размещаются в HDFS, то это в любом случае было бы невозможно, так как все файлы в системе HDFS неизменяемы.

У компонента записи в HFile простой алгоритм, состоящий всего из четырех шагов:

- открыть файл, установить параметры сжатия, размер блока и аргументы компаратора, эти параметры не изменяются на протяжении всего времени существования файла;
- добавлять ключи в отсортированном порядке, определяемом компаратором, любая попытка добавить ключ не в порядке сортировки приводит к исключению;
- добавить необязательные блоки метаданных, они бывают полезны для включения дополнительных данных или возможностей, например, фильтров Блума;
- закрыть файл, окончательно оформить индекс и записать концевые блоки.

При добавлении в конец HFile ключей и значений программа отслеживает размер текущего блока. Когда он превысит указанный максимальный размер блока, производится сброс на диск, а система сжатия заново инициализируется, готовясь к следующему блоку. Когда компонент записи в HFile дописывает блок, обновляется хранящийся в памяти индекс, в котором для каждого блока записан его первый ключ и смещение от начала файла. При вызове метода close индекс блоков записывается сразу за последним блоком. После него могут находиться необязательные блоки метаданных, а вслед за ними – индексы таких блоков. Наконец, выводится хвостовой блок, содержащий указатели на индексы, и файл закрывается.

Когда файл открывается для чтения, в память загружаются индексы блоков данных и блоков метаданных. Они остаются в памяти все время, пока существует объект читателя. Индекс позволяет быстро находить и считывать блоки данных. Чтобы найти в файле ключ, читатель производит двоичный поиск в индексе. Найдя номер блока, он считывает его, распаковывает и сохраняет в кэше блоков. Затем

программа обходит блок в памяти, пытаясь найти либо сам указанный ключ, либо ближайший к нему. Клиенту возвращаются указатели, позволяющие добраться до единственной хранящейся копии данных.

К достоинствам формата HFile следует отнести простоту идеи и реализации. Весь код классов чтения и записи находится в одном файле (не считая тестов) и занимает примерно 1600 строк.

На базе HFile построен весь внутренний механизм регионного сервера. Внутренние алгоритмы объединения нескольких файлов в один со временем развивались, и настало время взглянуть на них свежим взглядом. Йон Грей (Jon Gray) и Эрик Холстад (Erik Holstad) из компании Streamy.com спроектировали и написали совершенно новую реализацию, добавив семантику удаления и изменив внутренние форматы ключей. За счет применения более эффективных алгоритмов и реализации формата HFile с нуль-копированием удалось добиться дополнительного увеличения скорости.

В целом рост быстродействия оказался весьма внушительным – от 30 до 100 раз в зависимости от операции. Меньше всего – в 30 раз – увеличилась скорость последовательного сканирования строк. С другой стороны, ускорение при извлечении одной строки может быть и стократным. При таком приросте производительности на HBase смело можно вешать этикетку «к работе в веб готова».

HBase и параллелизм

HBase демонстрирует великолепную скорость, когда состав операций ограничивается чтением и записью. Но поскольку StumbleUpon хранил огромный объем данных в базе MySQL, важна также производительность вставки. Для копирования данных в HBase, были написаны задачи Hadoop, содержащие только распределители, которые читают данные из базы MySQL и затем записывают их в HBase. При запуске в кластере с 20 узлами, обеспечивающем примерно 80-кратный параллелизм, совокупная производительность вставки колебалась от 100 000 до 300 000 операций в секунду. При этом длина строки составляла около 100 байтов. Но как бы велика ни была производительность записи, чтение происходит еще быстрее. При 80-кратном параллелизме скорость MapReduce-программы чтения достигала 4,5 миллионов строк в секунду. При таком быстродействии на чтение даже самых больших наших таблиц уходило не больше часа. Раньше у нас не было возможности за один раз произвести анализ всей таблицы.

Все машины были оборудованы двумя четырехъядерными процессорами Intel с объемом памяти 16 ГБ. Каждый узел оснащен дву-

мя дисками SATA, емкостью 1 ТБ каждый. При таком сравнительно скромном стандартном оборудовании достигается очень высокий уровень производительности, а с увеличением количества узлов он только повысится.

12.3.3. Другие применения Hadoop на сайте StumbleUpon

В StumbleUpon мы свято блюдем заповедь «протоколируй рано, протоколируй часто, протоколируй всё». Любые данные, даже самые мелкие и незначительные, когда-нибудь да пригодятся. Hadoop прекрасно проявляет себя в этой традиционной области распределенной обработки: сборе и анализе данных из журналов. StumbleUpon в полной мере использует естественную приспособленность Hadoop к решению разнообразных задач анализа, в том числе агрегирования журналов Apache и анализа пользовательских сеансов.

Например, в наши дни, когда расплодились эксперты по оптимизации поисковых систем и хакеры в «черных шляпах», любой продукт для веб должен уметь анализировать сочетания строк, идентифицирующих пользовательский агент, с (видимым) IP-адресом и контекстом действий. А теперь представьте, что это нужно сделать в условиях, когда имеется огромная ферма веб-серверов, миллионы пользователей и миллиарды переходов по ссылкам.

Scribe[9], открытый проект, первоначально рожденный в Facebook, представляет собой платформу для агрегирования данных из журналов в реальном масштабе времени при наличии подобных условий. Эта служба устойчива к отказам отдельных машин и сети в целом и легко интегрируется практически с любой инфраструктурой.

В StumbleUpon Scribe используется для записи данных непосредственно в HDFS, где они обрабатываются различными системами. Комбинация Cascading с простыми MapReduce-программами для анализа позволяет извлекать из журналов тривиальную статистику (например, счетчики переходов по ссылкам), тогда как потребители с более сложными требованиями могут загружать данные в системы, допускающие обработку запросов в реальном масштабе времени, построенные на базе BerkeleyDB и TokyoCabinet. Другой набор систем использует тот же поток данных для обновления поисковых индексов и генерации миниатюр. На рис. 12.6 показано несколько модулей обработки данных, построенных вокруг Hadoop.

[9] http://github.com/facebook/scribe.

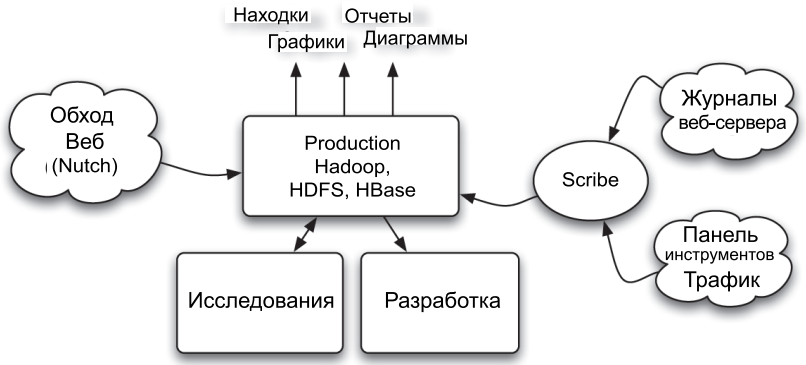


Рис. 12.6. Применение Hadoop для сбора, анализа и сохранения данных на сайте StumbleUpon

Демонстрационные данные были получены в результате обработки стандартных журналов Apache общим объемом 10 ГБ программой анализа на базе Cascading[10]. Используя Hadoop 0.19.1, Cascading 1.0.9 в кластере с вышеупомянутой конфигурацией, мы вычисляли по журналам Apache количество просмотров страниц в минуту, распределяя записи по задачам MapReduce. Мы также написали простенькую программу на Perl с использованием хешей, как пример типичного решения на скорую руку, которое мог бы придумать системный администратор. Приведенные в табл. 12.3 результаты подтверждают линейную (или даже лучшую) масштабируемость при добавлении узлов в кластер. Время усреднено по 10 прогонам, чтобы исключить влияние случайного разброса.

Таблица 12.3. Обработка журналов Apache с помощью Cascading

Система	Показатель	Результат
1 узел	Время работы	21 м 46 с
	Сек/МБ	0,127
	Сек/МБ/Узел	0,127
3 узла	Время работы	8 м 3 с
	Сек/МБ	0,0471
	Сек/МБ/Узел	0,0157
15 узлов	Время работы	1 м 30 с
	Сек/МБ	0,00878
	Сек/МБ/Узел	0,000585

[10] http://code.google.com/p/cascading/wiki/ApacheLogCascade.

Таблица 12.3. (окончание)

Система	Показатель	Результат
Программа на Perl	Время работы	42 м 49 с
	Сек/МБ	0,251
	Сек/МБ/Узел	0,251

Как видим, даже при наличии всего одного узла решение на базе Cascading оказывается вдвое быстрее наивной программы на Perl – за счет того, что интеллектуальный механизм сегментации и группировки, встроенный в каркас MapReduce, эффективнее, чем хранение всех данных в единственном хеше Perl. А познакомившись с Cascading поближе, вы, наверное, придете к выводу, что ко всему прочему написанный на Perl код сложнее оптимизировать (и сопровождать)!

Итак, в StumbleUpon используется встроенная в Hadoop функциональность распределения и редукции, а также тесно связанные с Hadoop продукты, в том числе Nutch, и специально написанные инспекторы содержимого для сбора, анализа и сохранения данных. Размещение результатов в непосредственной близости к конвейеру обработки позволяет получить максимальный выигрыш от локальности данныз.

Подводя итоги, отметим, что сайт StumbleUpon постарался как можно полнее раскрыть всю мощь, заключенную в парадигме MapReduce, используя и расширяя Hadoop, HDFS и HBase. Мы рады и гордимся тем, что помогаем прокладывать дорогу методикам распределенной обработки данных.

12.4. Построение аналитической системы для внутрикорпоративного поиска – проект IBM ES2

Материал любезно предоставили Вук Эрчеговач (Vuk Ercegovac), Раджасекар Кришнамурти (Rajasekar Krishnamurthy), Срирам Рагхаван (Sriram Raghavan), Фредерик Рейс (Frederick Reiss), Юджин Шекита (Eugene Shekita), Сандип Тата (Sandeep Tata), Шивакумар Вайтиянатхан (Shivakumar Vaithyanathan) и Хуа Ю Чжу (Huaiyu Zhu).

В отличие от состоявшихся в последние годы прорывов в области поиска в веб, поиск в корпоративных интрасетях остается трудной и в значительной степени нерешенной проблемой. На основе исследования интрасети IBM Фейджин и др. [1] описали некоторые принципиальные различия между поиском в интрасетях и в веб. Они обнаружили, что подавляющее большинство запросов в интрасети имеют «навигационный» характер. На них существует небольшой набор правильных ответов [2, 3]. Так, ручное изучение 6500 наиболее популярных запросов (по состоянию на июль 2008 года) в интрасети IBM показало, что более 90 процентов из них навигационные.

Поиск «правильного» ответа на такие запросы осложняется несколькими факторами, характерными для корпоративных интрасетей.

- У авторов содержимого отсутствуют экономические стимулы формировать страницу так, чтобы ее было легко найти (напротив, у авторов веб-страниц такой стимул очень силен).
- В поисковых запросах и на самих страницах активно используется внутрикорпоративная лексика, аббревиатуры и акронимы.
- «Правильный» ответ на один и тот же вопрос может зависеть от местоположения и роли в организации пользователя, задающего вопрос (и для корпорации типа IBM, сотрудники и филиалы которой находятся более чем в 80 странах, это особенно важно).

Первые попытки решить описанные проблемы в IBM [4] показали, что это очень трудно сделать, применяя традиционные методы поиска информации. Затем [5] мы предложили подход, в основу которого положен детальный автономный анализ для предварительного выявления навигационных страниц и построения специального навигационного индекса. Мы продемонстрировали жизнеспособность этого подхода, проведя эксперименты с массивом из 5,5 миллионов страниц, взятых из интрасети IBM. Описанная в [5] система была реализована с использованием коммерческих платформ и СУБД различных производителей. С тех пор мы обследовали значительно большую часть интрасети IBM, обнаружили свыше 100 миллионов URL-адресов и проиндексировали более 16 миллионов документов. Для обработки таких и еще больших объемов информации мы, наученные предыдущим опытом [4, 5], разработали ES2 – масштабируемую высококачественную систему поиска в интрасети IBM. ES2

основана на методах анализа, описанных в [5], но построена на базе различных открытых платформ и инструментов, в частности Hadoop, Nutch, Lucene и Jaql[11].

В принципе, встроенный в Nutch робот, каркас Hadoop MapReduce и система индексирования, взятая из Lucene, составляют весь набор программных компонентов, необходимых для построения полноценной поисковой системы. Но чтобы по-настоящему решить описанные выше проблемы, недостаточно просто взять и механически «сшить» эти компоненты. В этой статье мы опишем, как добиться высокого качества поиска с помощью изощренного анализа собранных страниц и специальных навигационных индексов в сочетании с интеллектуальной обработкой запросов. Чтобы стало понятнее, как эти элементы сопрягаются между собой, мы для начала приведем несколько демонстрационных запросов и соответствующие результаты поиска, полученные с помощью ES2 (рис. 12.7).

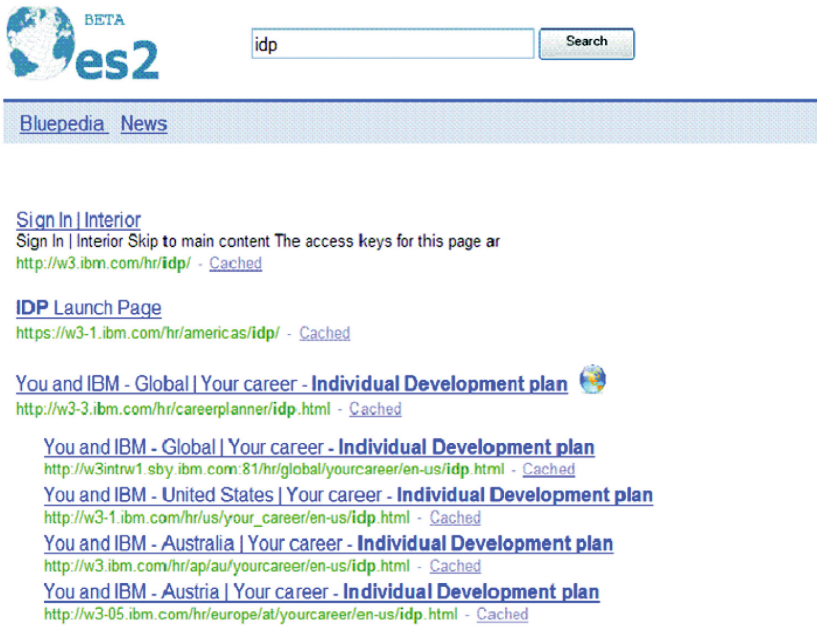


Рис. 12.7. Пример результатов поиска в системе ES2

На рис. 12.7 показан результаты выполнения запроса по ключевому слову *idp*. Этот акроним означает *Individual Development Plan* –раз-

[11] http://code.google.com/p/jaql/.

работанное в IBM веб-приложение для отдела кадров, призванное отслеживать карьерный рост сотрудников. Первые два возвращенных ES2 результата относятся к двум разным URL, каждый из которых позволяет запустить веб-приложение IDP. Третий результат – это фактически набор страниц, описывающих процесс IDP, по одной для каждой страны. Однако они сгруппированы вместе, на что визуально указывает отступ и значок глобуса. Отметим следующие моменты.

1. В первом результате на рис. 12.7 слово «idp» не встречается ни в заголовке, ни даже в тексте страницы. ES2 смогла ассоциировать эту страницу со словом «idp», потому что оно встречается в URL http://w3.ibm.com/hr/idp/. Во втором результате слово «idp» присутствует не только в URL, но и в заглавии, откуда оно выделено путем сопоставления с регулярным выражением, которое ищет заглавия, заканчивающиеся фразами «Launch Page», «Portal», «Main Page» и т. д. В ES2 используется несколько сотен таких тщательно составленных вручную регулярных выражений, применяемых к URL, заглавиям документов, заголовкам META и другим элементам веб-страницы. С их помощью мы обнаруживаем и включаем в индекс термины, ассоциируемые с навигационными страницами. В разделе 12.4.3 описано, как организуется параллельное выполнение такой процедуры, которую мы называем *локальным анализом*, в кластере Hadoop.
2. Наш поисковый робот обнаружил в интрасети IBM примерно 500 страниц, URL-адрес которых содержит слово *idp*, и свыше 1000 страниц, в заглавии которых встречаются слова *idp* или *individual development plan*. Чтобы сузить результирующее множество до двух URL, показанных на рис. 12.7, мы используем набор сложных алгоритмов. Эта процедура называется *глобальным анализом*. Ее реализация в системе ES2 с использованием Hadoop, описана в разделе 12.4.3.
3. Обратите внимание, что во всех ссылках, показанных как часть третьего результата на рис. 12.7, фраза *Individual Development Plan* в заглавии документа выделена, показывая, что она имеет непосредственное отношение к поисковому запросу *idp*. Чтобы установить наличие такой связи, во время автономного анализа потребовалось проделать два шага: (1) на этапе локального анализа фраза *Individual Development Plan* была выделена с помощью регулярных выражений, применен-

ных к заглавию; (2) на этапе индексирования было распознано, что эта фраза является расшифровкой акронима *idp*, поэтому в индекс был добавлен также термин *idp*. В общем случае мы применяем процедуру *генерации вариантов*, смысл которой в том, чтобы найти и добавить в индекс различные варианты терминов, выделенных на этапе локального анализа. В ES2 реализовано несколько стратегий генерации вариантов – от простого разложения выделенной фразы на n-граммы до более хитроумных алгоритмов. Из-за недостатка места мы не будем вдаваться в детали этих алгоритмов.

4. Наконец, чтобы можно было выдавать результаты с учетом географического местоположения пользователя и обеспечить показанную на рис. 12.7 группировку, мы снабдили каждую страницу в интрасети географической меткой (страна, регион и/или местонахождение отделения IBM). В ES2 пометка реализуется с помощью управляемого правилами классификатора, который использует ряд признаков извлеченных из страницы на этапе локального анализа.

Представленные на рис. 12.7 примеры иллюстрируют роль автономного анализа и генерации вариантов в системе ES2. Всякая страница, попадающая на вход ES2, пропускается через несколько логических конвейеров, каждый из которых включает локальный и глобальный анализ и подходящую стратегию генерации вариантов. На выходе конвейера получается некоторое подмножество входных страниц и набор поисковых терминов. В зависимости от применяемых регулярных выражений и правил генерации вариантов, результаты работы разных конвейеров могут отличаться по «точности». Например, конвейер, в котором из заглавия извлекается имя человека, а затем следует настроенная на обработку имен процедура генерации вариантов, вероятно, даст куда более точные ответы, чем конвейер, который просто генерирует все возможные n-граммы по заглавию страницы.

Создание индексной структуры, включающей результаты нескольких конвейеров с различными характеристиками точности, – только половина дела. Чтобы воспользоваться этим индексом, в ES2 применяется сложная стратегия обработки запроса во время выполнения. Обсуждение этого компонента ES2 выходит за рамки данной статьи. Мы сосредоточимся лишь на этапах автономного анализа и их реализации с помощью Hadoop.

12.4.1. Архитектура ES2

Мы предполагаем, что читатели в общих чертах знакомы с Hadoop и Nutch[12]. Nutch – это поисковый робот для веб с открытым исходным кодом, реализованный на платформе Hadoop MapReduce.

В ES2 используется также Jaql[13] – язык описания потоков данных, спроектированный для JSON (популярный способ представления полуструктурированных данных). Jaql позволяет определить стадии обработки полуструктурированных данных в формате JSON с помощью синтаксиса, напоминающего конвейеры в Unix. Конвейер в ES2 сводится к вызову тех или иных алгоритмов локального анализа, глобального анализа и генерации вариантов перед вставкой данных в индексы. Без адекватной поддержки управления данными составление такого сложного многоэтапного конвейера очень быстро превратилось бы в неподъемную проблему. Чтобы справиться с ней, в ES2 данные записываются в формате JSON, а для описания конвейера применяется Jaql (рис. 12.8).

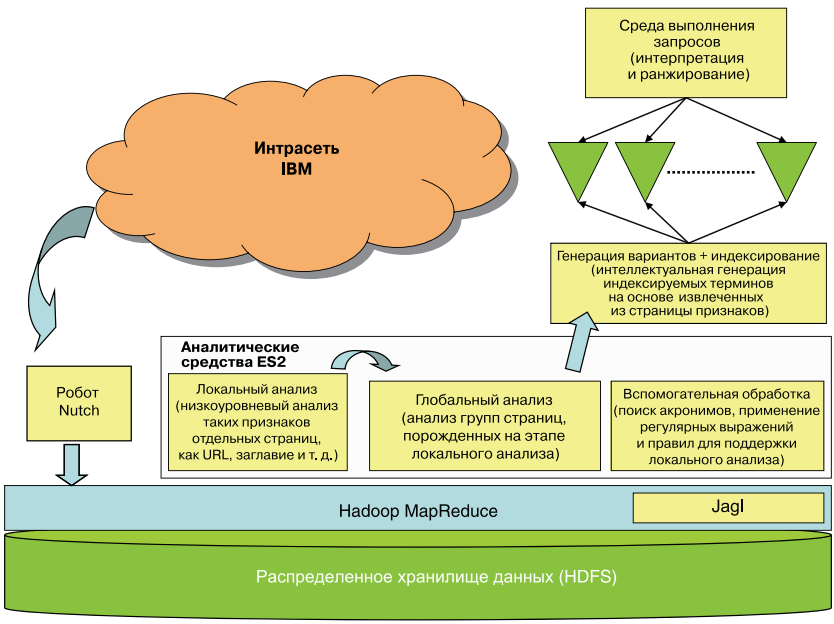


Рис. 12.8. Архитектура ES2

[12] http://nutch.apache.org/.

[13] http://code.google.com/p/jaql/.

На рис. 12.8 изображена архитектура ES2. Система состоит из шести компонентов: робот, локальный анализ, глобальный анализ, генерация вариантов и индексирование, вспомогательные методы обработки данных и среда исполнения поисковых запросов. В ES2 используется доработанная версия Nutch (0.9) – масштабируемый робот с открытым исходным кодом на базе платформы Hadoop. Кроме того, ES2 собирает информацию из службы социальных закладок IBM (под названием Dogear). Эта служба, очень похожая на delicious.com, хранит различные URL-адреса, заложенные членами сообщества IBM, и с каждым адресом ассоциирует набор тегов. Теги несут в себе ценную информацию о содержимом страницы, и ES2 использует ее для построения индексов. На всех этапах применяется единая распределенная файловая система HDFS – как для ввода, так и для вывода. На этапе локального анализа из каждой страницы извлекаются признаки, которые сохраняются в HDFS в виде JSON-объектов. С помощью составленного на Jaql описания ES2 пропускает страницу через оставшуюся часть конвейера, на каждой стадии которого производится то или иное преобразование данных. Jaql-запросы служат для того, чтобы свести воедино результаты различных видов локального анализа и передать их на этап глобального анализа. Jaql используется также для вызова генератора вариантов и индексатора, которым передаются результаты локального и глобального анализа. Индексы периодически копируются на другой набор машин, обслуживающий запросы пользователей.

Кроме того, ES2 периодически выполняет вспомогательные задачи анализа и классификации, не являющиеся частью основного конвейера: автоматическое создание библиотек акронимов, библиотек регулярных выражений [6] и правил географической классификации.

12.4.2. Робот ES2

В ES2 используется Nutch версии 0.9. Основной структурой данных в Nutch является база CrawlDB: множество пар ключ/значение, где ключом является известный Nutch URL-адрес, а значением – его состояние. Состояние содержит различные метаданные о URL-адресе: время его обнаружения, была ли произведена загрузка соответствующего ресурса и т. д. Архитектурно Nutch представляет собой последовательность трех задач MapReduce:

- *Генерация* – на этом этапе генерируется список загрузки, для чего из CrawlDB выбираются все пары ключ/значение, для

которых URL уже обнаружен, но еще не загружен. Обычно в список включаются первые k незагруженных URL, причем порядок определяется подходящим механизмом рейтингования (k – конфигурационный параметр Nutch).

- *Загрузка* – на этом этапе загружаются и разбираются страницы, соответствующие URL-адресам. На выходе получается сам URL и разобранное представление страницы.
- *Обновление* – здесь все URL-адреса, обнаруженные при разборе содержимого загруженных страниц, собираются и включаются в CrawlDB.

Множество страниц, загруженных на каждом цикле генерация-загрузка-обновление, называется *сегментом*. Первая проблема, с которой мы столкнулись, – низкая производительность робота. Nutch обрабатывает в среднем менее трех страниц в секунду – гораздо меньше, чем позволяет пропускная способность сети, доступная кластеру. Но после пробной загрузки 80 миллионов страниц мы обнаружили и другую, более серьезную проблему – удивительно низкое качество найденных страниц. В этом разделе мы расскажем о причинах обеих проблем и опишем изменения, которые были внесены в Nutch для его адаптации к интрасети IBM.

Модификации, направленные на повышение производительности

Nutch проектировался для работы в веб. Направив его на интрасеть IBM, мы обнаружили несколько узких мест. Причина оказалась в том, что количество хостов в интрасети гораздо меньше, чем в веб, а некоторые проектные решения, принятые в Nutch, основаны на предположении о наличии большого числа хостов. Мы опишем два проявления этой проблемы и примененный в ES2 подход к адаптации Nutch для задач внутрикорпоративного поиска.

Главное узкое место возникало на этапе загрузки. Мы назвали этот феномен проблемой длинных хвостов, а суть его заключается в следующем. На начальной стадии этапа загрузки скорость обхода страниц довольно высока (обычно несколько десятков страниц в секунду). Но она сравнительно быстро падает, и до завершения обработки сегмента остается на уровне меньше одной страницы в секунде. Разбирательство показало, что такое поведение определяется хостом с наибольшим числом URL в списке загрузки. И это легко объяснить, если принять во внимание, что скорость загрузки в Nutch контролируется двумя параметрами: количество различных хостов из списка

загрузки, которые Nutch может обходить параллельно, и время между последовательными запросами Nutch к одному хосту. Простейшее решение проблемы длинных хвостов состоит в том, чтобы ограничить количество URL для одного хоста в списке загрузки. К сожалению, этого недостаточно, потому что не все хосты одинаковы и время, необходимое для загрузки одного и того же числа страниц с разных хостов, может сильно различаться. В качестве паллиативного решения мы добавили параметр отсечения, который позволяет прервать работу загрузчика по истечении фиксированного времени. При этом этап загрузки завершается раньше (и, следовательно, в сегменте оказывается меньше страниц), зато не возникает медленных хвостов и сохраняется постоянная высокая скорость обхода. Практические эксперименты показали, что за счет правильного подбора параметра отсечения удается улучшить скорость примерно в три раза. В идеале величина параметра отсечения должна была бы устанавливаться в зависимости от текущей скорости загрузки, но, увы, для этого необходимо как-то обобществлять информацию между распределителями, а в нынешней версии Hadoop сделать это непросто.

Второе узкое место обусловлено структурой данных, которую загрузчик хранит в оперативной памяти. Сначала загрузчик создает множество очередей, в каждой из которых хранятся URL-адреса для одного хоста. Эту структуру мы называем *FetchQueues*. Под нее выделяется область памяти фиксированного размера, в которой размещаются все очереди. Загрузчик читает поданные на вход URL-адреса и помещает их в область FetchQueues, пока не будет исчерпана выделенная память. Рабочие потоки, ассоциированные с каждой очередью в FetchQueues, параллельно загружают страницы с разных хостов, пока не опустошат свои очереди. Проблема возникает из-за того, что входные URL упорядочены по хостам (в таком порядке они поступают с этапа генерации), и загрузчик исчерпывает память, отведенную под FetchQueues, URL-адресами, принадлежащими очень небольшому числу хостов. Такой дизайн вполне пригоден для обхода огромного числа хостов в веб, так как для каждого хоста в списке загрузки будет совсем немного адресов. Но в корпоративной системе число хостов вряд ли превышает несколько тысяч. В результате из области FetchQueues активно читают лишь несколько рабочих потоков, и ресурсы недоиспользуются. Мы решили эту проблему, заменив FetchQueues структурой данных на диске, не имеющей ограничений по размеру. Это позволяет загрузчику поместить в FetchQueues сразу все входные URL, и загрузить работой максимально возможное коли-

чество потоков. После такого простого изменения скорость загрузки возросла в несколько раз.

12.4.3. Аналитические средства в ES2

Наиболее сложная часть системы ES2, благодаря которой она и оказывается столь действенной, – аналитические средства. В этом разделе мы бегло опишем различные алгоритмы, уделяя особое внимание вопросам их реализации в контексте Hadoop.

Локальный анализ

На этапе локального анализа каждая страница анализируется отдельно и из нее извлекается информация, позволяющая решить, является ли она *кандидатом на роль навигационной страницы*. В ES2 имеется пять алгоритмов локального анализа: *TitleHomePage*, *PersonalHomePage*, *URLHomePage*, *AnchorHome* и *NavLink*. В них используются правила, основанные на регулярных выражениях, словарях и инструментах извлечения информации [7]. Например, применяя регулярное выражение вида «`\ A\ W*(.+)\s<Home>`» (в синтаксисе, принятом в Java), алгоритм *PersonalHomePage* может определить, что страница с заглавием «G. J. Chaitin's Home» является домашней страницей G. J. Chaitin. Алгоритм выводит название признака («Personal Home Page») и ассоциирует с этим признаком значение («G. J. Chaitin»). В следующем разделе мы опишем влияние переадресации на локальный анализ и обсудим решение этой проблемы.

Разрешение переадресации

На многих сайтах в интрасети IBM применяется переадресация для обновления, балансирования нагрузки, модернизации и поддержки внутренней реорганизации. К сожалению, переадресация затрудняет работу алгоритмов локального анализа. Например, алгоритм URLHomePage для выявления кандидатов на роль навигационной страницы использует текст. Но после переадресации конечный URL может не содержать тех же признаков, что исходный. В качестве примера рассмотрим URL http://w3.can.ibm.com/hr/erbp. Алгоритмы локального анализа способны распознать, что это URL начальной страницы программы поощрения сотрудников за рекомендации (Employee Referral Bonus Program – ERBP). Однако с этого URL идет переадресация на URL http://w3-03.ibm.com/hr/hrc.nsf/3f31db8c0ff0ac90852568f7006d51ea/ac3f2f04ba60a6d585256d05004cef97?OpenDocument, принадлежащий серверу Lotus Domino,

который и выдает информацию о программе. Конечный URL уже не содержит нужных признаков, поэтому алгоритм локального анализа не считает страницу навигационной. Для предотвращения такого развития событий ES2 самостоятельно разрешает все переадресации, сохраняет множество URL, которые приводят на конечную страницу после переадресации, и подвергает локальному анализу именно их.

Для отслеживания переадресаций мы модифицировали Nutch, так что конечная страница помечается исходным URL. Взгляните на рис. 12.9. Робот проследовал по ссылкам, переадресующим страницу A на страницу B, а страницу B – на страницу C. Мы запомнили эти переадресации, пометив страницы B и C исходным URL страницы A. Метка сохраняется в виде поля метаданных в *сегментном файле*. Сегментный файл представляет собой множество пар ключ/значение, где ключом является URL страницы, а значением – ее содержимое (вместе с дополнительными полями метаданных).

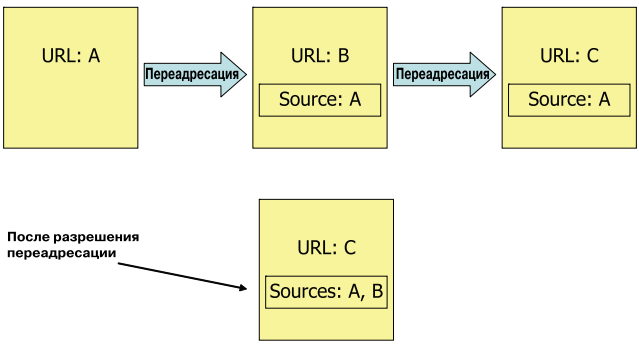


Рис. 12.9. Разрешение переадресации

В листинге 12.2 приведен псевдокод функций распределения и редукции, используемых для разрешения переадресации в сегменте и выполнения локального анализа. Результатом этапа распределения является вывод исходного URL и содержимого страницы. На этапе редукции все страницы с одним и тем же исходным URL объединяются в одну группу. В примере на рис. 12.9 страницы A, B и C имеют общий исходный URL A. Конечная страница этой группы (C) затем передается алгоритмам локального анализа вместе с другими URL из той же группы – A и B.

Листинг 12.2. ResolveSimple

```
Map (Key: URL, Value: PageData)
if PageData.SourceURL exists then
```

```
   Ouput [PageData.SourceURL, PageData]
else
   Output [URL, Pagedata]
end if
End

Reduce (Key: URL, Values: Pageset)
Let URLset = Set of all URLs in Pageset
Let page = Target of redirection in Pageset
result = LocalAnalysis(page, URLset)
output [page.URL, result]
End
```

Реализация в Hadoop

В коде из листинга ResolveSimple локальный анализ выполняется редукторами. Для этого Hadoop должен передавать содержимое каждой страницы с этапа распределения на этап редукции, что приводит к необходимости сортировки и передачи большого объема информации по сети. Чтобы избежать этого, мы модифицируем код ResolveSimple (листинг 12.2), разделив задачи разрешения переадресации и локального анализа, так чтобы алгоритмы локального анализа выполнялись на этапа распределения. Это позволит производить все вычисления там же, где находятся данные, а, значит, добиться значительного повышения производительности.

Псевдокод модифицированного алгоритма приведен в листинге 12.3. На этапе распределения мы передаем только метаданные, исключив содержимое страницы (на долю которого и приходится основной объем данных). На этапе редукции в алгоритме ResolveSimple мы выводим таблицу с двумя столбцами: URL конечной страницы для всей группы страниц и множество URL, которые следует ассоциировать с этой страницей при передаче на стадию локального анализа.

Листинг 12.3. Resolve2Step

```
1: Разрешение переадресации
Map (Key: URL, Value: Page)
if PageData.SourceURL exists then
   Ouput [PageData.SourceURL, PageData.metadata]
else
   Output [URL, Pagedata.metadata]
end if
End

Reduce (Key: URL, Values: Pageset)
Let URLset = Set of all URLs in Pageset
Let page = Target of redirections in Pageset
```

```
output [page.URL, URLset]
End

2: Выполнение локального анализа
Map (Key: URL, Value: Page)
Загрузить resolveTable из выходных данных предыдущего шага, если необходимо
Let URLSet = resolveTable[URL]
result = LocalAnalysis(page, URLset)
output [page.URL, result]
End
```

Если URL переадресуется, то мы не добавляем запись о нем в эту таблицу. В табл. 12.4 приведен пример такой таблицы. Цепочка переадресаций из примера на рис. 12.9, порождает первую строку в таблице 12.4. В следующей задаче без редукторов, в которой производится локальный анализ, распределители считывают таблицу переадресации в память. Для типичных сегментов эта таблица сравнительно мала и легко помещается в памяти. Для каждого URL из входного сегмента распределитель ищет соответствующую ему запись в таблице. Если запись имеется, то хранящиеся в ней URL передаются для локального анализа. Выполняя локальный анализ на этапе распределения, алгоритм Resolve2Step обходится без передачи содержимого страниц редукторам по сети, как в алгоритме ResolveSimple. Мы протестировали оба алгоритма на сегменте, содержащем примерно 400 000 страниц в кластере из восьми узлов. ResolveSimple работал 22 минуты, а Resolve2Step – только 7 минут.

Таблица 12.4. Таблица разрешения переадресации в алгоритме Resolve2Step

URL	Исходные URL
C	{A,B}
…	…
X	{Y}
…	…

Чтобы понять, насколько хорошо масштабируется алгоритм Resolve2Step, мы запускали его для одного и того же сегмента (400 000 страниц) в кластерах с количеством узлов от 1 до 8. Полученные результаты показаны на рис. 12.10. Из графика времени работы видно, что на начальном участке кривой масштабирование линейно, но потом добавление новых узлов дает меньший эффект. Объясняется это тем, что входные данные состоят из единственного сегмента, содержащего 400 000 страниц. Hadoop не может эффективно разбить эту задачу на еще более мелкие части. В следующем разделе мы увидим, что при увеличении входного набора данных Hadoop эффективно разбивает задачу и обеспечивает линейную масштабируемость.

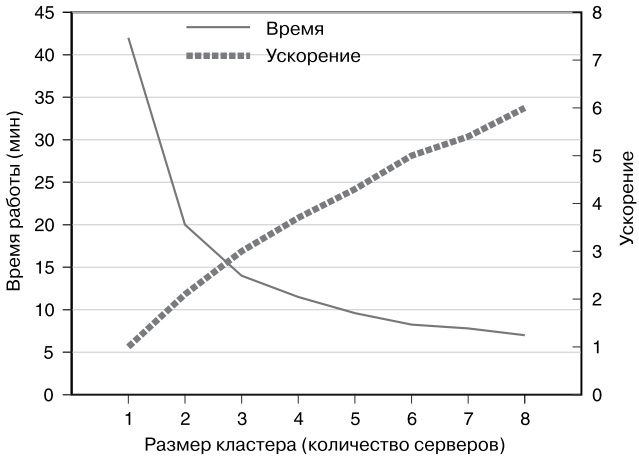


Рис. 12.10. При работе Resolve2Step для одного сегмента (400 000 страниц) наблюдается линейная зависимость производительности от размера кластера (количества узлов в нем).

Глобальный анализ

Описанные в предыдущем разделе задания, выполняемые на стадии локального анализа, находят кандидатов на роль навигационных страниц, извлекая из страницы различные признаки. Но, как описано в разделе 12.4.1, один и тот же навигационный признак может быть ассоциирован с несколькими страницами. Рассмотрим случай, когда автор домашней страницы одинаково озаглавливает многие страницы своего сайта. Например, на сайте G. J. Chaitin многие страницы озаглавлены «G. J. Chaitin home page». Алгоритм локального анализа, выявляющий персональные домашние страницы, сочтет все такие страницы кандидатами. В системе ES2 имеется также стадия глобального анализа, на которой выделяется некоторое подмножество страниц, которые можно рассматривать как навигационные. В работе [5] описано два алгоритма: *анализ корня сайта* и *анализ текста ссылок*. Мы вкратце рассмотрим их и покажем, как их можно реализовать для больших наборов данных с помощью Jaql.

Алгоритмы глобального анализа

На вход каждому заданию, выполняемому на стадии глобального анализа, поступает набор страниц и ассоциированных с ними признаков, сформированный в процессе локального анализа. В листинге 12.4 приведен пример JSON-объекта, соответствующего странице,

озаглавленной «G. J. Chaitin home page». Поля, созданные в процессе локального анализа, находятся в записи «LA», а в процессе глобального анализа – в записи «GA».

Листинг 12.4.
Пример результатов глобального анализа в формате JSON и Jaql-запроса

```
[...,
{docid: 1879495641814943578,
  url: "http://w3.watson.ibm.com/~chaitin/index.html",
  title: "G J Chaitin Home Page",
  ...
  LA: {
    personalHomepage: {name: G J Chaitin, begin: 0, end: 11},
    geography: {countries: "USA", ...}
    ...},
  GA: {
     personalHomepageSiteRootAnalysis: {marked: true, ...},
     ...}
  }, ...]

$alldocs = file "laDocs.json";
$results = file "phpGADocs.json";

$alldocs
-> filter not isnull($.LA.personalHomepage.name)
-> partition by $t = $.LA.personalHomepage.name
    |- SiteRootAnalysis($t, $) -|
-> write $results;
```

Опишем оба алгоритма глобального анализа.

- *Анализ корня сайта* – оба алгоритма служат одной цели: сгруппировать страницы-кандидаты и выделить множество репрезентативных страниц. Сначала набор страниц-кандидатов группируется по интересующему признаку, например PersonalHomePage. Для каждой группы строится лес страниц, в котором каждый URL является вершиной. Две вершины A и B находятся в одном дереве (A – родитель, B – потомок) и A – непосредственный родитель B, если A является самым длинным префиксом B (чем короче префикс, тем выше предок). Затем производится отсечение ветвей с помощью довольно сложной логики, в которой может учитываться информация, полученная другими алгоритмами локального анализа. Детали выходят за рамки настоящей статьи. Анализ корня сайта применяется не только к результатам работы

алгоритма PersonalHomePage, но и алгоритма TitleHomePage (например, страницы, озаглавленный «Working at Almaden Research Center» или «IT Help Central»).

- *Анализ текста ссылок* – этот алгоритм собирает тексты всех ссылок для каждой страницы, исследуя все страницы, указывающие на нее. Собранные данные обрабатываются для выделения репрезентативных терминов, соответствующих данному URL. Детали алгоритма изложены в [5].

Реализация на базе Hadoop

На первом шаге глобального анализа объединяются результаты локального анализа страниц, загруженных роботом, и метки URL, полученные от Dogear. Затем идет шаг дедупликации, на котором устраняются страницы-дубликаты. Далее каждое задание глобального анализа производит стандартные манипуляции с данными (разбиение, фильтрация, соединение) в сочетании с некоторой определенной пользователем функцией, специфичной для задания, например, построение леса и отсечение ветвей. Jaql позволяет сформулировать высокоуровневое описание этих заданий и выполнить их параллельно с помощью Hadoop.

Рассмотрим Jaql-запрос в листинге 12.4, используемый для глобального анализа по признаку PersonalHomePage. В первых двух строчках определяются входной и выходной файл. Предполагается, что на вход подается массив JSON – в данном случае массив записей, каждая из которых представляет страницу и результаты ее локального анализа. В третьей строке начинается конвейер Jaql: обработка страниц из входного файла, на который ссылается переменная *$allDocs*, последующими операторами. Переход от одного оператора конвейера к другому обозначается символом ->. Первым применяется оператор «filter», который порождает значение на выходе, если его предикат возвращает `true`. В нашем примере следующему оператору передаются только страницы, для которых в записи, сформированной на стадии локального анализа (LA), имеется поле PersonalHomePage, а в нем непустое поле name. Переменная *$* ссылается на текущее значение в конвейере. Прошедшие через фильтр страницы разбиваются на группы по полю name. Для каждой группы вычисляется пользовательская функция *SiteRootAnalysis*. Она принимает на входе поле, по которому производится группировка, – *$t* (эта переменная имеет значение *name*) и все попавшие в группу страницы (*$*). Наконец, аннотированные страницы записываются в файл *$results*.

Jaql вычисляет приведенный в листинге 12.4 запрос, транслируя его в задачу MapReduce и передавая ее Hadoop для исполнения. В данном случае на этапе распределения производится фильтрация страниц и извлечение ключа разбиения. На этапе редукции для каждой группы вычисляется функция SiteRootAnalysis и результаты записываются в файл. В общем случае Jaql автоматически транслирует набор определений конвейера в ориентированный ациклический граф задач MapReduce.

На рис. 12.11 и 12.12 показано, как масштабируется стадия глобального анализа на наборе из 16 миллионов документов, когда размер кластера Hadoop изменяется с двух до восьми узлов. На рис. 12.11 показано распределение времени выполнения каждого шага глобального анализа. На рис. 12.12 видно, что при добавлении узлов в кластер общее время, затрачиваемое на глобальный анализ, включая дедупликацию и объединение, пропорционально уменьшается.

Вспомогательная обработка данных

Система ES2 строит библиотеки акронимов, регулярные выражения и правила географической классификации автоматически, обрабатывая собранные роботом данные с помощью вспомогательных процедур добычи данных. Напомним, что все эти ресурсы используются на стадии локального анализа. Периодически, после их обновления, локальный анализ заново запускается для всех страниц. В качестве примера приведем краткое описание алгоритмов выделения акронимов и географической классификации.

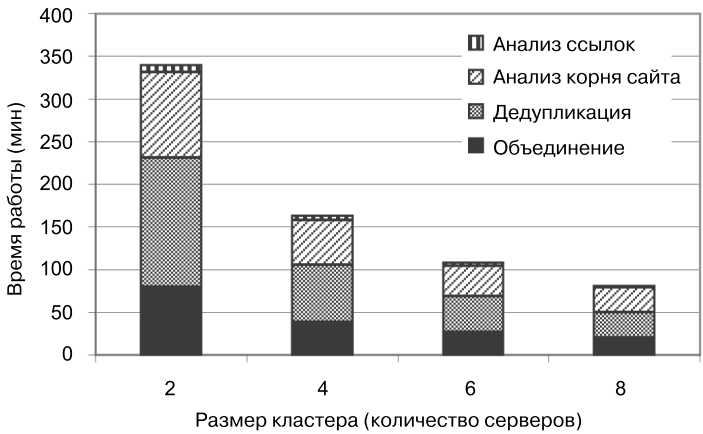


Рис. 12.11. Время глобального анализа

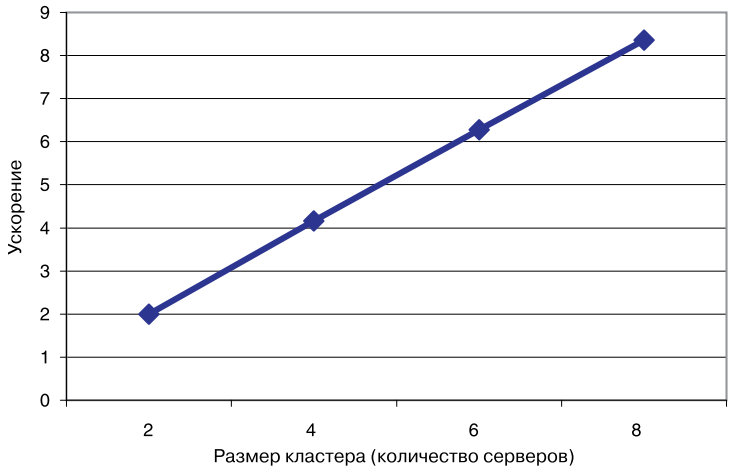


Рис. 12.12. Ускорение глобального анализа

Выделение акронимов – вычислительно сложная задача, выигрыш от распараллеливания ее реализации в кластере Hadoop значителен. В ES2 используется алгоритм, описанный в [8] и несколько доработанный. Сначала текст каждой страницы анализируется на предмет выявления образцов вида «*длиннаяФорма* (*краткаяФорма*)» или «*краткаяФорма* (*длиннаяФорма*)». Выявив акронимы и их возможные *длинныеФормы*, распределитель выводит в качестве ключа и значения пару [краткаяФорма, длиннаяФорма]. Редуктор группирует все возможные *длинныеФормы* для каждой *краткойФормы*, ранжирует их по частоте, а затем выводит. Редуктор объединяет почти идентичные *длинныеФормы*, например «Individual Development Plan» и «Individual Development Plans», ассоциируя их с *краткойФормой* «idp». Псевдокод функций map и reduce для этой задачи приведен в листинге 12.5. Она легко распараллеливается на несколько узлов кластера.

Листинг 12.5. Выделение акронимов

```
Map (Key: URL, Value: PageText)
Найти в тексте все пары вида (краткаяФорма, длиннаяФорма)
Для каждой пары вывести [краткаяФорма, длиннаяФорма]
End

Reduce (Key: shortForm, Values: longForms)
Привести мало отличающие длинныеФормы к каноническому виду
```

```
Вычислить частоты вхождения всех длинныхФорм
Вывести длинныеФормы в отсортированном порядке
End
```

На рис. 12.13 показано время работы этого алгоритма для набора из 10 миллионов документов при увеличении количества узлов кластера с двух до восьми. Как видим, задача работает меньше 25 минут даже при двух узлах. Но с увеличением размера кластера она масштабируется нелинейно. Предполагаемая причина в том, что входные данные представлены в виде нескольких сегментов, и Hadoop решает разбить задачу на много заданий распределения, что влечет за собой значительные фиксированные накладные расходы, не зависящие от размера кластера. Мы исследуем различные способы избавиться от этого падения производительности.

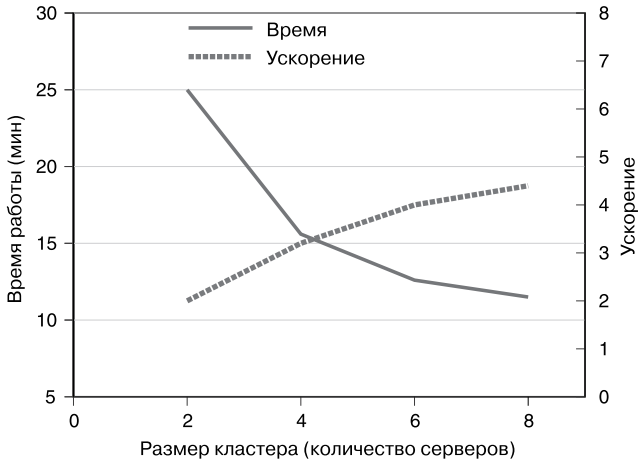


Рис. 12.13. Зависимость времени выделения акронимов от размера кластера

Цель географической классификации – снабдить каждую страницу в интрасети меткой, содержащей страну, регион и/или местоположение отделения IBM, для которых эта страница наиболее релевантна. Например, многие бизнес-процессы и веб-приложения сначала внедряются в США, а лишь затем распространяются на другие страны и регионы. Администраторы сайтов, ответственные за разработку содержимого на позднейших этапах внедрения, часто берут за отправную точку страницу для США, и вносят в нее необходимые изменения. Но при этом они часто забывают изменять

в HTML-коде заголовки META, несущие информацию о локали, языке и других географических особенностях[14]. Поэтому бесхитростный классификатор, обращающий внимание только на заголовки META, будет работать некорректно. В ES2 мы применяем сложный классификатор, управляемый правилами, причем набор правил был тщательно создан вручную и сориентирован на несколько извлекаемых из страницы признаков (присутствие названия страны в заголовке, кода страны в URL и т. д.). Хотя правила и были подобраны вручную для достижения максимальной точности, эффект – число страниц, которым классификатор сумел сопоставить геометки, – незначителен. Для его увеличения следует использовать гораздо больше признаков. Но вручную составлять точные правила для большого набора признаков очень трудоемко. Сейчас мы разрабатываем масштабируемый алгоритм добычи данных, который будет автоматически выводить дополнительные правила классификации по новым признакам, имея уже созданный набор высококачественных правил. Применение такой платформы, как Hadoop, абсолютно необходимо для обеспечения масштабируемости наших алгоритмов добычи данных, применяемых к миллионам страницам с сотнями признаков в каждой.

12.4.4. Выводы

Мы описали архитектуру ES2 – масштабируемой системы внутрикорпоративного поиска, которая разработана в IBM с использованием открытых компонентов: Nutch, Hadoop, Lucene и Jaql. Мы также рассказали об изменениях, которые пришлось внести в робот Nutch для обхода корпоративной интрасети. Мы переработали для Hadoop алгоритмы локального и глобального анализа, описанные в [5]. Мы обнаружили, что для реализации сложного конвейера, в который входят задачи обхода сети, локального анализа, глобального анализа и индексирования, JSON оказывается удобным форматом данных, а Jaql исключительно действенным средством. Короче говоря, мы считаем, что сочетание Hadoop, Nutch, Lucene и Jaql представляет собой мощный инструментарий, позволяющий конструировать такие сложные и масштабируемые системы, как ES2.

---

[14] Отметим, что заголовки META предназначены для браузеров и роботов; при визуализации страницы они не видны.

12.4.5. Библиография

1 Fagin, R., R. Kumar, K. S. McCurley, J. Novak, D. Sivakumar, J. A. Tomlin, and D. P. Williamson. "Searching the workplace web." In *WWW*, pages 366–375, 2003.

2 Broder, A. "A taxonomy of web search." *SIGIR Forum*, 36(2):3–10, 2002.

3 Hawking, D. "Challenges in enterprise search." In *ADC*, pages 15–24, 2004.

4 Fontura, M., E. J. Shekita, J. Y. Zien, S. Rajagopalan, and A. Neumann. "High performance index build algorithms for intranet search engines." In *VLDB*, pages 1158–1169, 2004.

5 Zhu, H., S. Raghavan, S. Vaithyanathan, and A. Lцser. "Navigating the intranet with high precision." In *WWW*, pages 491–500, 2007.

6 Li, Y., R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. "Regular expression learning for information extraction." In *EMNLP*, 2008.

7 Reiss, F., S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. "An algebraic approach to rule-based information extraction." In *ICDE*, pages 933–942, 2008.

8 Schwartz, A. S., and M. A. Hearst. "A simple algorithm for identifying abbreviation definitions in biomedical text." In *Pacific Symposium on Biocomputing*, pages 451–462, 2003.

ПРИЛОЖЕНИЕ. Команды HDFS

В этом приложении перечислены команды HDFS для работы с файлами. Каждая команда имеет вид:

```
hadoop fs -cmd <args>
```

где `cmd` — имя команды, а `<args>` — переменное число аргументов. Параметры в квадратных скобках необязательны, а многоточие (`...`) означает, что необязательный параметр может повторяться несколько раз. Словом FILE обозначаются имена файлов, а словом PATH — пути к файлам или каталогам. SRC и DST — пути, играющие роль источника и места назначения соответственно. LOCALSRC и LOCALDST употребляются в качестве путей в локальной файловой системе.

Команда	Описание и порядок вызова
`cat`	`hadoop fs –cat FILE [FILE ...]` Выводит содержимое файлов. Для чтения сжатых файлов используйте команду `text`.
`chgrp`	`hadoop fs –chgrp [-R] GROUP PATH [PATH ...]` Изменяет группу для файлов и каталогов. Флаг `–R` означает, что применять команду следует рекурсивно. Выполнять команду имеет право владелец файла или суперпользователь О системе разрешений в HDFS см. раздел 8.3.
`chmod`	`hadoop fs –chmod [-R] MODE[,MODE ...] PATH [PATH ...]` Изменяет права доступа к файлам и каталогам. По аналогии с одноименной командой Unix `MODE` — трехзначный восьмеричный режим или `{augo}+/-{rwxX}`. Флаг `–R` означает, что применять команду следует рекурсивно. Выполнять команду имеет право владелец файла или суперпользователь. О системе разрешений в HDFS см. раздел 8.3.

Команда	Описание и порядок вызова
chown	`hadoop fs –chown [-R] [OWNER][:[GROUP]] PATH [PATH ...]` Изменяет владельца файлов или каталогов. Флаг `–R` означает, что применять команду следует рекурсивно. Выполнять команду имеет право владелец файла или суперпользователь. О системе разрешений в HDFS см. раздел 8.3.
copyFromLocal	`hadoop fs –copyFromLocal LOCALSRC [LOCALSRC ...] DST` То же, что `put` (копирует файлы из локальной файловой системы).
copyToLocal	`hadoop fs –copyToLocal [-ignorecrc] [-crc] SRC [SRC ...] LOCALDST` То же, что `get` (копирует файлы в локальную файловую систему).
count	`hadoop fs –count [-q] PATH [PATH ...]` Выводит количество подкаталогов, файлов, размер в байтах и имена для всех файлов и каталогов, заданных аргументом `PATH`. Если задан флаг `–q`, выводится информация о квотах.
cp	`hadoop fs –cp SRC [SRC ...] DST` Копирует файлы из одного места в другое. Если задано несколько входных файлов, то местом назначения должен быть каталог.
du	`hadoop fs –du PATH [PATH ...]` Выводит размеры файлов. Если `PATH` — каталог, то выводятся размеры всех находящихся в нем файлов. Именам файлов предшествует схема URI. Хотя `du` означает «disk usage», не следует понимать это буквально, поскольку реально занятое на диске место зависит от размера блока и коэффициента репликации.
dus	`hadoop fs –dus PATH [PATH ...]` Аналогична `du`, но для каталога выводит суммарный размер файлов в нем, а не размер каждого отдельного файла.
expunge	`hadoop fs –expunge` Очищает корзину. Если механизм корзины включен, то удаленные файлы сначала перемещаются в каталог .Trash/. Из этого каталога файл автоматически удаляется спустя заданное в конфигурационном файле время. Команда `expunge` принудительно удаляет файлы из каталога .Trash/. Пока файл находится в .Trash/, его можно восстановить, то есть вернуть на прежнее место.

Команда	Описание и порядок вызова
get	`hadoop fs –get [-ignorecrc] [-crc] SRC [SRC ...] LOCALDST` Копирует файлы в локальную файловую систему. Если задано несколько входных файлов, то местом назначения должен быть каталог. Если в качестве `LOCALDST` указан -, то файлы копируются в stdout. HDFS вычисляет контрольную сумму каждого блока каждого файла. Контрольные суммы хранятся отдельно, в скрытом файле. При чтении файла из HDFS, контрольные суммы используются для проверки целостности файла. Команда `get` с флагом `–crc` копирует скрытый файл с контрольными суммами. Если задан флаг `–ignorecrc`, то проверка контрольной суммы при копировании не производится.
getmerge	`hadoop fs –getmerge SRC [SRC ...] LOCALDST [addnl]` Выбирает все файлы, заданные в аргументах `SRC`, объединяет их и записывает в один файл `LOCALDST` в локальной файловой системе. Если задан параметр `addnl`, то в конец каждого файла добавляется символ новой строки.
help	`hadoop fs –help [CMD]` Выводит сведения о порядке вызова команды `CMD`. Если параметр `CMD` опущен, выводит справку по всем командам.
ls	`hadoop fs –ls PATH [PATH ...]` Выводит список файлов и каталогов. Для каждого элемента выводится имя, разрешения, владелец, группа, размер и дата последнего изменения. Для файлов выводится также коэффициент репликации.
lsr	`hadoop fs –lsr PATH [PATH ...]` Рекурсивный вариант команды `ls`.
mkdir	`hadoop fs –mkdir PATH [PATH ...]` Создает каталоги. Отсутствующие родительские каталоги также создаются (по аналогии с командой Unix `mkdir –p`).
moveFromLocal	`hadoop fs –moveFromLocal LOCALSRC [LOCALSRC ...] DST` Аналогична `put`, но локальный исходный файл удаляется после успешного копирования в HDFS.
moveToLocal	`hadoop fs –moveToLocal [-crc] SRC [SRC ...] LOCALDST` Выводит сообщение «not implemented yet» (пока не реализована).

Команда	Описание и порядок вызова
mv	`hadoop fs –mv SRC [SRC ...] DST` Перемещает файлы из одного места в другое. Если задано несколько входных файлов, то местом назначения должен быть каталог. Перемещать файлы между файловыми системами запрещено.
put	`hadoop fs –put LOCALSRC [LOCALSRC ...] DST` Копирует файлы или каталоги из локальной файловой системы в место назначения. Если в качестве `LOCALSRC` указан -, то входом является `stdin`, а `DST` должен быть файлом.
rm	`hadoop fs –rm PATH [PATH ...]` Удаляет файлы и пустые каталоги.
rmr	`hadoop fs –rmr PATH [PATH ...]` Рекурсивный вариант команды `rm`.
setrep	`hadoop fs –setrep [-R] [-w] REP PATH [PATH ...]` Задает для указанных файлов коэффициент репликации `REP`. Если задан флаг `-R`, то команда рекурсивно применяется к файлам, находящимся внутри каталогов `PATH`. Для выхода на заданный коэффициент требуется некоторое время. Если задан флаг `–w`, то команда не завершается, пока коэффициент репликации для всех указанных файлов не станет равным заданному.
stat	`hadoop fs –stat [FORMAT] PATH [PATH ...]` Выводит «статистическую» информацию о файлах. В форматной строке `FORMAT` распознаются и заменяются соответствующими данными следующие спецификаторы: `%b` – размер файла в блоках; `%F` – строка «directory» или «regular file» в зависимости от типа файла; `%n` – имя файла; `%o` – размер блока; `%r` – коэффициент репликации; `%y` – дата UTC в формате yyyy-MM-dd HH:mm:ss; `%Y` – количество миллисекунд с 1 января 1970 года UTC. Все прочие символы форматной строки выводятся буквально.
tail	`hadoop fs –tail [-f] FILE` Выводит последний килобайт файла `FILE`.

Команда	Описание и порядок вызова
`test`	`hadoop fs –test –[ezd] PATH` Выполняет одну из следующих проверок для пути `PATH`: `-e` – существование. Возвращает 0, если путь PATH существует. `-z` – пустой файл. Возвращает 0, если длина файла равна 0. `-d` – возвращает 0, если `PATH` – каталог.
`text`	`hadoop fs –text FILE [FILE ...]` Выводит текстовое содержимое файлов. То же, что `cat`, если файлы текстовые. Файлы в известном сжатом формате (gzip и двоичный формат файла последовательности, применяемый в Hadoop) сначала распаковываются.
`touchz`	`hadoop fs –touchz FILE [FILE ...]` Создает файлы нулевой длины. Завершается с ошибкой, если указанные файлы существуют и имеют ненулевую длину.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Д

Е

Ж

З

И

К

Л

М

Н

О

П

P

C

Т

У

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**;

Электронный адрес **books@alians-kniga.ru**.

Чак Лэм

Hadoop в действии

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 02.10.2011. Формат 60×90 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 28,42. Тираж 500 экз.
заказ №

Web-сайт издательства: www.dmk-press.ru