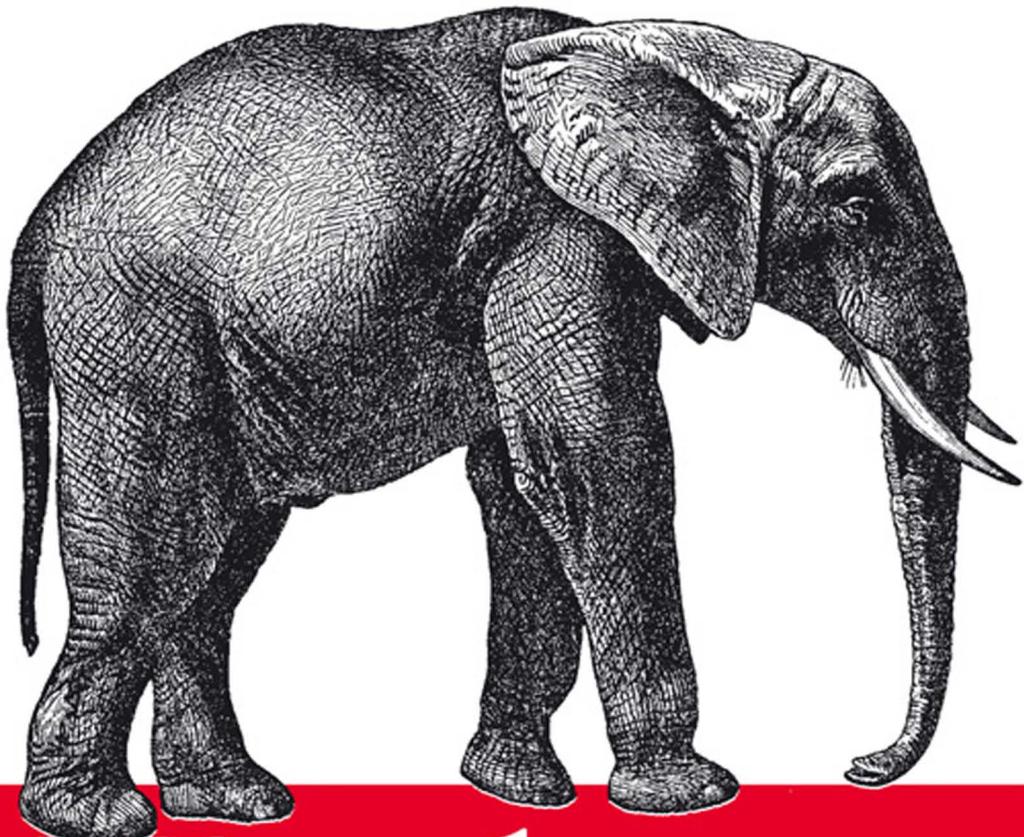


Хранение и Анализ Интернет-масштаба



Надооп

Подробное руководство

Том Уайт

O'REILLY®

 ПИТЕР®

Tom White

Hadoop: **The Definitive Guide**

THIRD EDITION

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Том Уайт

Hadoop

Подробное руководство



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.988.02

УДК 004.438.5

У13

Яйт Т.

У13 Hadoop: Подробное руководство. — СПб.: Питер, 2013. — 672 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-00662-0

Apache Hadoop — фреймворк с открытым исходным кодом, в котором реализована вычислительная парадигма, известная как MapReduce, позволившая Google построить свою империю. Эта книга покажет вам, как использовать всю мощь Hadoop, чтобы создавать надежные, масштабируемые, распределенные системы и обрабатывать гигантские наборы данных. Программисты найдут здесь методики анализа, администраторы узнают, как установить и запустить кластеры Hadoop. Если вы работаете с большими массивами данных, гигабайтами или петабайтами информации, то Hadoop — это идеальное решение. «Hadoop: Подробное руководство» — книга, в которой досконально и доступно описаны все возможности Apache Hadoop. Издание охватывает последние изменения Hadoop, в том числе материалы по новой исполнительной среде MapReduce, называемой MapReduce 2, которая реализована на базе системы YARN (Yet Another Resource Negotiator) — общей системы управления ресурсами для распределенных приложений.

12+ (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02

УДК 004.438.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1449311520 англ.

© 2013, Piter Inter Ltd. Authorized Russian translation of the English edition of Hadoop: The Definitive Guide, 3rd Edition (ISBN 9781449311520) © 2012 Tom White. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-496-00662-0

© Перевод на русский язык ООО Издательство «Питер», 2013
© Издание на русском языке, оформление ООО Издательство «Питер», 2013

Краткое содержание

Предисловие	19
Введение	21
Глава 1. Знакомство с Hadoop	27
Глава 2. MapReduce	47
Глава 3. HDFS	79
Глава 4. Ввод/вывод в Hadoop	125
Глава 5. Разработка приложений MapReduce	202
Глава 6. Как работает MapReduce	256

Глава 7. Типы и форматы MapReduce	296
Глава 8. Дополнительные возможности MapReduce	337
Глава 9. Создание кластера Hadoop	384
Глава 10. Администрирование Hadoop	435
Глава 11. Pig	467
Глава 12. Hive	523
Глава 13. HBase	577
Глава 14. ZooKeeper	607
Глава 15. Sqoop	647

Содержание

Предисловие	19
 Введение	21
Замечания	22
О чем рассказано в книге?	23
Что нового во втором издании?	23
Что нового в третьем издании?	23
Использование примеров кода	24
Благодарности	25
От издательства	26
 Глава 1. Знакомство с Hadoop	27
Данные!	27
Хранение и анализ данных	29
Сравнение с другими системами	30

Hadoop и РСУБД	31
Распределенные вычисления	34
Добровольные вычисления	35
Краткая история Hadoop.....	36
Apache Hadoop и экосистема Hadoop	41
Выпуски Hadoop.....	42
О чем рассказано в книге.....	44
Имена конфигураций	44
MapReduce API	45
Совместимость	45
Глава 2. MapReduce.....	47
Набор метеорологических данных.....	47
Формат данных	48
Анализ данных средствами Unix	49
Анализ данных в Hadoop	51
Отображение и свертка.....	51
Программа MapReduce на языке Java	53
Тестовый запуск.....	57
MapReduce в перспективе	63
Поток данных.....	64
Комбинирующие функции.....	68
Определение комбинирующей функции	69
Выполнение распределенного задания MapReduce	70
Hadoop Streaming	71
Ruby	71
Python	74
Hadoop Pipes	75
Компилирование и запуск.....	77
Глава 3. HDFS	79
Строение HDFS	80
Основные концепции HDFS	81
Блоки	81
Узлы имен и узлы данных	83

HDFS Federation	84
Высокая доступность HDFS	85
Преодоление сбоев и изоляция	86
Интерфейс командной строки.....	87
Основные операции файловой системы	88
Файловые системы Hadoop	90
Интерфейсы	93
Интерфейс Java	95
Чтение данных Hadoop по URL-адресу	95
Чтение данных с использованием Filesystem API	97
Запись данных	100
Получение информации от файловой системы	103
Удаление данных	109
Поток данных	109
Чтение файла	109
Запись в файлы	113
Модель целостности	116
Перемещение данных: Flume и Sqoop	118
Параллельное копирование с использованием distcp	118
Сбалансированность кластеров HDFS	121
HAR	121
Использование HAR	121
Ограничения	123
Глава 4. Ввод/вывод в Hadoop	125
Целостность данных	125
Целостность данных в HDFS	126
LocalFileSystem	127
ChecksumFileSystem	128
Сжатие	128
Кодеки	130
Сжатие и разбиение входных данных	135
Использование сжатия в MapReduce	137
Сериализация	140
Интерфейс Writable	141

Классы Writable	144
Пользовательские реализации Writable	153
Программные среды сериализации	158
 Avro.	161
Типы данных и схемы Avro.	162
Сериализация и десериализация в памяти	166
Файлы данных Avro	170
Совместимость	172
Преобразование схемы	175
Порядок сортировки.	177
Avro и MapReduce	179
Сортировка с использованием Avro MapReduce	183
Avro MapReduce в других языках.....	186
Файловые структуры данных.	186
SequenceFile.....	186
Запись SequenceFile.	187
Чтение из SequenceFile	189
MapFile.....	195
 Глава 5. Разработка приложений MapReduce	202
API конфигурации	203
Объединение ресурсов.....	204
Расширение переменных	205
Настройка среды разработки	206
Управление конфигурацией	208
GenericOptionsParser, Tool и ToolRunner	211
Написание модульных тестов с MRUnit	215
Функция отображения.	215
Функция свертки.	218
Локальное выполнение с тестовыми данными	219
Локальный запуск задания	219
Тестирование управляющей программы.	223
Запуск в кластере.	225
Упаковка задания	225
Запуск задания	227

Веб-интерфейс MapReduce	229
Получение результатов	232
Отладка задания	235
Журналы Hadoop	240
Удаленная отладка	242
Оптимизация задания	243
Профилирование	244
Модель MapReduce	247
Разложение задачи на задания MapReduce	248
JobControl	249
Apache Oozie	250
Определение потока операций Oozie	251
Глава 6. Как работает MapReduce	256
Выполнение задания MapReduce	256
Классическая реализация MapReduce (MapReduce 1)	257
Отправка заданий	258
YARN (MapReduce 2)	265
Сбои	271
Сбои в классической модели MapReduce	272
Сбои в YARN	274
Планирование заданий	277
Fair Scheduler	277
Capacity Scheduler	278
Тасовка и сортировка	279
На стороне отображения	279
На стороне свертки	281
Настройка конфигурации	283
Выполнение задач	287
Среда выполнения задач	287
Спекулятивное выполнение	288
OutputCommitter	290
Файлы побочных эффектов	292
Повторное использование JVM задач	292
Пропуск некорректных записей	293

Глава 7. Типы и форматы MapReduce	296
Типы MapReduce	296
Задание MapReduce по умолчанию	299
Форматы входных данных	309
Входные сплиты и записи	309
FileInputFormat	311
Входные пути FileInputFormat	311
Текстовые входные данные.	322
Двоичные входные данные	326
Множественные источники входных данных	328
Операции ввода (и вывода) с базами данных	329
Форматы выходных данных	329
Текстовые выходные данные	330
Двоичные выходные данные	330
Множественный вывод	331
Отложенный вывод	336
Вывод в базы данных	336
Глава 8. Дополнительные возможности MapReduce	337
Счетчики	337
Встроенные счетчики	338
Счетчики Java, определяемые пользователем	344
Пользовательские счетчики в Streaming	349
Сортировка	350
Подготовка	350
Частичная сортировка	351
Полная сортировка	357
Вторичная сортировка	361
Соединения	368
Соединения на стороне отображения	369
Соединения на стороне свертки	370
Распространение побочных данных	374
Использование конфигурации задания	375
Распределенный кэш	375
Библиотечные классы MapReduce	383

Глава 9. Создание кластера Hadoop	384
Оборудование кластера	384
Сетевая топология	387
Настройка и установка кластера	389
Установка Java	389
Создание пользователя Hadoop	390
Установка Hadoop	390
Тестирование установки	391
Конфигурация SSH	391
Конфигурация Hadoop	392
Управление конфигурацией	393
Настройки окружения	396
Важные свойства демонов Hadoop	401
Адреса и порты демонов Hadoop	407
Другие свойства Hadoop	408
Создание учетных записей пользователей	412
Конфигурация YARN	412
Важные свойства демонов YARN	413
Адреса и порты демонов YARN	417
Безопасность	419
Kerberos и Hadoop	420
Маркеры делегирования	423
Другие улучшения в области безопасности	424
Тестирование кластера Hadoop	426
Пользовательские задания	429
Hadoop в облаке	429
Apache Whirr	430
Глава 10. Администрирование Hadoop	435
HDFS	435
Дисковые структуры данных	435
Безопасный режим	441
Журналы аудита	443
Инструменты	444

Мониторинг	450
Ведение журналов	450
Метрики	451
Сопровождение	458
Стандартные административные процедуры	458
Включение и исключение узлов.	459
Обновления	463
Глава 11. Pig	467
Установка и запуск Pig	469
Режимы исполнения	469
Запуск программ Pig	471
Grunt	471
Редакторы Pig Latin	472
Пример	472
Генерирование примеров	475
Сравнение с базами данных	477
Pig Latin	478
Структура	478
Инструкции	479
Выражение	485
Типы	487
Схемы	489
Функции	494
Макросы	496
Пользовательские функции	498
Фильтрующая пользовательская функция	498
Вычисляющая пользовательская функция	502
Пользовательская функция загрузки	504
Операторы обработки данных	508
Загрузка и сохранение	508
Фильтрация данных	508
Группировка и соединение данных	512
Сортировка данных	518
Комбинирование и разбиение данных	518
Практическое использование Pig	519

Параллелизм	519
Подстановка параметров	520
Глава 12. Hive	523
Установка Hive	524
Оболочка Hive	525
Пример	526
Администрирование Hive	528
Настройка конфигурации Hive	528
Сервисные функции Hive	530
Метахранилище	533
Сравнение с традиционными базами данных	535
Проверка схемы при чтении и записи	536
Обновления, транзакции и индексы	536
HiveQL	537
Типы данных	539
Операторы и функции	542
Таблицы	543
Управляемые и внешние таблицы	543
Разделы и гнезда	545
Форматы хранения данных	550
Импортирование данных	557
Модификация таблиц	559
Удаление таблиц	560
Запросы к данным	560
Сортировка и агрегирование	560
Сценарии MapReduce	561
Подзапросы	566
Пользовательские функции	568
Написание пользовательской функции	570
Написание UDAF	572
Глава 13. HBase	577
Знакомство с HBase	577
История	578

Концепции	578
Краткий обзор модели данных	578
Реализация	580
Установка	583
Пробный запуск	584
Клиенты	586
Java	586
Avro, REST и Thrift	591
Пример	592
Схемы	592
Загрузка данных	593
Веб-запросы	597
HBase и РСУБД	600
Масштабирование успешного сервиса	601
HBase	603
Пример из практики: HBase в Streamy.com	603
Переход на HBase	605
Глава 14. ZooKeeper	607
Установка и запуск ZooKeeper	609
Пример	611
Реализация списка принадлежности в ZooKeeper	611
Создание группы	612
Присоединение к группе	614
Вывод списка участников группы	616
Удаление группы	618
Сервис ZooKeeper	619
Модель данных	619
Операции	622
Реализация	627
Согласованность данных	628
Сеансы	630
Состояния	632
Построение приложений с использованием ZooKeeper	633
Конфигурация	633
Отказоустойчивое приложение ZooKeeper	637

Блокировка	641
Другие распределенные структуры данных и протоколы	643
Практическое использование ZooKeeper	644
Надежность и производительность	644
Конфигурация	645
Глава 15. Sqoop	647
Установка и запуск Sqoop	647
Коннекторы Sqoop	649
Пример импортирования	649
Текстовые и двоичные форматы	652
Сгенерированный код	653
Другие системы сериализации	653
Подробнее об импортировании	654
Управление импортированием	656
Импортирование и согласованность данных	656
Прямое импортирование	657
Работа с импортированными данными	657
Импортирование данных в Hive	658
Импортирование больших объектов	661
Экспортирование	663
Подробнее об экспортировании	665
Экспортирование и транзакционность	666
Экспортирование в SequenceFile	667

Предисловие

Hadoop начинался с проекта Nutch. Группа разработчиков попыталась построить систему веб-поиска с открытым кодом, однако проблемы с обработкой данных стали возникать даже на относительно небольшом множестве компьютеров. После того как компания Google опубликовала статьи о GFS и MapReduce, направление работы прояснилось. Системы Google разрабатывались для решения именно тех проблем, которые у нас возникали с Nutch. И тогда мы — два разработчика — начали в свободное время работать над воссозданием этих систем как составной части Nutch.

Мы кое-как заставили Nutch работать на 20 машинах, но вскоре стало ясно, что в огромных масштабах Web речь идет о выполнении на тысячах компьютеров, и что еще важнее — объем работы был слишком большим для двух программистов с неполным рабочим временем.

Примерно в это же время компания Yahoo! заинтересовалась проектом и быстро собрала группу, к которой присоединился я. Мы выделили часть Nutch, относящуюся к распределенным вычислениям, в отдельный подпроект, который получил название «Hadoop». При поддержке Yahoo! проект Hadoop вскоре превратился в технологию, действительно способную работать в масштабах Web.

В 2006 году к работе над Hadoop присоединился Том Уайт. Я уже был знаком с Томом по отличной статье, которую он написал о Nutch, и знал, что он умеет четко и ясно излагать сложные идеи. Вскоре выяснилось, что его программный код читается так же легко, как его тексты.

С самого начала вклад Тома в Hadoop отражал его заботу о пользователях и проекте. В отличие от многих участников проектов с открытым кодом, Том стремился не подстроить систему под свои собственные потребности, а прежде всего упростить ее использование всеми остальными.

Сначала Том специализировался на обеспечении нормальной работы Hadoop на серверах Amazon EC2 и S3. В дальнейшем он занимался разнообразными задачами, включая усовершенствование MapReduce API, доработку нашего веб-сайта и проектирование инфраструктуры сериализации объектов. При этом Том представлял свои идеи с неизменной точностью. Вскоре Том получил роль ответственного участника (committer), а по прошествии некоторого времени вошел в комитет по управлению проектом Hadoop.

Сейчас Том является авторитетным руководителем сообщества разработчиков Hadoop. Хотя он превосходно разбирается во многих технических тонкостях проекта, его основная специальность — делать Hadoop простым для понимания и использования.

Поэтому я очень обрадовался, когда узнал, что Том собирается написать книгу о Hadoop. Разве кто-нибудь справится с этим делом лучше него? Теперь вы можете узнать о Hadoop от настоящего мастера — блестяще владеющего не только технологией, но и здравым смыслом и умением объяснять.

*Дуг Каттинг
Калифорния*

Введение

Мартин Гарднер, автор многих книг по математике и популярной науке, однажды сказал в интервью:

«Я не чувствую себя уверенно ни в чем, кроме вычислений. И в этом кроется секрет успеха моей рубрики. Я так долго разбирался в том, о чем пишу, что в итоге я понимаю, как сделать свои тексты понятными для большинства читателей»¹.

Во многих отношениях я точно так же отношусь к Hadoop. Внутренние механизмы Hadoop сложны; они основываются на сочетании теории распределенных систем, инженерной практике и здравом смысле. И непосвященному Hadoop кажется чем-то недоступным для понимания.

Но это совершенно необязательно. В сущности, инструменты, предоставляемые Hadoop для построения распределенных систем (хранение данных, анализ данных и координация), достаточно просты. Причем если между ними и существует сходство, то это повышение уровня абстракции с целью создания структурных элементов для программистов, которым требуется хранить или анализировать большие объемы данных или координировать работу множества машин, но которые не имеют времени, квалификации или желания становиться экспертами в области

¹ «The science of fun», Alex Bellos, The Guardian, May 31, 2008, <http://www.guardian.co.uk/science/2008/may/31/mathscience>.

распределенных систем для самостоятельного построения инфраструктуры, решающей эти задачи.

Когда я только начал использовать Hadoop, мне казалось совершенно очевидным, что с такими простыми и универсальными возможностями Hadoop заслуживает широкого применения. Однако в то время (в начале 2006 года) настройка и написание программ, использующих Hadoop, оставались искусством. Бессспорно, ситуация с того времени улучшилась: стало больше документации, стало больше примеров, появились рассылки, в которые можно обратиться с вопросами. И все же самая большая проблема у новичков — понять, на что способна эта технология, каковы ее преимущества и как ее использовать. Вот почему я написал эту книгу.

Сообщество Apache Hadoop прошло долгий путь. За три года проект Hadoop расцвел и породил с полдюжины подпроектов. В это время были достигнуты качественные улучшения производительности, надежности, масштабируемости и управляемости продукта. Тем не менее я считаю, что для еще более широкого распространения Hadoop необходимо сделать еще проще в использовании. Для этого придется написать новые инструменты, обеспечить интеграцию с новыми системами и разработать новые, улучшенные API. Я с нетерпением жду возможности поучаствовать в этой работе и надеюсь, что эта книга поможет подключиться к ней другим разработчикам.

Замечания

При обсуждении конкретного класса Java я часто опускаю имя пакета, чтобы не загромождать текст. Если вам потребуется узнать, в каком пакете находится класс, обратитесь к документации Hadoop по Java API для соответствующего проекта; ссылка на нее находится на домашней странице Apache Hadoop по адресу <http://hadoop.apache.org/>. Если вы работаете в интегрированной среде разработки (IDE), воспользуйтесь механизмом автозаполнения.

Хотя это и не соответствует обычным стилем рекомендациям, в листингах программ, импортирующих несколько классов из одного пакета, для экономии места может использоваться универсальный символ * (например, `import org.apache.hadoop.io.*`).

Примеры программ, приведенные в книге, можно загрузить на веб-сайте книги по адресу <http://www.hadoopbook.com/>. Там же находятся инструкции по получению наборов данных, используемых в примерах, дополнительные комментарии по поводу запуска приведенных программ, ссылки на обновления, дополнительные ресурсы и мой блог.

О чём рассказано в книге?

В главе 1 объясняется необходимость создания Hadoop и описывается история проекта. Глава 2 знакомит читателя с MapReduce. В главе 3 подробно рассматриваются файловые системы Hadoop (прежде всего HDFS). В главе 4 изложены основные принципы ввода/вывода в Hadoop: целостность данных, сжатие, сериализация и структуры данных на базе файлов.

Следующие четыре главы содержат подробное описание MapReduce. В главе 5 перечислены практические действия, необходимые для разработки приложений MapReduce. В главе 6 рассматривается реализация MapReduce в Hadoop с точки зрения пользователя. Глава 7 посвящена модели программирования MapReduce и различным форматам данных, с которыми может работать MapReduce. В главе 8 рассматриваются нетривиальные аспекты MapReduce, включая сортировку и объединение данных.

Главы 9 и 10 предназначены для администраторов Hadoop. В них читатель узнает, как организуется настройка и сопровождение кластеров Hadoop с HDFS и MapReduce.

Дальнейшие главы посвящены проектам, построенным на базе Hadoop или связанным с этой технологией. В главах 11 и 12 представлены Pig и Hive — аналитические платформы на базе HDFS и MapReduce, тогда как в главах 13, 14 и 15 описаны HBase, ZooKeeper и Sqoop соответственно.

Что нового во втором издании?

Во второе издание вошли две новые главы, посвященные Hive и Sqoop (главы 12 и 15), новый раздел с описанием Avro (в главе 4) и вводное описание новых средств безопасности Hadoop (в главе 9).

В этом издании также описывается серия версий Apache Hadoop 0.20, потому что это были последние стабильные версии на момент написания книги. В тексте периодически упоминаются новые возможности из последующих выпусков (с указанием версии, в которой они появились).

Что нового в третьем издании?

В третьем издании рассматривается серия версий Apache Hadoop 1.x (бывшая 0.20), а также более новые серии — 0.22 и 2.x (бывшая 0.23). За некоторыми исключениями, упоминаемыми в тексте, все примеры в книге работают в этих версиях.

Общее описание возможностей каждой серии версий приведено в разделе «Выпуски Hadoop», с. 42.

В большинстве примеров этого издания используется новый MapReduce API. Так как старый API продолжает широко использоваться, он описывается в тексте вместе с новым API; эквивалентный код, использующий старый API, доступен на веб-сайте книги.

Главным новшеством Hadoop 2.0 стала новая исполнительная среда MapReduce – MapReduce 2, построенная на основе новой системы управления распределенными ресурсами YARN. В это издание вошли новые разделы, посвященные реализации MapReduce на базе YARN: как работает эта технология (глава 6) и как организовать ее выполнение (глава 9).

Также в книге рассмотрены и другие аспекты MapReduce: практические вопросы разработки (например, упаковка заданий MapReduce с использованием Maven, настройка пользовательских путей классов Java, написание тестов с использованием MRUnit – все эти темы рассматриваются в главе 5); дополнительные сведения о таких возможностях, как `OutputCommitter` и распределенный кэш (глава 8) и анализ памяти задач (глава 9). В новом разделе описана подготовка заданий MapReduce для обработки данных Avro (глава 4), а в другом – организация выполнения простой рабочей схемы MapReduce в Oozie (глава 5).

В главу, посвященную HDFS (глава 3), были включены вводные описания высокой доступности, федеративности и новых файловых систем, WebHDFS и HttpFS.

Главы, посвященные Pig, Hive, Sqoop и ZooKeeper, были расширены: в них вошли описания новых возможностей и изменений последних выпусков.

Кроме того, в книгу были внесены многочисленные исправления и усовершенствования.

Использование примеров кода

Эта книга написана для того, чтобы помочь вам в решении конкретных задач. В общем случае вы можете использовать приводимые примеры кода в своих программах и документации. Связываться с авторами для получения разрешения не нужно, если только вы не воспроизводите значительный объем кода. Например, если ваша программа использует несколько фрагментов кода из книги, обращаться за разрешением не нужно. Вместе с тем для продажи или распространения дисков CD-ROM с примерами из книг O'Reilly потребуется разрешение. Если вы отвечаете на вопрос на форуме, приводя цитату из книги с примерами кода, обращаться за разрешением не нужно. Если значительный объем кода из примеров книги включается в документацию по вашему продукту, разрешение необходимо.

Мы будем признательны за ссылку на источник информации, хотя и не требуем ее. Обычно в ссылке указывается название, автор, издательство и код ISBN. Если вы полагаете, что ваши потребности выходят за рамки оправданного использования примеров кода или разрешений, приведенных выше, свяжитесь с нами по адресу permissions@oreilly.com.

Благодарности

В работе над книгой мне (прямо или косвенно) помогало множество людей. Благодарю сообщество Hadoop, в котором я многому научился — и продолжаю учиться.

Хочу особо поблагодарить Майкла Стека (Michael Stack) и Джонатана Грея (Jonathan Gray) за главу о HBase. Спасибо Эдриану Вудхеду (Adrian Woodhead), Марку де Палолу (Marc de Palol), Джойдипу Сен Сарме (Joydeep Sen Sarma), Ашишу Тусы (Ashish Thusoo), Анджею Бялецки (Andrzej Bialecki), Стюю Худу (Stu Hood), Крису К. Уэнзелу (Chris K. Wensel) и Оуэну О’Малли (Owen O’Malley).

Я благодарен всем рецензентам, внесшим немало полезных предложений и улучшений в мои черновики: Рагу Ангади (Raghu Angadi), Мэтту Биддалфу (Matt Biddulph), Кристофу Бисилье (Christophe Bisciglia), Райану Коксу (Ryan Cox), Девараджу Дасу (Devaraj Das), Алексу Дорману (Alex Dorman), Крису Дугласу (Chris Douglas), Аллану Гейтсу (Alan Gates), Ларсу Джорджу (Lars George), Патрику Ханту (Patrick Hunt), Аарону Кимболлу (Aaron Kimball), Питеру Крейю (Peter Krey), Хайрону Куангу (Hairong Kuang), Саймону Максену (Simon Maxen), Ольге Наткович (Olga Natkovich), Бенджамину Риду (Benjamin Reed), Константину Швачко (Konstantin Shvachko), Аллену Уиттенauerу (Allen Wittenauer), Матею Захарии (Matei Zaharia) и Филиппу Зейлigerу (Philip Zeyliger). Аджай Ананд (Ajay Anand) помог нормально организовать процесс рецензирования. Филипп «флип» Кромер (Philip «flip» Kromer) любезно помог мне с примерами наборов метеорологических данных NCDC, использованных в примерах книги. Особого упоминания заслуживают Оуэн О’Малли (Owen O’Malley) и Арун С. Марти (Arun S. Murthy), объяснившие мне многие нюансы MapReduce. Конечно, ответственность за все оставшиеся ошибки лежит исключительно на мне.

Также я благодарен всем участникам, предоставившим подробные рецензии и обратную связь ко второму изданию книги: Джеффу Бину (Jeff Bean), Дугу Каттингу (Doug Cutting), Глинну Дархэмму (Glynn Durham), Аллану Гейтсу (Alan Gates), Джеффу Хаммербахеру (Jeff Hammerbacher), Алексу Козлову (Alex Kozlov), Кену Круглеру (Ken Krugler), Джимми Лину (Jimmy Lin), Toddzu Lipkonу (Todd Lipcon), Саре Спронле (Sarah Sproehnle), Винитре Варадхарараджан (Vinithra Varadharajan) и Иэну Ригли (Ian Wrigley) — и всем читателям, сообщившим об ошибках в первом издании.

Спасибо Аарону Кимболлу за главу о Sqoop, а Филиппу «флипу» Кромеру — за примеры.

Что касается третьего издания, я выражаю благодарность Александро Абдельнуру (Alejandro Abdnur), Еве Андреассон (Eva Andreasson), Эли Коллинзу (Eli Collins), Дугу Каттингу, Патрику Ханту, Аарону Кимболлу, Аарону Т. Майерсу (Aaron T. Myers), Броку Ноланду (Brock Noland), Арвинду Прабхакару (Arvind Prabhakar), Ахмеду Радвану (Ahmed Radwan) и Тому Уилеру (Tom Wheeler) за отзывы и предложения. Роб Уэлтман (Rob Weltman) любезно предоставил очень подробную обратную связь по всей книге, которая значительно улучшила окончательный вариант рукописи. Спасибо всем читателям, сообщившим об ошибках во втором издании.

Я особенно благодарен Дугу Каттингу за моральную поддержку и дружбу, а также за написанное им предисловие.

Спасибо всем, с кем я беседовал или общался по электронное почте в ходе работы над книгой.

Примерно на середине написания книги я поступил на работу в Cloudera. Благодарю своих коллег за поддержку, которая позволила мне найти необходимое время и быстро дописать книгу.

Благодарю своего редактора Майка Лукидеса (Mike Loukides) и его коллег из O'Reilly за помощь в подготовке книги. Майк постоянно находился на связи, отвечая на мои вопросы, знакомясь с первыми черновиками и помогая мне выдерживать график.

Наконец, работа над книгой оказалась весьма масштабной, и я ни за что не справился бы с ней без постоянной поддержки со стороны моей семьи. Моя жена Элиан не только взяла на себя работу по дому, но и участвовала в рецензировании, правке и анализе примеров. Мои дочери Эмилия и Лотти с пониманием относились к моим занятиям. Я с нетерпением жду, когда смогу проводить с ними больше времени.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

1

Знакомство с Hadoop

Первопроходцы перевозили тяжести на быках. И если бык не мог сдвинуть бревно, они не пытались вырастить быка побольше. Мы должны стремиться не к повышению мощности отдельных компьютеров, а к повышению численности компьютерных систем.

Греис Хоннер

Данные!

Мы живем в эпоху данных. Трудно оценить общий объем данных, хранимых в электронном виде, но по оценкам IDC, размер «цифровой Вселенной» составлял около 0,18 зеттабайта в 2006 году, а к 2011 году прогнозировалось его десятикратное увеличение до 1,8 зеттабайта¹. Один зеттабайт равен 10^{21} байтов, тысяче экзабайтов, миллиону петабайтов или миллиарду терабайтов. Величина получается примерно такой, как если бы у каждого жителя Земли в Сети был свой жесткий диск.

У моря данных многоисточков. Лишь несколько примеров²:

- Нью-Йоркская фондовая биржа генерирует около одного терабайта коммерческих данных в день.

¹ См. Gantz et al., «The Diverse and Exploding Digital Universe», март 2008 г. (<http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf>).

² <http://www.intelligententerprise.com/showArticle.jhtml?articleID=207800705>, <http://mashable.com/2008/10/15/facebook-10-billion-photos/>, <http://blog.familytreemagazine.com/insider/Inside+Ancestrycoms+TopSecret+Data+Center.aspx>, <http://www.archive.org/about/faqs.php>, and <http://www.interactions.org/cms/?pid=1027032>

- На Facebook хранится приблизительно 10 миллиардов фотографий, занимающих около одного петабайта.
- На генеалогическом сайте Ancestry.com хранится около 2,5 петабайта данных.
- Internet Archive хранит около 2 петабайт данных и растет со скоростью 20 терабайт в месяц.
- Большой адронный коллайдер, расположенный недалеко от Женевы (Швейцария), будет выдавать около 15 петабайт данных в год.

Так что объем электронных данных огромен. Но вас, вероятно, интересует, какое отношение это имеет к вам? Ведь большая часть данных хранится в крупных веб-хранилищах (например, поисковых системах) и научных и финансовых организациях? Отразится ли причество «Больших Данных», как это принято называть, на жизни меньших организаций и отдельных людей?

Я считаю, что отразится. Возьмем к примеру фотографии. Дед моей жены — страстный фотограф, занимавшийся любимым делом на протяжении всей своей жизни. Полный архив его снимков, слайдов и 35-миллиметровых пленок, отсканированных в высоком разрешении, занимает около 10 гигабайт. Сравните с цифровыми фотографиями, которые моя семья сделала в 2008 году — они занимали около 5 гигабайт. Моя семья производит фотографические данные в 35 раз быстрее, чем дед жены, и с каждым годом эта скорость только возрастает: снимать становится все проще, люди делают все больше снимков.

В более общем смысле цифровые потоки, производимые отдельными людьми, быстро растут. Проект MyLifeBits (Microsoft Research) дает представление об архивировании личной информации, которое может стать обыденным явлением в недалеком будущем. В ходе эксперимента MyLifeBits все электронные взаимодействия отдельного человека — телефонные звонки, электронная почта, документы — сохраняются в электронном виде. В собираемые данные включаются ежеминутные фотографии, в результате общий объем данных составляет до гигабайта в месяц. Когда стоимость хранения данных снизится до уровня, при котором станет приемлемым хранение непрерывных аудио- и видеоданных, объем данных будущего сервиса MyLifeBits многократно возрастет.

Короче, объем данных, производимых отдельным человеком, растет. Но еще важнее другое: объем данных, производимых машинами, растет еще быстрее. Журналы, системы отслеживания RFID, сенсорные сети, данные GPS, розничные сделки — все это вносит свой вклад в растущую гору данных.

Объем общедоступных данных тоже растет с каждым годом. Организации уже не ограничиваются обработкой только собственных данных; в будущем успех будет в значительной степени определяться их способностью извлекать полезную информацию из данных других организаций. Такие проекты, как Public Data Sets

в Amazon Web Services, Infochimps.org и theinfo.org, предоставляют «информационные площадки» для хранения данных, которые могут бесплатно (или за умеренную плату в случае AWS) загружаться и анализироваться любым желающим. Совмещение данных нескольких информационных источников открывает много неожиданных, а следовательно, невообразимых ранее возможностей.

Возьмем хотя бы проект Astrometry.net, отслеживающий новые фотографии ночного неба в группе Astrometry (Flickr). Каждое изображение анализируется с определением того, какая часть неба на нем изображена и присутствуют ли на нем интересные небесные тела (звезды, галактики и т. д.). Перед нами хороший пример того, что происходит, когда данные (в нашем случае фотографии) становятся общедоступными и используются для целей, не предусмотренных их создателем (анализ изображений).

Есть такая поговорка: «больше данных — лучше, чем хорошие алгоритмы». Другими словами, какие бы хитроумные алгоритмы вы ни применяли, некоторые задачи (скажем, рекомендации фильмов или музыки на основании прошлых предпочтений) часто просто решаются за счет накопления большего объема данных (с менее сложными алгоритмами)¹.

В общем, Большие Данные — это хорошо. Плохо то, что с их хранением и анализом возникает множество проблем.

Хранение и анализ данных

Суть этих проблем проста: хотя за прошедшие годы емкость жестких дисков значительно возросла, скорость доступа (то есть скорость чтения данных с диска) от нее отстает. Типичный жесткий диск 1990 года вмещал 1370 Мбайт данных со скоростью передачи 4,4 Мбайт/с², так что чтение всех данных с заполненного диска занимало около 5 минут. Спустя 20 лет терабайтные диски стали обыденным явлением, но скорость передачи составляет около 100 Мбайт/с, так что чтение всех данных с диска займет около 2,5 часа.

Чтение всех данных с одного диска выполняется слишком медленно, а запись — и того медленнее. Очевидный способ сокращения времени чтения заключается

¹ Цитата позаимствована из статьи Ананда Раджарамана (Anand Rajaraman) о Netflix Challenge (<http://anand.typepad.com/datawocky/2008/03/more-data-usual.html>). Алон Халеви (Alon Halevy), Питер Норвиг (Peter Norvig) и Фернандо Перейра (Fernando Pereira) высказывают аналогичное мнение в статье «The Unreasonable Effectiveness of Data», IEEE Intelligent Systems, март/апрель 2009 г.

² Приведены характеристики жесткого диска Seagate ST-41600n.

в одновременном чтении данных с нескольких дисков. Представьте, что у вас имеется 100 дисков, на каждом из которых хранится 1/100 часть данных. При параллельной работе этих дисков данные будут прочитаны за две минуты.

На первый взгляд идея использования одной сотой части диска кажется расточительной. Но мы можем хранить сто наборов данных, каждый из которых занимает один терабайт, и организовать совместный доступ к ним. Возможно, пользователи такой системы охотно согласятся на общий доступ в обмен на ускорение анализа данных; кроме того, статистически их задания по анализу данных с большей вероятностью окажутся распределенными по времени и будут в меньшей степени мешать друг другу. Однако концепция параллельного чтения и записи данных на нескольких дисках не так проста.

Во-первых, необходимо учесть возможность сбоев оборудования; как только вы начинаете использовать много устройств вместо одного, вероятность сбоя на одном из них значительно повышается. Стандартным способом предотвращения потери данных является репликация: система хранит избыточные копии данных, чтобы в случае сбоя была доступна другая копия. Например, так работают массивы RAID, хотя, как вы вскоре узнаете, файловая система Hadoop – HDFS (Hadoop Distributed Filesystem) – использует несколько иной подход.

Во-вторых, в большинстве задач анализа данных требуется каким-то образом объединять данные. Может оказаться, что данные, прочитанные с одного диска, должны объединяться с данными всех остальных 99 дисков. Разные распределенные системы позволяют объединять данные из нескольких источников, но задача эта пользуется дурной славой из-за своей сложности. MapReduce предоставляет модель программирования, которая абстрагирует задачу от дисковых операций чтения и записи, преобразуя ее в вычисления с наборами ключей и значений. Эта модель подробно рассматривается в следующих главах, а сейчас важно упомянуть, что обработка данных состоит из двух шагов: отображения (Map) и свертки (Reduce); объединение данных происходит на их границе. Как и HDFS, MapReduce обладает встроенными механизмами надежности.

Собственно, в этом и заключается функциональность Hadoop: система надежного общего хранения и анализа данных. HDFS обеспечивает хранение, а MapReduce – анализ. Hadoop содержит и другие компоненты, но эти возможности образуют ядро системы.

Сравнение с другими системами

На первый взгляд кажется, что подход, используемый MapReduce, основан на методе «грубой силы». Изначально предполагается, что с каждым запросом

обрабатывается весь набор данных — или, по крайней мере, его значительная часть. Но в этом заключается его мощь. MapReduce представляет собой процессор *пакетных* запросов, и его способность выполнить произвольный запрос ко всему набору данных и получить результаты за разумное время имеет решающее значение. Она изменяет отношение к данными и открывает доступ к данным, которые прежде хранились в архивах на лентах или дисках. У пользователей появляется возможность экспериментировать с данными. Ответы, которые ранее занимали слишком много времени, сейчас находятся достаточно быстро, а это, в свою очередь, порождает новые вопросы и новые идеи.

Например, Mailtrust (почтовое подразделение Rackspace) использует Hadoop для обработки журналов электронной почты. Один из написанных ими запросов должен был определять географическое распределение пользователей. По их словам, результат был таким:

«Информация оказалась настолько полезной, что мы организовали ежемесячное выполнение задания MapReduce. На основании полученных данных мы принимали решения о том, в каких центрах обработки данных Rackspace следует размещать новые серверы электронной почты в процессе роста».

Консолидация нескольких сотен гигабайт данных и наличие инструментов для их анализа позволили специалистам Rackspace получить представление данных, которое в противном случае было бы для них недоступно. Более того, полученная информация была использована для повышения качества обслуживания клиентов.

Hadoop и РСУБД

Почему бы не использовать для выполнения крупномасштабного пакетного анализа базы данных с множеством дисков? Почему необходима технология MapReduce?

Ответ на эти вопросы обусловлен еще одной тенденцией в области дисковых накопителей: скорость позиционирования улучшается медленнее скорости передачи данных. Позиционированием называется процесс перемещениячитывающей головки к определенному месту диска для чтения или записи данных. Скорость позиционирования определяет задержку при выполнении дисковых операций, тогда как скорость передачи данных определяет пропускную способность канала взаимодействия с диском.

Если в схеме обращения к данным преобладают операции позиционирования, то чтение и запись больших частей набора данных займут больше времени, чем при

потоковых операциях, выполняемых со скоростью передачи данных. С другой стороны, для обновления относительно небольшой части записей в базе данных хорошо подходят традиционные В-деревья (структура данных, используемая в реляционных базах данных, недостатком которой является ограниченная скорость позиционирования). При обновлениях значительной части базы данных В-дерево оказывается менее эффективным, чем технология MapReduce, использующая сортировку со слиянием для обновления базы данных.

Во многих отношениях MapReduce может рассматриваться как дополнение к технологии реляционных систем управления базами данных (РСУБД) (различия между двумя системами представлены в табл. 1.1). MapReduce хорошо подходит для задач, которые требуют пакетного (и особенно несистематического) анализа всего набора данных. РСУБД хорошо подходит для точечных запросов или обновлений, при которых база данных была предварительно проиндексирована для обеспечения быстрой выборки и обновления при относительно небольших объемах данных. MapReduce подходит для приложений, в которых данные записываются один раз, а читаются многократно, тогда как реляционные базы данных хорошо подходят для часто обновляемых наборов данных.

Таблица 1.1. Сравнение РСУБД с MapReduce

Параметр	Традиционная РСУБД	MapReduce
Размер данных	Гигабайты	Петабайты
Доступ	Интерактивный и пакетный	Пакетный
Обновления	Многократное чтение и запись	Однократная запись, многократное чтение
Структура	Статическая схема	Динамическая схема
Целостность	Высокая	Низкая
Масштабирование	Нелинейное	Линейное

Другое различие между MapReduce и РСУБД — структурированность наборов данных, с которыми они работают. *Структурированные данные* разделены на сущности, имеющие определенный формат, — например, документы XML или таблицы базы данных, соответствующие конкретной, заранее определенной схеме. В этой области правит РСУБД РСРСР. С другой стороны, наполовину структурированные данные устроены менее формально; даже если схема присутствует, она часто описывает только общую структуру данных: например, в электронной таблице структурой является сетка из ячеек, тогда как сами ячейки могут содержать произвольные данные. *Неструктурированные данные* не имеют никакой

определенной внутренней структуры: это может быть простой текст, графические данные и т. д. MapReduce хорошо работает с неструктурированными или полу-структурированными данными, потому что эта технология проектировалась для интерпретирования данных во время обработки. Иначе говоря, входные ключи и значения MapReduce не являются внутренними свойствами данных, а выбираются человеком, анализирующим эти данные.

Реляционные данные часто *нормализуются* для сохранения целостности и устранения избыточности. Нормализация создает проблемы для MapReduce, потому что с ней чтение записи становится не-локальной операцией, а одной из важнейших предпосылок работы MapReduce является возможность выполнения (высокоскоростных) потоковых операций чтения и записи.

Хороший пример *ненормализованного* набора записей — журнал веб-сервера (например, в каждой записи клиентские имена хостов указываются полностью, несмотря на то, что каждый клиент может встречаться в журнале многократно). По этой причине всевозможные журнальные файлы особенно хорошо подходят для анализа средствами MapReduce.

Модель программирования MapReduce является линейно масштабируемой. Программа пишет две функции (функцию отображения и функцию свертки), каждая из которых определяет отображение одного набора пар «ключ-значение» на другой набор. Эти функции не зависят от размера данных или кластера, с которым они работают, поэтому они могут использоваться без изменений как для небольшого, так и для гигантского набора данных. Важнее другое: при удвоении размера входных данных задание будет выполняться вдвое медленнее, но при удвоении размера кластера задание будет выполняться так же быстро, как и исходное. Для запросов SQL в общем случае это утверждение не выполняется.

Впрочем, со временем различия между реляционными базами данных и системами MapReduce будут стираться — по мере того, как в реляционных базах данных начнут реализовываться некоторые концепции MapReduce (как, например, в базах данных Aster Data и Greenplum), и, с другой стороны, высокоуровневые языки запросов на базе MapReduce (такие, как Pig и Hive) сделают системы MapReduce более доступными для программистов традиционных баз данных¹.

¹ В январе 2007 года Дэвид Дж. ДеВитт (David J. DeWitt) и Майкл Стоунбрекер (Michael Stonebraker) подняли шум своей статьей «MapReduce: большой шаг назад». В ней они критиковали MapReduce как плохую замену для реляционных баз данных. Многие комментаторы посчитали такое сравнение некорректным — как, например, Марк К. Чу-Кэррол (Mark C. Chu-Carroll), назвавший свой ответ «База данных — молоток, MapReduce — отвертка». В своей следующей статье, «MapReduce II», ДеВитт и Стоунбрекер уделили внимание темам, поднятым их оппонентами.

Распределенные вычисления

Сообщества высокопроизводительных (HPC, High Performance Computing) и распределенных вычислений (Grid Computing) годами занимались крупномасштабной обработкой данных с использованием таких прикладных интерфейсов (API), как MPI (Message Passing Interface). Не вдаваясь в подробности, подход HPC заключался в распределении работы по кластерам машин, работающих с общей файловой системой, находящейся под управлением сети хранения данных SAN (Storage Area Network). Такой подход хорош для заданий, требующих большого объема вычислений. Однако он создает проблемы, когда у узлов появляется необходимость в обращении к большим объемам данных (сотни гигабайт — порог, после которого MapReduce предстает в полном блеске), так как пропускная способность сети становится «узким местом», а узлы начинают простаивать.

MapReduce стремится размещать данные в пределах вычислительных узлов; при этом обращения к данным выполняются быстро, так как фактически являются локальными¹. Эта особенность, называемая *локальностью данных*, лежит в основе технологии MapReduce и является причиной ее хорошей производительности. Проектировщики понимают, что пропускная способность сети — самый ценный ресурс в среде центров обработки данных (лишнее копирование данных легко приводит к перегрузке сетевых каналов), и реализации MapReduce стараются по возможности экономить ее за счет явного моделирования топологии сети. Следует учесть, что этот подход не препятствует проведению интенсивных вычислений на базе MapReduce.

С MPI программист может полностью управлять происходящим, но ему приходится явно определять реализацию всей механики передачи данных, выражая ее в виде низкоуровневых функций C и таких конструкций, как сокеты (наряду с высокоуровневыми алгоритмами анализа). MapReduce работает только на высоком уровне: программист мыслит понятиями функций пар «ключ-значение», а управление потоком данных осуществляется неявно.

Координирование процессов в крупномасштабной распределенной среде создает немало трудностей. Самый сложный аспект — корректная обработка неполных сбоев (когда вы не знаете, произошел сбой в удаленном процессе или нет) с продвижением вычислений в целом. MapReduce избавляет программиста от необходимости думать о сбоях, потому что реализация обнаруживает сбойные задачи отображения или свертки и планирует их заново на работоспособных машинах. MapReduce может делать это, потому что относится к категории архитектур без разделения (*shared-nothing*), то есть задачи не зависят друг от друга. (Вообще

¹ Джим Грей (Jim Gray) был одним из первых сторонников проведения вычислений вблизи данных. См. «Distributed Computing Economics», март 2003 г., <http://research.microsoft.com/apps/pubs/default.aspx?id=70001>.

говоря, это утверждение излишне упрощено, потому что вывод функций отображения передается функциям свертки, но это происходит под контролем системы MapReduce; в этом случае повторное выполнение сбойной функции свертки требует несколько больших усилий, чем повторное выполнение сбойной функции отображения, потому что система должна убедиться в доступности необходимых результатов отображения и, если они недоступны, сгенерировать их заново посредством выполнения соответствующих отображений.) Таким образом, с точки зрения программиста, порядок выполнения задач значения не имеет. Напротив, программы MPI вынуждены явно управлять своими контрольными точками и восстановлением. Это повышает степень контроля программиста за происходящим, но усложняет написание программ.

На первый взгляд может показаться, что модель программирования MapReduce ограничивает программиста, и в каком-то смысле это действительно так: вы ограничиваетесь типами ключей и значений, связанных определенным образом, а функции отображения и свертки выполняются с минимальной координацией (функции отображения передают ключи и значения функциям свертки). Возникает естественный вопрос: можно ли сделать в таких условиях что-то полезное или нетривиальное?

Ответ — да, можно. Технология MapReduce была разработана инженерами Google для построения реальных поисковых индексов, так как они обнаружили, что им приходится снова и снова решать одну и ту же задачу (причем разработка MapReduce вдохновлялась традиционными идеями функционального программирования, распределенных вычислений и баз данных), но с тех пор она нашла множество применений в самых разных отраслях. Просто удивительно, насколько широкий спектр алгоритмов может быть выражен средствами MapReduce — от анализа изображений до обработки графов и машинного самообучения¹. Конечно, MapReduce не является универсальным средством решения любых задач, но это отличный инструмент обработки данных общего назначения.

Добровольные вычисления

Люди, впервые услышавшие о Hadoop и MapReduce, часто спрашивают: «И чем это отличается от SETI@home?» Организация SETI (Search for Extra-Terrestrial Intelligence, то есть «поиск внеземного разума») ведет проект SETI@home, в котором добровольцы предоставляют процессорное время своих пристаивающих компьютеров для анализа данных радиотелескопов с целью поиска признаков

¹ Apache Mahout (<http://mahout.apache.org/>) — проект построения библиотек машинного самообучения (алгоритмы группировки, классификации и т.д.) на базе Hadoop.

разумной внеземной жизни. SETI@home — самый известный из многочисленных проектов «добровольных вычислений» (volunteer computing); другие проекты такого рода — GIMPS (Great Internet Mersenne Prime Search) (для поиска больших простых чисел) и Folding@home (для понимания принципов фолдинга белка и его отношения к заболеваниям).

Задача, которую пытается решить проект добровольных вычислений, разбивается на фрагменты, называемые «рабочими блоками» (work units). Фрагменты рассылаются на компьютеры, находящиеся в разных странах, для анализа. Например, рабочий блок SETI@home содержит около 0,35 Мбайт данных радиотелескопов, а его анализ на типичном домашнем компьютере занимает несколько часов или дней. После завершения работы результаты отправляются обратно на сервер, а клиент получает следующий рабочий блок. Для предотвращения фальсификаций каждый рабочий блок отправляется на три разных компьютера, и результат принимается только в случае совпадения двух и более результатов.

Возможно, между SETI@home и MapReduce на первый взгляд действительно существует некоторое сходство (задача разбивается на независимые фрагменты, которые могут выполняться параллельно), но есть и серьезные различия. Задача SETI@home требует больших затрат вычислительных ресурсов, поэтому он хорошо подходит для выполнения на тысячах компьютеров по всему миру¹, потому что время пересылки заданий было ничтожно мало по сравнению с временем их обработки. Добровольцы жертвовали вычислительные ресурсы, а не пропускную способность своих каналов.

Технология MapReduce проектировалась для выполнения заданий, которые за несколько минут или часов выполняются на доверенном оборудовании, находящемся в одном центре обработки данных с очень большой совокупной пропускной способностью. Задания SETI@home выполняются на компьютерах в Интернете, которые не являются доверенными, имеют разную скорость каналов связи и не обеспечивают локальности данных.

Краткая история Hadoop

Hadoop создал Дуг Каттинг — создатель Apache Lucene, широко используемой библиотеки текстового поиска. Hadoop происходит от Apache Nutch — системы веб-поиска с открытым кодом, которая сама по себе являлась частью проекта Lucene.

¹ В январе 2008 года проект SETI@home по данным http://www.planetary.org/programs/projects/setiathome_setiathome_20080115.html ежедневно обрабатывал 300 Гбайт данных на 320 000 компьютеров (большинство из которых не специализировалось на SETI@home, а также выполняло другую работу).

ПРОИСХОЖДЕНИЕ НАЗВАНИЯ

Название Hadoop не является сокращением. Создатель проекта Дуг Каттинг объясняет, откуда оно произошло:

«Это имя, которое мой сын придумал для плюшевого желтого слона. Короткое, относительно легко произносимое, бессмысленное и не используемое в другом контексте: это мои критерии выбора имен. Детям хорошо удаются такие имена. Слово Googol тоже придумал ребенок».

Подпроектам и модулям Hadoop также обычно присваиваются имена, никак не связанные с их функциями, часто связанные с темой животных (как, например, «Pig»). Меньшим компонентам даются более содержательные (а следовательно, более привычные) названия. Это полезный принцип, так как он обычно позволяет определить, что делает тот или иной компонент, по его имени — например, jobtracker отслеживает задания MapReduce.

Построение поисковой системы «с нуля» было амбициозной целью — не только из-за сложности написания программного обеспечения обхода и индексирования веб-сайтов, но и из-за сложности ведения проекта (содержащего огромное количество «подвижных частей») без специально выделенной группы. Кроме того, задача была весьма дорогостоящей: по оценкам Майка Кафареллы и Дуга Каттинга, оборудование системы, ведущей индекс для миллиарда страниц, стоило полмиллиона долларов, а ежемесячные затраты на ее содержание составляли \$30 000¹. Тем не менее они считали, что цель того стоит, так как результатом будет открытие и исключительная демократизация алгоритмов поисковых систем.

Проект Nutch был запущен в 2002 году. Работоспособный обходчик и поисковая система появились очень быстро. Однако разработчики поняли, что их архитектура не будет масштабироваться на миллиарды веб-страниц. Помощь пришла в 2003 году, когда была опубликована статья с описанием архитектуры GFS (Google File System) — распределенной файловой системы, которая использовалась в реальных проектах Google². Система GFS или что-нибудь в этом роде решила бы проблемы с необходимостью хранения очень больших файлов, генерируемых в процессе обхода и индексирования. В частности, система GFS сэкономила бы время на

¹ См. Mike Cafarella and Doug Cutting, «Building Nutch: Open Source Search», ACM Queue, April 2004, <http://queue.acm.org/detail.cfm?id=988408>.

² Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, «The Google File System», октябрь 2003 г., <http://labs.google.com/papers/gfs.html>.

выполнение таких административных задач, как управление узлами хранения данных. В 2004 году разработчики взялись за написание реализации такой системы с открытым кодом — NDFS (Nutch Distributed Filesystem).

В 2004 году была опубликована статья, в которой компания Google представила миру технологию MapReduce¹. В начале 2005 года у разработчиков Nutch появилась работоспособная реализация MapReduce на базе Nutch, а к середине года все основные алгоритмы Nutch были адаптированы для использования MapReduce и NDFS.

Возможности применения NDFS и реализации MapReduce в Nutch выходили далеко за рамки поиска, и в феврале 2006 года был образован независимый подпроект Lucene, получивший название Hadoop. Примерно в то же время Дуг Каттинг поступил в компанию Yahoo!, которая предоставила группу и ресурсы для превращения Hadoop в систему, работающую в веб-масштабах (см. далее врезку «Hadoop в Yahoo!»). Результаты были продемонстрированы в феврале 2008 года, когда компания Yahoo! объявила, что используемый ею поисковый индекс был сгенерирован 10 000-ядерным кластером Hadoop².

В январе 2008 года проект Hadoop стал одним из ведущих проектов Apache, что доказало его успех и наличие разнообразного, активного сообщества. К этому времени технология Hadoop использовалась не только в Yahoo!, но и во многих других компаниях — например, в Last.fm, Facebook и New York Times. Примеры практического применения Hadoop приведены в вики Hadoop.

Хорошо известный факт: в New York Times облако Amazon EC2 было использовано для обработки свыше 4 терабайт отсканированных архивов газеты с преобразованием их в формат PDF для Web³. Обработка на 100 машинах заняла менее 24 часов. Вероятно, этот проект не был бы воплощен в жизнь, если бы не сочетание модели почасовой оплаты Amazon (позволившей NYT получить доступ к большому количеству компьютеров на непродолжительное время) с простой и удобной параллельной моделью программирования Hadoop.

В апреле 2008 года технология Hadoop побила мировой рекорд по скорости сортировки терабайта данных. На 910-узловом кластере один терабайт был отсортирован за 209 секунд (чуть менее 3,5 минуты), тогда как рекорд предыдущего года составлял 297 секунд. В ноябре того же года компания Google сообщила, что ее

¹ Jeffrey Dean, Sanjay Ghemawat, «MapReduce: Simplified Data Processing on Large Clusters», декабрь 2004 года, <http://labs.google.com/papers/mapreduce.html>.

² «Yahoo! Launches World's Largest Hadoop Production Application», 19 февраля 2008 г., <http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html>.

³ Derek Gottfrid, «Self-service, Prorated Super Computing Fun!», 1 ноября 2007 г., <http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/>.

реализация MapReduce отсортировала один терабайт за 68 секунд¹. На момент публикации первого издания этой книги (май 2009 года) было объявлено, что рабочая группа в Yahoo! использовала Hadoop для сортировки одного терабайта за 62 секунды.

С того времени технология Hadoop стремительно входит в повседневную деятельность крупных компаний. Отрасль признала ее роль как платформы хранения и анализа данных общего назначения, и этот факт отражен в количестве продуктов, тем или иным образом использующих или интегрирующих Hadoop. Существуют дистрибутивы Hadoop как от крупных общепризнанных фирм, включая EMC, IBM, Microsoft и Oracle, так и от компаний с узкой специализацией — таких, как Cloudera, Hortonworks и MapR.

HADOOP В YAHOO!

Для построения поисковых систем, работающих в масштабе Интернета, необходимы огромные объемы данных, а следовательно, множество машин, занятых их обработкой. Система Yahoo! Search состоит из четырех основных компонентов: обходчик Crawler, загружающий страницы с веб-серверов; WebMap, строящий граф известной части Web; Indexer, строящий обратный индекс лучших страниц; и Runtime, отвечающий на запросы пользователей. Граф WebMap состоит приблизительно из 1 триллиона (10^{12}) ребер, каждое из которых представляет собой веб-ссылку, и 100 миллиардов (10^{11}) узлов, каждый из которых представляет собой отдельный URL-адрес. Создание и анализ таких больших графов требуют многодневной работы множества компьютеров. В начале 2005 года возникла необходимость в переработке инфраструктуры WebMap, называемой Dreadnaught, с целью масштабирования для большего количества узлов. Инфраструктура Dreadnaught успешно масштабировалась от 20 до 600 узлов, но для дальнейшего масштабирования ее пришлось спроектировать заново. Dreadnaught во многих отношениях напоминает MapReduce, но обладает большей гибкостью при меньшей структуризации. В частности, каждый фрагмент задания Dreadnaught может пересыпать выходные данные всем фрагментам следующей стадии задания, но вся сортировка выполнялась библиотечным кодом. На практике большинство фаз WebMap представляло собой пары, соответствующие структуре MapReduce. Таким образом, приложения WebMap могли адаптироваться для MapReduce без значительной переработки.

¹ «Sorting 1PB with MapReduce», 21 ноября 2008 г., <http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html>.

Эрик Бальдешвилер (Eric Baldeschwieler) (Eric14) создал небольшую группу, и мы приступили к проектированию и построению прототипа новой инфраструктуры, написанной на C++ по образцу GFS и MapReduce, которая должна была заменить Dreadnaught. Хотя на тот момент существовала насущная потребность в новой инфраструктуре для WebMap, было очевидно, что стандартизация пакетных платформ для Yahoo! Search играет очень важную роль. Если мы сделаем свою инфраструктуру достаточно общей для поддержки других пользователей, это повысит эффективность вложений в новую платформу.

В то же время мы следили за ходом проекта Hadoop, который тогда был частью Nutch. В январе 2006 года компания Yahoo! приняла на работу Дуга Каттинга, а еще через месяц мы решили отказаться от своего прототипа и перейти на Hadoop. Преимущество Hadoop перед нашим прототипом заключалось в том, что технология уже работала в реальном приложении (Nutch) с 20 узлами. Это позволило нам через два месяца запустить исследовательский кластер и помогать реальным клиентам использовать новую инфраструктуру намного раньше, чем если бы мы пошли по другому пути. Конечно, было и другое преимущество: так как проект Hadoop уже существовал на условиях открытого кода, нам было легче (хотя не так чтобы совсем легко!) получить разрешение от юридического отдела Yahoo! на работу с открытым кодом. Итак, в начале 2006 года мы развернули 200-узловой кластер для исследователей. Планы по преобразованию WebMap были временно приостановлены, пока мы занимались поддержкой и совершенствованием Hadoop для пользователей, ведущих исследовательскую работу.

Краткий график того, как проходила работа:

- 2004: Дуг Каттинг и Майк Кафарелла создают первые реализации того, что позднее станет HDFS и MapReduce.
- Декабрь 2005: Nutch перерабатывается для новой инфраструктуры. Hadoop надежно работает на 20 узлах.
- Январь 2006: Дуг Каттинг поступает в Yahoo!.
- Февраль 2006: официальный запуск проекта Apache Hadoop для поддержки автономной разработки MapReduce и HDFS.
- Февраль 2006: группа Yahoo! Grid переходит на Hadoop.
- Апрель 2006: эталонные тесты сортировки (10 Гбайт/узел) выполняются на 188 узлах в течение 47,9 часа.

- Май 2006: Yahoo! развертывает исследовательский кластер Hadoop из 300 узлов.
- Май 2006: эталонные тесты сортировки выполняются на 500 узлах в течение 42 часов (на более мощном оборудовании, чем в апрельских тестах).
- Октябрь 2006: исследовательский кластер достигает 600 узлов.
- Декабрь 2006: эталонные тесты выполняются на 20 узлах в течение 1,8 часа, 100 узлах в течение 3,3 часа, 500 узлах в течении 5,2 часа, и на 900 узлах в течение 7,8 часа.
- Январь 2007: исследовательский кластер достигает 900 узлов.
- Апрель 2007: вместо одного исследовательского кластера появляются два кластера по 1000 узлов.
- Апрель 2008: рекордный результат в эталонном тесте 1-терабайтовой сортировки: 209 секунд при 900 узлах.
- Октябрь 2008: ежедневная загрузка 10 терабайт данных в исследовательских кластерах.
- Март 2009: 17 кластеров, содержащие в общей сложности 24000 узлов.
- Апрель 2009: очередная победа в эталонных тестах: 500 Гбайт отсортированы за 59 секунд (1400 узлов), а 100 терабайт отсортированы за 173 минуты (3400 узлов).

Оуэн О’Мэлли

Apache Hadoop и экосистема Hadoop

Хотя Hadoop чаще всего ассоциируется с MapReduce и распределенной файловой системой (HDFS, ранее называвшейся NDFS), этим термином часто обозначают целое семейство взаимосвязанных проектов, объединенных инфраструктурой распределенных вычислений и крупномасштабной обработки данных.

Все базовые проекты, рассматриваемые в книге, ведутся фондом Apache Software Foundation, предоставляющим поддержку сообщества проектов с открытым кодом — включая исходный HTTP-сервер, от которого произошло название. С расширением экосистемы Hadoop появляются новые проекты, не обязательно находящиеся под управлением Apache, но предоставляющие дополнительные функции Hadoop или образующие абстракции более высокого уровня на основе базовой функциональности.

Ниже кратко перечислены проекты Hadoop, рассмотренные в книге.

Common – набор компонентов и интерфейсов для распределенных файловых систем и общего ввода/вывода (сериализация, Java RPC, структуры данных).

Avro – система сериализации для выполнения эффективных межъязыковых вызовов RPC и долгосрочного хранения данных.

MapReduce – модель распределенной обработки данных и исполнительная среда, работающая на больших кластерах типовых машин.

HDFS – распределенная файловая система, работающая на больших кластерах стандартных машин.

Pig – язык управления потоком данных и исполнительная среда для анализа очень больших наборов данных. Pig работает в HDFS и кластерах MapReduce.

Hive – распределенное хранилище данных. Hive управляет данными, хранимыми в HDFS, и предоставляет язык запросов на базе SQL (которые преобразуются ядром времени выполнения в задания MapReduce) для работы с этим данными.

HBase – распределенная столбцово-ориентированная база данных. HBase использует HDFS для организации хранения данных и поддерживает как пакетные вычисления с использованием MapReduce, так и точечные запросы (произвольное чтение данных).

ZooKeeper – распределенный координационный сервис высокой доступности. ZooKeeper предоставляет примитивы, которые могут использоваться для построения распределенных приложений (например, распределенные блокировки).

Sqoop – инструмент эффективной массовой пересылки данных между структурированными хранилищами (такими, как реляционные базы данных) и HDFS.

Oozie – сервис запуска и планирования заданий Hadoop (включая задания MapReduce, Pig, Hive и Sqoop jobs).

Выпуски Hadoop

Какую версию Hadoop стоит использовать? Конечно, ответ на этот вопрос изменяется со временем и зависит от того, какая функциональность вам нужна. В этом разделе кратко описаны высокоуровневые возможности последних серий выпусков Hadoop.

Существует несколько активных серий выпусков. Серия 1.x является продолжением версии 0.20 и содержит самые стабильные версии Hadoop из доступных на данный момент. В эту серию включена поддержка аутентификации Kerberos, предотвращающей несанкционированный доступ к данным Hadoop (см.

«Безопасность», с. 419). Почти все кластеры, находящиеся в эксплуатации, используют эти или производные версии (например, коммерческие дистрибутивы).

Серии 0.22 и 2.x¹ пока не стабильны (на начало 2012 года), но, вероятно, ситуация изменится со временем, когда они будут лучше протестированы в ходе реальной работы (за последней информацией о статусе обращайтесь к страницам выпусков Apache Hadoop). Серия 2.x включает в себя ряд важных новых возможностей:

- Новая исполнительная среда MapReduce, называемая MapReduce 2, реализована на базе новой системы YARN (Yet Another Resource Negotiator) – общей системы управления ресурсами для распределенных приложений. MapReduce 2 заменяет «классическую» исполнительную среду предыдущих версий. Более подробное описание MapReduce 2 приведено в разделе «YARN (MapReduce 2)» на с. 265.
- HDFS Federation – механизм разбиения пространства имен HDFS по узлам имен для поддержки кластеров с очень большим количеством файлов. См. «HDFS Federation» на с. 84.
- Механизм высокой доступности HDFS исключает из архитектуры единые точки сбоев в виде узлов имен; для этого поддерживаются резервные узлы имен, обеспечивающие восстановление работоспособности. См. «Высокая доступность HDFS» на с. 85.

В табл. 1.2 представлены только функции HDFS и MapReduce. Другие продукты экосистемы Hadoop тоже постоянно развиваются, и выбрать набор компонентов, хорошо работающих вместе, бывает непросто. К счастью, вам не придется заниматься этой работой самостоятельно. Проект Apache Bigtop (<http://incubator.apache.org/bigtop/>) проводит тестирование совместимости для стеков компонентов Hadoop и предоставляет пакеты Linux (RPM и пакеты Debian) для простой установки. Также некоторые коммерческие фирмы предлагают дистрибутивы Hadoop с наборами совместимых компонентов.

Таблица 1.2. Возможности, поддерживаемые разными сериями выпусков Hadoop

Возможность	1.x	0.22	2.x
Аутентификация	Да	Нет	Да
Старые имена конфигурационных свойств	Да	Не рекомендуется	Не рекомендуется

продолжение ↗

¹ Когда книга уже была отправлена в печать, сообщество Hadoop проголосовало за переименование серии версий 0.23 в серию 2.x. Термином «версии после 1.x» в книге обозначаются версии серий 0.22 и 2.x (ранее 0.23).

Таблица 1.2 (продолжение)

Возможность	1.x	0.22	2.x
Новые имена конфигурационных свойств	Нет	Да	Да
Старый API MapReduce	Да	Да	Да
Новый API MapReduce	Да (без некоторых библиотек)	Да	Да
Исполнительная среда MapReduce 1 (классическая)	Да	Да	Нет
Исполнительная среда MapReduce 2 (YARN)	Нет	Нет	Да
HDFS Federation	Нет	Нет	Да
Высокая доступность HDFS	Нет	Нет	Да

О чем рассказано в книге

Материал книги относится ко всем выпускам из табл. 1.2. В тех случаях, когда функция доступна только в одном конкретном выпуске, это оговаривается в тексте.

Приведенный в книге код работает во всех сериях выпусков. Исключение составляет небольшое подмножество примеров, на что также явно указывается в тексте. В примерах кода на веб-сайте перечислены версии, в которых они были протестированы.

Имена конфигураций

Имена конфигурационных свойств были изменены после выхода версии 1.x для создания более стройной схемы выбора имен. Например, свойства HDFS, относящиеся к узлам имен, были снабжены префиксом *dfs.namenode*, так что свойство *dfs.name.dir* превратилось в *dfs.namenode.name.dir*. Аналогичным образом свойства MapReduce имеют префикс *mapreduce* вместо старого префикса *mapred*, поэтому имя *mapred.job.name* было заменено именем *mapreduce.job.name*.

Для свойств, существующих в версии 1.x, в книге используются старые (не рекомендуемые) имена, потому что они работают во всех перечисленных версиях Hadoop. Если вы используете выпуск Hadoop после 1.x, возможно, вам стоит использовать новые имена свойств в файлах конфигурации и коде, чтобы избежать

выдачи предупреждений. Таблица с перечнем устаревших имен свойств и их замен находится на сайте Hadoop по адресу <http://hadoop.apache.org/common/docs/r0.23.0/hadoop-project-dist/hadoop-common/DeprecatedProperties.html>.

MapReduce API

Hadoop предоставляет два Java MapReduce API, более подробно описанных в разделе «Старый и новый JavaScript MapReduce API» на с. 60. В примерах этого издания книги используется новый API, работающий со всеми перечисленными версиями (кроме немногочисленных случаев недоступности библиотеки MapReduce, использующей новый API, в выпусках серии 1.x). Все приведенные примеры доступны в версии со старым API (в пакете oldapi) на сайте книги.

В тех случаях, где между двумя API существуют ощутимые различия, они обсуждаются в тексте.

Совместимость

Прежде чем переходить с одной версии на другую, важно принять во внимание все необходимые действия по обновлению. При этом требуется учесть следующие аспекты: совместимость API, совместимость данных и канальная совместимость.

Концепция совместимости API относится к контракту между пользовательским кодом и опубликованными API Hadoop — например, Java MapReduce API. Основным (major) выпускам (например, при переходе с 1.x.y на 2.0.0) разрешено нарушать совместимость API, что может потребовать изменения и перекомпилирования пользовательских программ. Дополнительные выпуски (например, при переходе с 1.0.x на 1.1.0) и точечные выпуски (например, при переходе с 1.0.1 на 1.0.2) не должны нарушать совместимость¹.

Концепция совместимости данных относится к форматам данных длительного хранения и метаданных — например, к формату, в котором узел имен HDFS сохраняет свои данные длительного хранения. Форматы могут изменяться между основными и дополнительными выпусками, но эти изменения прозрачны для пользователей, потому что обновление сопровождается автоматической миграцией данных. При этом могут действовать какие-то ограничения, относящиеся

¹ Для версий, предшествующих 1.0, действуют правила основных версий. Таким образом, переход с версии 0.1.0 на 0.2.0 считается выпуском новой основной версии, а следовательно, может нарушать совместимость API.

к путям обновления; они описаны в примечаниях к выпуску. Например, может потребоваться обновление через промежуточный выпуск вместо прямого перехода к финальному выпуску за один шаг. Обновления Hadoop более подробно обсуждаются в разделе «Обновления» на с. 463.



Hadoop использует схему классификации элементов API для обозначения их стабильности. Приведенные выше правила совместимости API распространяются на элементы с аннотацией `InterfaceStability.Stable`. С другой стороны, некоторые элементы открытых API Hadoop снабжены аннотацией `InterfaceStability.Evolving` или `InterfaceStability.Unstable` (все эти аннотации входят в пакет `org.apache.hadoop.classification`), это означает, что им разрешены нарушения совместимости в дополнительных и точечных выпусках соответственно.

Концепция канальной совместимости относится к взаимодействию между клиентами и серверами через канальные протоколы — такие, как RPC и HTTP. Клиенты делятся на два типа: внешние (запускаемые пользователями) и внутренние (запускаемые в кластерах как часть системы — как, например, демоны `datanode` и `tasktracker`). Как правило, внутренние клиенты должны обновляться согласованно; например, старая версия `tasktracker` не будет работать с новой версией `jobtracker`. Возможно, в будущем будет поддерживаться механизм последовательных (*rolling*) обновлений, чтобы кластерные демоны могли обновляться по фазам, а кластер оставался доступным для внешних клиентов во время обновления.

Внешние клиенты, запускаемые пользователями (например, программа, выполняющая чтение и запись из HDFS, или клиент отправки заданий MapReduce), должны иметь тот же основной номер версии, что и сервер, но могут иметь меньший номер дополнительного или точечного выпуска (например, клиент версии 1.0.1 будет работать с сервером 1.0.2 или 1.1.0, но не с сервером 2.0.0). Любое исключение из этого правила должно быть оговорено в примечаниях к выпуску.

2 MapReduce

MapReduce — модель программирования, ориентированная на обработку данных. Эта модель проста, но не настолько, чтобы в ее контексте нельзя было реализовать полезные программы. Hadoop позволяет запускать программы MapReduce, написанные на разных языках; в этой главе мы рассмотрим одну и ту же программу, написанную на языках Java, Ruby, Python и C++. Но самое важное заключается в том, что программы MapReduce параллельны по своей природе, а следовательно, крупномасштабный анализ данных становится доступным для всех, у кого в распоряжении имеется достаточно компьютеров. Достоинства MapReduce в полной мере проявляются в работе с большими наборами данных, так что начнем с рассмотрения одного из таких наборов.

Набор метеорологических данных

В качестве примера мы напишем программу для анализа метеорологических данных. Метеорологические датчики ежечасно собирают данные во многих точках по всему земному шару. Собранные данные хорошо подходят для анализа средствами MapReduce, потому что они частично структурированы и ориентированы на работу с записями.

Формат данных

Используемые данные были получены в Национальном центре климатических данных (NCDC, <http://www.ncdc.noaa.gov/>). Данные хранятся в строчно-ориентированном формате ASCII, в котором каждая строка соответствует одной записи. Формат поддерживает разнообразные метеорологические элементы, многие из которых являются необязательными или содержат данные переменной длины. Простоты ради мы сосредоточимся на основных элементах (таких, как температура) — они присутствуют всегда и имеют фиксированную длину.

В листинге 2.1 приведен пример записи с выделением основных полей. Запись разбита на строки, каждая из которых содержит одно поле; в реальном файле поля упакованы в одну строку без ограничителей.

Листинг 2.1. Формат записи Национального центра климатических данных

```
0057
332130 # Идентификатор метеостанции BBC США
99999 # Идентификатор метеостанции метеослужбы армии и флота
19500101 # Дата наблюдения
0300 # Время наблюдения
4
+51317 # Широта (градусы x 1000)
+028783 # Долгота (градусы x 1000)
FM-12
+0171 # Высота над уровнем моря (метры)
99999
V020
320 # Направление ветра (градусы)
1 # Код качества
N
0072
1
00450 # Высота облачности (метры)
1 # Код качества
C
N
010000 # Расстояние видимости (метры)
1 # Код качества
N
9
-0128 # Температура воздуха (градусы Цельсия x 10)
1 # Код качества
-0139 # Температура точки росы (градусы Цельсия x 10)
1 # Код качества
```

```
10268 # Атмосферное давление (гектопаскали x 10)
1 # Код качества
```

Файлы данных упорядочены по дате и метеостанции. Для каждого года с 1901 до 2001 имеется отдельный каталог, в котором находится gzip-архив для каждой метеостанции с данными за этот год. Например, начало списка файлов для 1990 года выглядит так:

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

Так как данные собираются на десятках тысяч метеостанций, весь набор данных состоит из множества относительно маленьких файлов. Обычно данные проще и эффективнее обрабатываются в меньшем количестве относительно больших файлов, поэтому данные были предварительно обработаны так, чтобы данные за каждый год были объединены в один файл.

Анализ данных средствами Unix

Какая максимальная глобальная температура была зарегистрирована в наборе данных за каждый год? Сначала мы попробуем ответить на этот вопрос без использования Hadoop, так как полученная информация даст нам базовый уровень для оценки производительности и пригодится для проверки наших результатов.

Одним из классических инструментов обработки текстовых данных является *awk*. В листинге 2.2 приведен небольшой сценарий для вычисления максимальной температуры за каждый год.

Листинг 2.2. Программа поиска максимальной зарегистрированной температуры за год по данным Национального центра климатических данных

```
#!/usr/bin/env bash
for year in all/*
do
```

продолжение ↗

Листинг 2.2 (продолжение)

```
echo -ne `basename $year .gz`\t"
gunzip -c $year | \
awk '{ temp = substr($0, 88, 5) + 0;
      q = substr($0, 93, 1);
      if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
END { print max }'
done
```

Сценарий перебирает файлы с данными по годам, выводит год, а потом обрабатывает каждый файл при помощи awk. Сценарий awk извлекает из данных два поля: температуру воздуха и код качества. Температура воздуха преобразуется в целое число (для этого к ней прибавляется 0). Затем сценарий проверяет, действительно ли полученное значение (код 9999 в наборе данных NCDC обозначает отсутствующие данные) и не указывает ли код качества на ошибочность или ненадежность данных. Если данные достоверны, то значение сравнивается с максимумом, найденным на данный момент; в случае обнаружения нового максимума последний обновляется. Блок END, выполняемый после обработки всех строк в файле, выводит максимальное значение.

Начало выходных данных выглядит так:

```
% ./max_temperature.sh
1901    317
1902    244
1903    289
1904    256
1905    283
...
```

Температуры в исходном файле умножены на 10, так что максимальная температура за 1901 год составила 31,7°C (данных за начало века очень мало, поэтому этот результат правдоподобен). Полная обработка данных за один век заняла 42 минуты с одним проходом на одном экземпляре EC2 High-CPU Extra Large Instance.

Чтобы ускорить обработку данных, необходимо организовать параллельное выполнение частей программы. Теоретически это несложно: разные годы обрабатываются в разных процессах, с использованием всех аппаратных программных потоков на машине. Однако при таком подходе возникает ряд проблем.

Во-первых, разбиение работы на части равного размера не всегда просто или очевидно. В данном случае размеры файлов за разные годы сильно различаются, поэтому некоторые процессы завершатся намного раньше других. Даже если они получат дополнительную работу, время выполнения всей серии будет определяться временем обработки самого длинного файла. Другое, более эффективное (хотя

и требующее большей работы) решение — разбиение входных данных на блоки фиксированного размера с назначением каждого фрагмента процессу.

Во-вторых, объединение результатов независимых процессов может потребовать дополнительной обработки. В нашем случае результаты каждого года не зависят от других лет, а для получения выходных данных достаточно склеить все результаты, отсортированные по годам. При использовании решения с фрагментами фиксированного размера объединение потребует больших усилий. Например, данные некоторого года можно разбить на фрагменты, каждый из которых обрабатывается независимо от других. Мы получаем максимальную температуру по каждому фрагменту, поэтому завершающим шагом станет определение наибольшего из этих «частичных максимумов» за каждый год.

В-третьих, вычисления по-прежнему ограничиваются вычислительной мощностью одной машины. Если с имеющимся количеством процессоров лучшее время составляет 20 минут — все, ускорить уже не удастся. Кроме того, некоторые наборы данных могут превысить возможности одного компьютера. Когда мы начинаем использовать в вычислениях несколько машин, также приходится учитывать множество других факторов, главным образом относящихся к категории координации и надежности. Кто запускает основное задание? Что делать с процессами, в которых произошли сбои?

Итак, хотя параллельные вычисления возможны, на практике это дело хлопотное. Поручив все эти тонкости специализированной инфраструктуре — такой, как Hadoop, мы сильно упростим свою работу.

Анализ данных в Hadoop

Чтобы воспользоваться теми преимуществами параллельной обработки данных, которые перед нами открывает Hadoop, необходимо представить запрос в виде задания MapReduce. После некоторого локального, маломасштабного тестирования мы сможем запустить это задание в компьютерном кластере.

Отображение и свертка

Работа MapReduce основана на разбиении обработки данных на две фазы: фазу отображения (map) и фазу свертки (reduce). Каждая фаза использует в качестве входных и выходных данных пары «ключ-значение», типы которых выбираются программистом. Программист также задает две функции: функцию отображения и функцию свертки.

Входными данными для фазы отображения в нашем примере являются исходные данные NCDC. Мы выбираем текстовый формат входных данных, при котором каждая строка набора данных интерпретируется как текстовое значение. Ключом является смещение начала строки от начала файла, но так как эта информация нам не нужна, мы ее просто игнорируем.

Функция отображения устроена просто. Мы извлекаем год и температуру воздуха, потому что нас интересуют только эти поля. В данном случае функция отображения всего лишь готовит данные к использованию так, чтобы функция свертки могла выполнить свою работу: определение максимальной температуры за каждый год. Функция отображения также хорошо подходит для исключения нежелательных записей: здесь отфильтровываются отсутствующие, сомнительные или ошибочные значения температуры.

Чтобы представить, как работает функция отображения, рассмотрим несколько строк входных данных (неиспользуемые столбцы, опущенные для экономии места, обозначены многоточиями):

```
006701199099991950051507004...9999999N9+00001+9999999999...
004301199099991950051512004...9999999N9+00221+9999999999...
004301199099991950051518004...9999999N9-00111+9999999999...
004301265099991949032412004...0500001N9+01111+9999999999...
004301265099991949032418004...0500001N9+00781+9999999999...
```

Эти строки передаются функции отображения в виде пар «ключ-значение»:

```
(0, 006701199099991950051507004...9999999N9+00001+9999999999...)
(106, 004301199099991950051512004...9999999N9+00221+9999999999...)
(212, 004301199099991950051518004...9999999N9-00111+9999999999...)
(318, 004301265099991949032412004...0500001N9+01111+9999999999...)
(424, 004301265099991949032418004...0500001N9+00781+9999999999...)
```

Ключи (смещения строк в файле) в функции отображения игнорируются. Функция отображения просто извлекает год и температуру воздуха (выделенные жирным шрифтом) и передает их как выходные данные (значения температуры интерпретируются как целые числа):

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Выходные данные функции отображения обрабатываются инфраструктурой MapReduce перед тем, как они будут переданы функции свертки. В ходе этой

обработки пары «ключ-значение» сортируются и группируются по ключу. Таким образом, в нашем примере функция свертки получит следующие входные данные:

```
(1949, [111, 78])
(1950, [0, 22, -11])
```

Каждый год сопровождается списком его значений температуры. Теперь функции свертки остается лишь перебрать элементы списка и найти максимум:

```
(1949, 111)
(1950, 22)
```

Перед нами результат: максимальные глобальные температуры, зарегистрированные в каждом году.

Схема потока данных изображена на рис. 2.1. В нижней части диаграммы располагается конвейер Unix, повторяющий все операции MapReduce; мы еще вернемся к нему в этой главе при рассмотрении Hadoop Streaming.

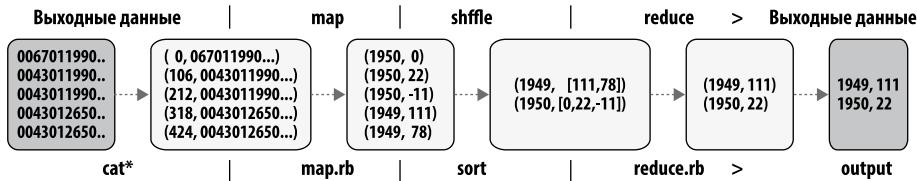


Рис. 2.1. Логическая схема потока данных MapReduce

Программа MapReduce на языке Java

Итак, мы разобрались с тем, как работает программа MapReduce. Следующим шагом должно стать ее выражение в виде программного кода. Нам понадобятся: функция отображения, функция свертки и код выполнения задания. Функция отображения представлена классом `Mapper`, объявляющим абстрактный метод `map()`. Реализация метода `map` представлена в листинге 2.3.

Листинг 2.3. Функция отображения для примера с максимальной температурой

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
```

продолжение ⇨

Листинг 2.3 (продолжение)

```
import org.apache.hadoop.mapreduce.Mapper;
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    private static final int MISSING = 9999;

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

        String line = value.toString();
        String year = line.substring(15, 19);
        int airTemperature;
        if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
            airTemperature = Integer.parseInt(line.substring(88, 92));
        } else {
            airTemperature = Integer.parseInt(line.substring(87, 92));
        }
        String quality = line.substring(92, 93);
        if (airTemperature != MISSING && quality.matches("[01459]")) {
            context.write(new Text(year), new IntWritable(airTemperature));
        }
    }
}
```

Класс `Mapper` является параметризованным; его четыре параметра типов определяют типы входного ключа, входного значения, выходного ключа и выходного значения функции отображения. В текущем примере входной ключ представляет собой длинное целочисленное смещение, входное значение — строку текста, выходной ключ — год и выходное значение — температуру воздуха (целое число). Вместо использования встроенных типов Java Hadoop предоставляет собственные базовые типы, оптимизированные для сетевой сериализации. Они находятся в пакете `org.apache.hadoop.io`. В нашем примере используются типы `LongWritable` (аналог типа Java `Long`), `Text` (аналог Java `String`) и `IntWritable` (аналог Java `Integer`).

Метод `map()` получает ключ и значение. Значение `Text`, содержащее входную строку, преобразуется в тип Java `String`, после чего метод `substring()` извлекает интересующие нас столбцы.

Метод `map()` также предоставляет экземпляр `Context` для записи выходных данных. В нашем случае год записывается в виде объекта `Text` (так как он используется только в качестве ключа), а температура упаковывается в тип `IntWritable`. Выход записывается только в том случае, если значение температуры присутствует, а код качества указывает на то, что данные температуры действительны.

Функция `reduce` определяется аналогичным образом с использованием класса `Reducer`, как показано в листинге 2.4.

Листинг 2.4. Функция свертки для примера с максимальной температурой

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

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

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}
```

Как и в предыдущем случае, четыре формальных параметра определяют типы входных и выходных данных — на этот раз для функции свертки. Входные типы функции свертки должны соответствовать выходным типам функции отображения: `Text` и `IntWritable`. Выходными типами функции свертки являются `Text` и `IntWritable` — для года и максимальной температуры, определяемой перебором температур и сравнением элементов с текущим найденным максимумом.

Третий фрагмент кода запускает задание `MapReduce` (листинг 2.5).

Листинг 2.5. Приложение для определения максимальной температуры в наборе данных

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

продолжение ↗

Листинг 2.5 (продолжение)

```
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperature <input path> <output path>");
            System.exit(-1);
        }

        Job job = new Job();
        job.setJarByClass(MaxTemperature.class);
        job.setJobName("Max temperature");

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

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setReducerClass(MaxTemperatureReducer.class);

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

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

Объект `Job` формирует спецификацию задания и позволяет управлять его выполнением. При запуске этого задания в кластере Hadoop код упаковывается в файл JAR (который Hadoop распределяет по кластеру). Вместо того, чтобы явно указывать имя файла JAR, мы можем передать класс методу `setJarByClass()` класса `Job`, а Hadoop находит файл JAR, содержащий указанный класс.

После создания объекта `Job` задаются пути к входным и выходным данным (или проще, входные и выходные пути). Входной путь определяется вызовом статического метода `addInputPath()` для объекта `FileInputFormat`; он может определять отдельный файл, каталог (в этом случае входные данные состоят из всех файлов в заданном каталоге) или шаблон. Как нетрудно догадаться по названию, метод `addInputPath()` может вызываться повторно для добавления входных данных из нескольких путей. Выходной путь (который может быть только один) задается статическим методом `setOutputPath()` объекта `FileOutputFormat`. Он задает каталог, в который записываются выходные файлы функций свертки. Каталог не должен существовать перед запуском задания, в противном случае Hadoop выдаст

предупреждающее сообщение и не запустит задание. Это делается для предотвращения возможной потери данных (очень обидно случайно перезаписать выходные данные долго выполнявшегося задания другими данными).

Затем типы отображения и свертки задаются методами `setMapperClass()` и `setReducerClass()`.

Методы `setOutputKeyClass()` и `setOutputValueClass()` управляют выходными типами функций отображения и свертки, которые часто совпадают (как в нашем случае). Если типы различаются, то выходные типы отображения задаются методами `setMapOutputKeyClass()` и `setMapOutputValueClass()`.

Входные типы определяются через формат входных данных, который мы не задаем явно, поскольку в нашем примере используется формат по умолчанию `TextInputFormat`.

После настройки классов, определяющих функции отображения и свертки, можно переходить к выполнению задания. Метод `waitForCompletion()` класса `Job` передает задание на выполнение и ожидает его завершения. Логический аргумент метода определяет флаг расширенного вывода; в нашем случае информация о ходе выполнения задания выводится на консоль.

Возвращаемое значение метода `waitForCompletion()` представляет собой логический признак успеха (`true`) или неудачи (`false`), который преобразуется в код завершения программы 0 или 1.

Тестовый запуск

Готовое задание MapReduce стоит опробовать на небольшом наборе данных, чтобы сразу выявить все очевидные проблемы с кодом. Для начала установите Hadoop в автономном режиме. В этом режиме Hadoop работает с локальной файловой системой и локальной системой выполнения заданий. Установите и откомпилируйте примеры по инструкциям на сайте книги.

Протестируем наше задание на приведенном выше подмножестве из пяти строк (результат отформатирован по ширине страницы):

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
12/02/04 11:50:41 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
12/02/04 11:50:41 WARN mapred.JobClient: Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
```

продолжение ↗

```
12/02/04 11:50:41 INFO input.FileInputFormat: Total input paths to process : 1
12/02/04 11:50:41 INFO mapred.JobClient: Running job: job_local_0001
12/02/04 11:50:41 INFO mapred.Task: Using ResourceCalculatorPlugin : null
12/02/04 11:50:41 INFO mapred.MapTask: io.sort.mb = 100
12/02/04 11:50:42 INFO mapred.MapTask: data buffer = 79691776/99614720
12/02/04 11:50:42 INFO mapred.MapTask: record buffer = 262144/327680
12/02/04 11:50:42 INFO mapred.MapTask: Starting flush of map output
12/02/04 11:50:42 INFO mapred.MapTask: Finished spill 0
12/02/04 11:50:42 INFO mapred.Task: Task:attempt_local_0001_m_000000_0
                                is done. And i
                                s in the process of committing
12/02/04 11:50:42 INFO mapred.JobClient: map 0% reduce 0%
12/02/04 11:50:44 INFO mapred.LocalJobRunner:
12/02/04 11:50:44 INFO mapred.Task: Task 'attempt_local_0001_m_000000_0' done.
12/02/04 11:50:44 INFO mapred.Task: Using ResourceCalculatorPlugin : null
12/02/04 11:50:44 INFO mapred.LocalJobRunner:
12/02/04 11:50:44 INFO mapred.Merger: Merging 1 sorted segments
12/02/04 11:50:44 INFO mapred.Merger: Down to the last merge-pass,
                                with 1 segments
left of total size: 57 bytes
12/02/04 11:50:44 INFO mapred.LocalJobRunner:
12/02/04 11:50:45 INFO mapred.Task: Task:attempt_local_0001_r_000000_0
                                is done. And
                                is in the process of committing
12/02/04 11:50:45 INFO mapred.LocalJobRunner:
12/02/04 11:50:45 INFO mapred.Task: Task attempt_local_0001_r_000000_0
                                is allowed to
commit now
12/02/04 11:50:45 INFO output.FileOutputCommitter: Saved output of task
                                'attempt_local
                                _0001_r_000000_0' to output
12/02/04 11:50:45 INFO mapred.JobClient: map 100% reduce 0%
12/02/04 11:50:47 INFO mapred.LocalJobRunner: reduce > reduce
12/02/04 11:50:47 INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done.
12/02/04 11:50:48 INFO mapred.JobClient: map 100% reduce 100%
12/02/04 11:50:48 INFO mapred.JobClient: Job complete: job_local_0001
12/02/04 11:50:48 INFO mapred.JobClient: Counters: 17
12/02/04 11:50:48 INFO mapred.JobClient: File Output Format Counters
12/02/04 11:50:48 INFO mapred.JobClient: Bytes Written=29
12/02/04 11:50:48 INFO mapred.JobClient: FileSystemCounters
12/02/04 11:50:48 INFO mapred.JobClient: FILE_BYTES_READ=357503
12/02/04 11:50:48 INFO mapred.JobClient: FILE_BYTES_WRITTEN=425817
12/02/04 11:50:48 INFO mapred.JobClient: File Input Format Counters
12/02/04 11:50:48 INFO mapred.JobClient: Bytes Read=529
```

```
12/02/04 11:50:48 INFO mapred.JobClient: Map-Reduce Framework
12/02/04 11:50:48 INFO mapred.JobClient: Map output materialized bytes=61
12/02/04 11:50:48 INFO mapred.JobClient: Map input records=5
12/02/04 11:50:48 INFO mapred.JobClient: Reduce shuffle bytes=0
12/02/04 11:50:48 INFO mapred.JobClient: Spilled Records=10
12/02/04 11:50:48 INFO mapred.JobClient: Map output bytes=45
12/02/04 11:50:48 INFO mapred.JobClient: Total committed heap
12/02/04 11:50:48 INFO mapred.JobClient: usage (bytes)=36923

8016
12/02/04 11:50:48 INFO mapred.JobClient: SPLIT_RAW_BYTES=129
12/02/04 11:50:48 INFO mapred.JobClient: Combine input records=0
12/02/04 11:50:48 INFO mapred.JobClient: Reduce input records=5
12/02/04 11:50:48 INFO mapred.JobClient: Reduce input groups=2
12/02/04 11:50:48 INFO mapred.JobClient: Combine output records=0
12/02/04 11:50:48 INFO mapred.JobClient: Reduce output records=2
12/02/04 11:50:48 INFO mapred.JobClient: Map output records=5
```

При вызове с передачей имени класса в первом аргументе команда `hadoop` запускает виртуальную машину Java (JVM) для выполнения класса. Команда `hadoop` удобнее прямого запуска `java`, поскольку она включает в путь классов библиотеки Hadoop (и их зависимости) и выбирает конфигурацию Hadoop. Чтобы добавить в путь классов классы приложения, мы определяем переменную окружения с именем `HADOOP_CLASSPATH`, которая используется сценарием `hadoop`.



При выполнении в локальном (автономном) режиме все программы, приведенные в книге, предполагают, что переменная `HADOOP_CLASSPATH` была задана тем или иным образом. Команда должна выполняться из каталога, в котором установлен исходный код.

Выходные данные, полученные при выполнении задания, содержат полезную информацию. Например, мы видим, что заданию присвоен идентификатор `job_local_0001` и оно выполнило одну задачу отображения и одну задачу свертки (с идентификаторами `attempt_local_0001_m_000000_0` и `attempt_local_0001_r_000000_0`). Идентификаторы задания и задач сильно пригодятся при отладке заданий MapReduce.

Последняя секция вывода («Counters») содержит статистику, сгенерированную Hadoop для каждого выполненного задания. По этим данным можно проверить, совпадает ли объем обработанных данных с ожидаемым. Например, можно отследить количество записей, обработанных системой: для пяти входных записей функции отображения создано пять выходных записей, а для пяти входных записей функции свертки создано две выходные записи.

Вывод записывается в каталог *output*; для каждой функции свертки создается один выходной файл. Задание имеет одну функцию свертки, поэтому в каталоге создается один файл с именем *part-r-00000*:

```
% cat output/part-r-00000
1949    111
1950    22
```

Результат совпадает с тем, который был получен ранее при ручном выполнении. В 1949 году была зарегистрирована максимальная температура 11,1°C, а в 1950 – 2,2°C.

Старый и новый Java API MapReduce

Java API MapReduce, использованный в предыдущем разделе, появился впервые в версии 0.20.0. Он был разработан для упрощения возможного расширения в будущем. Однако новый API несовместим со старым, поэтому для его использования приложения придется переписывать.

Новый API в основном завершен в последних выпусках серии 1.x (продолжение серии 0.20), не считая нескольких отсутствующих библиотек MapReduce (проверьте доступность библиотеки, которую вы хотите использовать, в подпакете `org.apache.hadoop.mapreduce.lib` последней версии).

Предыдущие издания этой книги были основаны на серии 0.20, и в них использовался старый API. В этом издании в основном используется новый API (за исключением нескольких мест). Но если вы предпочитаете использовать старый API, код во всех примерах со старым API доступен на веб-сайте книги. (В нескольких ранних выпусках 0.20 старый API был объявлен не рекомендуемым к использованию, но это ограничение было снято в последующих выпусках, так что теперь выпуски 1.x и 2.x поддерживают как старый, так и новый API без выдачи предупреждений.)

Наиболее заметные различия между двумя API:

- Новый API предпочтает абстрактные классы интерфейсам, так как они упрощают дальнейшую эволюцию кода. Это означает, что в абстрактный класс можно добавить метод (с реализацией по умолчанию) без нарушения работоспособности старых реализаций класса¹. Например, интерфейсы `Mapper` и `Reducer` старого API преобразованы в абстрактные классы.

¹ Строго говоря, такое изменение почти наверняка нарушит работоспособность реализаций, уже определяющих метод с такой же сигнатурой, как у нового метода, но, как объясняется в статье http://wiki.eclipse.org/Evolving_Java-based_APIs#Example_4_-_Adding_an_API_method, в практическом контексте такое изменение следует рассматривать как совместимое.

- Новый API находится в пакете `org.apache.hadoop.mapreduce` (и его подпакетах). Старый API по-прежнему находится в пакете `org.apache.hadoop.mapred`.
- В новом API широко используются контекстные объекты, позволяющие пользовательскому коду взаимодействовать с системой MapReduce. Например, новый объект `Context`, по сути, объединяет роли объектов `JobConf`, `OutputCollector` и `Reporter` из старого API.
- В обоих API пары «ключ-значение» передаются функциям отображения и свертки, но кроме того, новый API позволяет как функциям отображения, так и функциям свертки управлять процессом выполнения посредством переопределения метода `run()`. Например, записи могут обрабатываться пакетами, или же выполнение может быть прервано до полной обработки всех записей. В старом API такая возможность была доступна функциям отображения при использовании `MapRunnable`, но эквивалентной возможности для функций свертки не существовало.
- Управление заданиями в новом API осуществляется через класс `Job`, а не через старый класс `JobClient`, отсутствующий в новом API.
- Унификация конфигурации в новом API. В старом API конфигурация заданий определялась специальным объектом `JobConf`, который представляет собой расширение объекта Hadoop `Configuration` (используемого для конфигурации демонов; см. «API конфигурации» на с. 203). В новом API конфигурация заданий определяется через `Configuration` с возможным использованием вспомогательных методов `Job`.
- Имена выходных файлов выглядят немного иначе. В старом API выходные файлы функций отображения и свертки назывались `part-nnnnnn`; в новом API выходные файлы функции отображения называются `part-m-nnnnnn`, а выходные файлы функции свертки — `part-r-nnnnnn` (где `nnnnn` — номер части, нумерация начинается с нуля).
- Методы, переопределяемые пользователем, в новом API объявляются как выдающие исключение `java.lang.InterruptedException`. Таким образом, вы можете написать собственный код обработки прерывания, чтобы инфраструктура при необходимости могла корректно отменять затянувшиеся операции¹.
- В новом API метод `reduce()` передает значения в виде `java.lang.Iterable` (а не `java.lang.Iterator`, как в старом API). Это изменение позволяет легко перебирать значения в цикле Java `for/each`:

```
for (VALUEIN value : values) { ... }
```

¹ Данная возможность подробно описана в статье Брайана Гетца (Brian Goetz) «Dealing with `InterruptedException`».

В листинге 2.6 приведено приложение MaxTemperature, переписанное для старого API. Различия выделены жирным шрифтом.



Преобразуя свои классы Mapper и Reducer для нового API, не забудьте изменить сигнатуру методов map() и reduce(). Простое изменение класса с расширением нового класса Mapper или Reducer не приведет к выдаче ошибки компилятора или предупреждения, потому что эти классы представляют тождественную форму метода map() или reduce() (соответственно). Однако при этом ваш код функции отображения или свертки вызван не будет, что приведет к появлению трудноуловимых ошибок.

Пометка методов map() и reduce() аннотацией @Override поможет компилятору Java обнаружить эти ошибки.

Листинг 2.6. Приложение для определения максимальной температуры с использованием старого MapReduce API

```
public class OldMaxTemperature {  
  
    static class OldMaxTemperatureMapper extends MapReduceBase  
        implements Mapper<LongWritable, Text, Text, IntWritable> {  
  
        private static final int MISSING = 9999;  
  
        @Override  
        public void map(LongWritable key, Text value,  
                        OutputCollector<Text, IntWritable> output, Reporter reporter)  
            throws IOException {  
  
            String line = value.toString();  
            String year = line.substring(15, 19);  
            int airTemperature;  
            if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs  
                airTemperature = Integer.parseInt(line.substring(88, 92));  
            } else {  
                airTemperature = Integer.parseInt(line.substring(87, 92));  
            }  
            String quality = line.substring(92, 93);  
            if (airTemperature != MISSING && quality.matches("[01459]")) {  
                output.collect(new Text(year), new IntWritable(airTemperature));  
            }  
        }  
  
        static class OldMaxTemperatureReducer extends MapReduceBase
```

```
implements Reducer<Text, IntWritable, Text, IntWritable> {
@Override
public void reduce(Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output, Reporter reporter)
    throws IOException {

    int maxValue = Integer.MIN_VALUE;
    while (values.hasNext()) {
        maxValue = Math.max(maxValue, values.next().get());
    }
    output.collect(key, new IntWritable(maxValue));
}

public static void main(String[] args) throws IOException {
    if (args.length != 2) {
        System.err.println("Usage: OldMaxTemperature <input path> <output path>");
        System.exit(-1);
    }

    JobConf conf = new JobConf(OldMaxTemperature.class);
    conf.setJobName("Max temperature");

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

    conf.setMapperClass(OldMaxTemperatureMapper.class);
    conf.setReducerClass(OldMaxTemperatureReducer.class);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    JobClient.runJob(conf);
}
}
```

MapReduce в перспективе

Вы видели, как MapReduce работает для малых входных данных; пришло время взглянуть на систему «издалека» и рассмотреть, как происходит передача данных для больших входных наборов. Для простоты в предыдущих примерах использовались файлы локальной файловой системы. Чтобы картина была полноценной,

данные должны храниться в распределенной файловой системе (обычно в системе HDFS, о которой вы узнаете в следующей главе). Это позволит Hadoop разместить вычисления MapReduce на каждом компьютере, обрабатывающем свою часть данных. Давайте посмотрим, как работает эта схема.

Поток данных

Для начала немного терминологии. *Задание* (job) MapReduce представляет собой единицу работы, которую хочет выполнить клиент: оно состоит из входных данных, программы MapReduce и конфигурационной информации. Чтобы выполнить задание, Hadoop разбивает его на *задачи* (tasks), которые делятся на два типа: задачи отображения и задачи свертки.

Узлы, управляющие процессом выполнения задания, делятся на два типа: *трекер заданий* (jobtracker) и несколько *трекеров задач* (tasktracker). Трекер заданий координирует все задания, выполняемые системой; для этого он планирует выполнение задач на трекерах задач. Трекеры задач выполняют задачи и отправляют отчеты о ходе работы трекеру заданий, который отслеживает общий прогресс каждого задания. Если попытка выполнения задачи завершается неудачей, трекер может заново спланировать ее на другом трекере.

Hadoop делит входные данные заданий MapReduce на фрагменты фиксированного размера, называемые *сплитами* (splits). Hadoop создает для каждого сплита одну задачу отображения, которая выполняет определенную пользователем функцию отображения для каждой записи в сплите.

Наличие многих сплитов означает, что время, затраченное на обработку каждого сплита, относительно мало по сравнению с временем обработки всех входных данных. Таким образом, если сплиты будут обрабатываться параллельно, нагрузка будет лучше сбалансирована при малом размере сплитов, поскольку более быстрая машина сможет обработать больше сплитов, чем медленная. Даже если машины идентичны, сбойные процессы или другие одновременно выполняемые задания заставляют обратить внимание на распределение нагрузки, а качество распределения нагрузки возрастает с уменьшением сплитов.

С другой стороны, при слишком малом размере сплита затраты на управление сплитами и создание задач отображения занимают слишком большую долю общего времени выполнения. Для большинства заданий желательный размер сплита обычно соответствует размеру блока HDFS — 64 Мбайт по умолчанию, хотя эту величину можно изменить для кластера (для всех вновь создаваемых файлов) или задать при создании файла.

Hadoop старается по возможности выполнять задачи отображения в узле, в котором входные данные хранятся в HDFS. Этот принцип, называемый *оптимизацией*

локальности данных, стремится снизить нагрузку на цennую пропускную способность кластера. Тем не менее в некоторых случаях все три узла, на которых размещены реплики блока HDFS с входным сплитом задачи отображения, заняты выполнением других задач отображения; тогда планировщик заданий ищет свободный слот отображения на узле в том же сегменте (rack), что и один из блоков. Крайне редко даже такое решение оказывается невозможным, и тогда используется внесегментный узел, что приводит к межсегментной передаче данных по сети. Эти три возможности представлены на рис. 2.2.

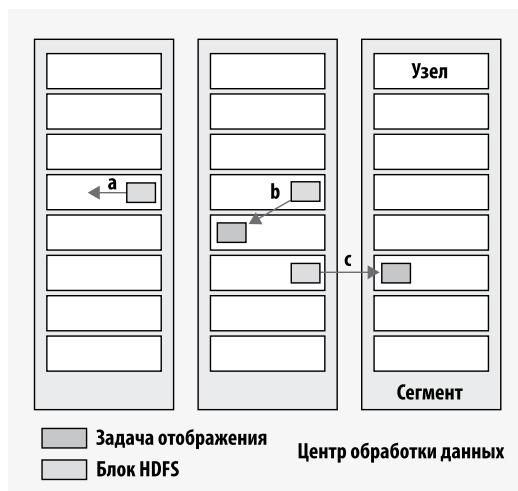


Рис. 2.2. Задачи отображения: локальность данных (a),
сегментная локальность (b), внесегментная задача (c)

Понятно, почему оптимальный размер сплита совпадает с размером блока: это максимальный размер данных, которые могут гарантированно храниться на одном узле. Если сплит занимает два блока, вряд ли найдется узел HDFS, на котором хранятся оба блока, и часть сплита придется передавать по сети узлу, на котором выполняется задача отображения. Конечно, в этом случае выполнение станет менее эффективным, чем при выполнении всей задачи отображения с использованием локальных данных.

Задачи отображения записывают свои выходные данные на локальный диск, а не в HDFS. Почему? Вывод отображений является промежуточным: он обрабатывается задачами свертки для получения итогового вывода, и после завершения задания вывод задач отображения можно удалить. Таким образом, хранение его в HDFS с репликацией неэффективно. Если на узле, на котором работает задача

отображения, произойдет сбой до того, как вывод отображения будет получен задачей свертки, Hadoop автоматически заново выполнит задачу отображения на другом узле, чтобы воссоздать вывод задачи отображения.

Задачи свертки не пользуются преимуществами локальности данных; входные данные одной задачи свертки обычно образуются из выходных данных всех задач отображения. В нашем примере одна задача свертки получает данные от всех задач отображения. Таким образом, отсортированный вывод отображений должен быть передан по сети узлу, на котором работает задача свертки; там данные объединяются и передаются функции свертки, определенной пользователем. Вывод свертки обычно хранится в HDFS по соображениям надежности. Как объясняется в главе 3, для каждого блока HDFS, содержащего выходные данные свертки, первая реплика хранится в локальном узле, а другие реплики хранятся во внесегментных узлах. Соответственно, запись вывода свертки приводит к расходованию пропускной способности сети, но не больше, чем для обычного конвейера записи HDFS.

Весь поток данных для одной задачи свертки изображен на рис. 2.3. Пунктирные прямоугольники обозначают узлы, пунктируемые стрелки — передачу данных узлам, а жирные стрелки — передачу данных между узлами.

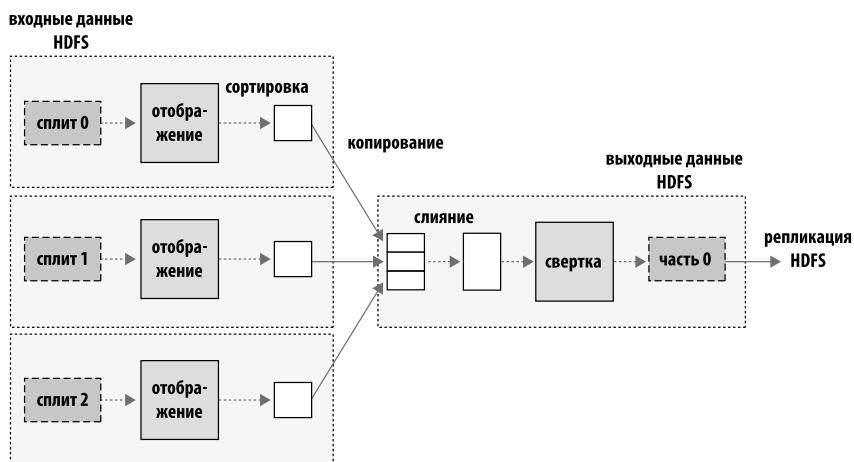


Рис. 2.3. Поток данных MapReduce для одной задачи свертки

Количество задач свертки не определяется размером входных данных, а задается независимо. В разделе «Задание MapReduce по умолчанию» на с. 299 вы узнаете, как выбрать количество задач свертки для задания.

При наличии нескольких функций свертки задачи отображения разделяют свои выходные данные, при этом каждая создает один раздел (partition) для одной

задачи свертки. В каждом разделе может быть много ключей (и связанных с ними значений), но все записи для заданного ключа находятся в одном разделе. Разделением можно управлять при помощи функции, определяемой пользователем, но обычно стандартная реализация разделения (с хеш-функцией) работает очень хорошо.

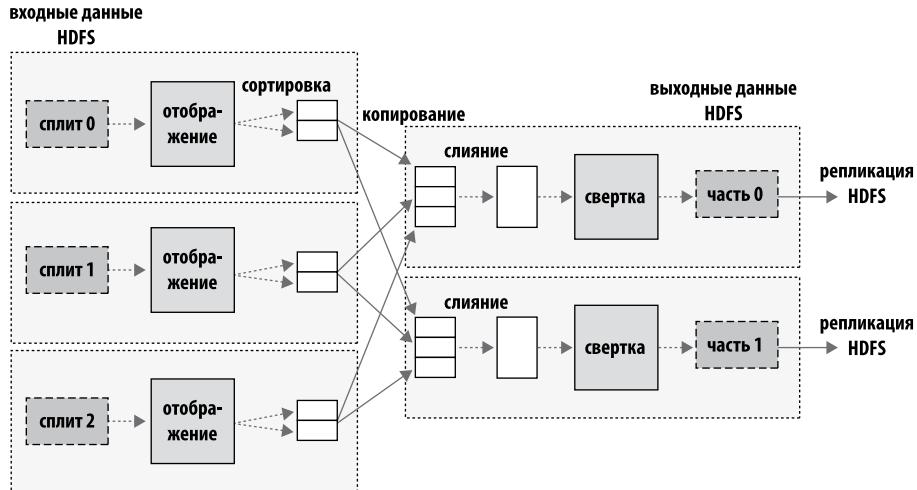


Рис. 2.4. Поток данных MapReduce для нескольких задач свертки

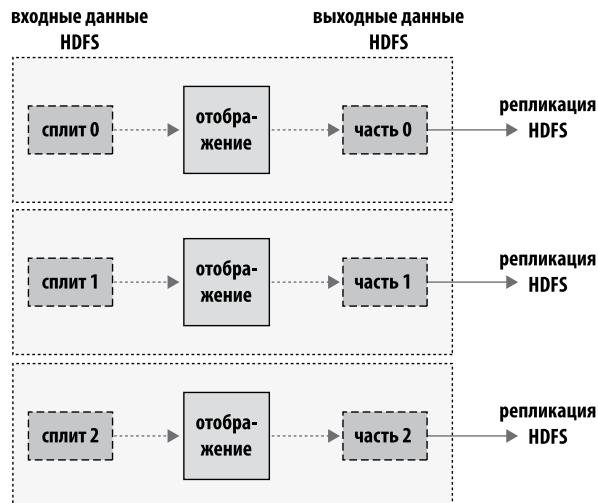


Рис. 2.5. Поток данных MapReduce без задач свертки

Поток данных для общего случая нескольких задач свертки показан на рис. 2.4. Из диаграммы ясно видно, почему поток данных между задачами свертки и отображения известен под неформальным названием «тасовка» (shuffle) — каждая задача свертки получает данные от многих задач отображения. Процесс тасовки сложнее, чем может показаться из диаграммы, а его оптимизация может оказать большое влияние на время выполнения задания, как вы увидите в разделе «Тасовка и сортировка» на с. 279.

Наконец, задание может содержать нуль задач свертки. Это может быть уместно, когда тасовка не нужна, потому что обработка осуществляется полностью параллельно (примеры рассматриваются в разделе «NLineInputFormat» на с. 324). В этом случае внеузловая пересылка данных происходит только при выполнении записи задачами отображения в HDFS (рис. 2.5).

Комбинирующие функции

Многие задания MapReduce ограничиваются по пропускной способности, доступной в кластере, поэтому передачу данных между задачами отображения и свертки желательно свести к минимуму. Hadoop позволяет пользователю задать *комбинирующую функцию*, которая будет выполняться для выходных данных отображения; выходные данные комбинирующей функции образуют ввод функции свертки. Так как комбинирующая функция является оптимизацией, Hadoop не предоставляет гарантий относительно того, сколько раз она будет вызвана для конкретной записи выходных данных отображения (и будет ли вызвана вообще). Иначе говоря, при вызове комбинирующей функции нуль, один или несколько раз функция свертки должна выдавать одинаковый результат.

Контракт комбинирующей функции ограничивает тип используемой функции. Лучше всего пояснить это на примере. Предположим, в нашем примере с максимальными температурами данные за 1950 год обрабатывались двумя отображениями (потому что находились в разных сплитах). Первое отображение выдало следующие данные:

```
(1950, 0)  
(1950, 20)  
(1950, 10)
```

Выходные данные второго отображения:

```
(1950, 25)  
(1950, 15)
```

Функция свертки будет вызвана со списком всех значений:

```
(1950, [0, 20, 10, 25, 15])
```

Результатом будет запись

`(1950, 25),`

так как 25 — максимальное значение в списке. Мы могли бы использовать комбинирующую функцию, которая, как и функция свертки, находит максимальную температуру для вывода каждого отображения. В этом случае функция свертки будет вызвана с данными:

`(1950, [20, 25]),`

Функция свертки выдаст тот же результат, что и прежде. В более формальной записи вызовы функций для значений температур в этом случае могут быть выражены в следующем виде:

`max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25.`

Не все функции обладают этим свойством¹. Например, при вычислении средних температур мы не смогли бы использовать среднее арифметическое как комбинирующую функцию, потому что

`mean(0, 20, 10, 25, 15) = 14,`

но

`mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15.`

Комбинирующая функция не заменяет функцию свертки. (Да и как она могла бы ее заменить? Функция свертки все равно необходима для обработки записей с одним ключом от разных отображений.) Но она может сократить объем данных, передаваемых между функциями отображения и свертки, и уже по одной этой причине всегда стоит рассмотреть возможность использования комбинирующей функции в задании MapReduce.

Определение комбинирующей функции

Вернемся к программам MapReduce на языке Java: комбинирующая функция определяется при помощи класса `Reducer` и в нашем приложении имеет такую же реализацию, как и функция свертки в `MaxTemperatureReducer`. Далее остается только назначить класс комбинирующей функции в `Job` (листинг 2.7).

¹ Функции, обладающие этим свойством, называются коммутативными и ассоциативными. Иногда они также называются дистрибутивными, как в статье «Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals», Gray et al. (1995).

Листинг 2.7. Приложение для определения максимальной температуры с использованием комбинирующей функции для эффективности

```
public class MaxTemperatureWithCombiner {  
  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +  
                "<output path>");  
            System.exit(-1);  
        }  
  
        Job job = new Job();  
        job.setJarByClass(MaxTemperatureWithCombiner.class);  
        job.setJobName("Max temperature");  
  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.setMapperClass(MaxTemperatureMapper.class);  
        job.setCombinerClass(MaxTemperatureReducer.class);  
        job.setReducerClass(MaxTemperatureReducer.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        System.exit(job.waitForCompletion(true) ? 0 : 1);  
    }  
}
```

Выполнение распределенного задания MapReduce

Та же программа будет работать без каких-либо изменений с полным набором данных. Для этого и нужна технология MapReduce: она масштабируется по размерам данных и оборудования. На 10-узловом кластере EC2 с экземплярами High-CPU Extra Large Instance выполнение программы заняло шесть минут¹.

Механика запуска программ в кластерах будет рассмотрена в главе 5.

¹ То есть в 7 раз быстрее, чем при последовательном выполнении на одном компьютере с использованием *awk*. Основная причина, по которой ускорение не было пропорциональным, заключается в том, что входные данные не были равномерно распределены. Для удобства входные файлы заархивированы по годам; файлы последних лет, с гораздо большим объемом зарегистрированных метеорологических данных, имеют существенно больший размер.

Hadoop Streaming

Hadoop предоставляет API для MapReduce, позволяющий записывать функции отображения и свертки на других языках, кроме Java. Технология Hadoop Streaming использует стандартный поток Unix для организации взаимодействия Hadoop с программами, так что при написании программ MapReduce можно использовать любой язык, поддерживающий чтение из стандартного входного потока (стандартного ввода) и запись в стандартный выходной поток (стандартный вывод). Функция свертки читает строки из стандартного ввода (которые, как гарантирует инфраструктура, отсортированы по ключу) и записывает свои результаты в стандартный вывод.

Чтобы продемонстрировать, как работает Hadoop Streaming, мы перепишем нашу программу MapReduce для поиска максимальных температур.

Ruby

Примерная реализация функции отображения на языке Ruby представлена в листинге 2.8.

Листинг 2.8. Функция отображения для определения максимальной температуры на языке Ruby

```
#!/usr/bin/env ruby
STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

Программа перебирает строки из стандартного ввода, выполняя программный блок для каждой строки из STDIN (глобальная константа типа IO). Блок извлекает нужные поля из каждой строки ввода и, если температура действительна, записывает год и температуру, разделенные символом табуляции \t, в стандартный вывод (при помощи функции puts).

Стоит отметить важное архитектурное различие между Streaming и Java MapReduce API. Java API ориентирован на последовательную обработку записей функцией отображения. Инфраструктура вызывает метод map() вашей реализации Mapper для каждой записи во входном наборе, тогда как при использовании Streaming программа отображения сама решает, как обрабатывать ввод — например, она может легко прочитать и обработать сразу несколько строк, поскольку процесс чтения находится под ее контролем. Пользовательская реализация отображения

на Java последовательно получает записи, но при этом она все равно может обрабатывать сразу несколько записей, накапливая предыдущие строки в переменной экземпляра класса `Mapper`¹. В этом случае необходимо реализовать метод `close()`, чтобы вы узнали о том, что была прочитана последняя запись, и завершили обработку последней группы строк.

Так как сценарий просто выполняет операции со стандартным вводом и выводом, его можно элементарно протестировать без использования Hadoop:

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb
1950    +0000
1950    +0022
1950    -0011
1949    +0111
1949    +0078
```

Функция свертки, приведенная в листинге 2.9, получается чуть более сложной.

Листинг 2.9. Функция свертки для определения максимальной температуры на языке Ruby

```
#!/usr/bin/env ruby

last_key, max_val = nil, -1000000
STDIN.each_line do |line|
  key, val = line.split("\t")
  if last_key && last_key != key
    puts "#{last_key}\t#{max_val}"
    last_key, max_val = key, val.to_i
  else
    last_key, max_val = key, [max_val, val.to_i].max
  end
end
puts "#{last_key}\t#{max_val}" if last_key
```

Программа также перебирает строки из стандартного ввода, но на этот раз нам приходится хранить информацию состояния при обработке каждой группы ключей. В нашем примере ключами являются годы, мы сохраняем последний обнаруженный ключ и максимальную температуру, найденную для этого ключа. Инфраструктура MapReduce гарантирует упорядочение ключей; следовательно, если ключ отличается от предыдущего, это означает, что мы перешли к новой группе ключей. В отличие от Java API, где для каждой группы ключей предоставляется

¹ Так же можно использовать «pull-механику» обработки нового MapReduce API; см. раздел «Старый и новый Java API MapReduce» на с. 60.

итератор, при использовании Streaming границы группы приходится определять в программе.

Для каждой строки извлекается ключ и значение. Если группа только что завершилась (`last_key && last_key != key`), мы записываем ключ и максимальную температуру в группе, разделенные символом табуляции, перед сбросом максимальной температуры для нового ключа. Если группа еще не завершена, программа только обновляет максимальную температуру для текущего ключа.

Последняя строка программы обеспечивает запись строки для последней группы ключей во входных данных.

Теперь весь конвейер MapReduce можно имитировать при помощи конвейера Unix (эквивалентного конвейеру Unix на рис. 2.1):

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb | \
  sort | ch02/src/main/ruby/max_temperature_reduce.rb
1949    111
1950    22
```

Вывод не отличается от вывода программы на Java, так что следующим шагом должно стать выполнение с использованием Hadoop.

Команда `hadoop` не поддерживает параметра, напрямую включающего использование Streaming; вместо этого следует задать JAR-файл Streaming в параметре `jar`. Параметры Streaming определяют входной и выходной путь, а также сценарии отображения и свертки. Вот как это выглядит:

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-*streaming.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper ch02/src/main/ruby/max_temperature_map.rb \
  -reducer ch02/src/main/ruby/max_temperature_reduce.rb
```

При выполнении с большим набором данных в кластере следует использовать параметр `-combiner` для определения комбинирующей функции.

В выпусках после 1.x комбинирующей функцией может быть любая команда Streaming. В более ранних выпусках комбинирующая функция должна быть написана на Java, поэтому на практике часто использовалось обходное решение — ручное комбинирование в функции отображения без использования Java. В нашем случае функцию отображения можно заменить конвейером:

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-*streaming.jar \
  -input input/ncdc/all \
  -output output \
  продолжение ⇨
```

```
-mapper "ch02/src/main/ruby/max_temperature_map.rb | sort |  
    ch02/src/main/ruby/max_temperature_reduce.rb" \  
-reducer ch02/src/main/ruby/max_temperature_reduce.rb \  
-file ch02/src/main/ruby/max_temperature_map.rb \  
-file ch02/src/main/ruby/max_temperature_reduce.rb
```

Обратите внимание на параметр `-file`, который используется при запуске программ Streaming в кластере для доставки сценариев в кластер.

Python

Streaming поддерживает любые языки программирования с возможностью чтения из стандартного ввода и записи в стандартный вывод. Для читателей, лучше знакомых с языком Python, ниже приводится та же программа¹. Сценарий отображения приведен в листинге 2.10, а сценарий свертки — в листинге 2.11.

Листинг 2.10. Функция отображения для определения максимальной температуры на языке Python

```
#!/usr/bin/env python  
  
import re  
import sys  
  
for line in sys.stdin:  
    val = line.strip()  
    (year, temp, q) = (val[15:19], val[87:92], val[92:93])  
    if (temp != "+9999" and re.match("[01459]", q)):  
        print "%s\t%s" % (year, temp)
```

Листинг 2.11. Функция свертки для определения максимальной температуры на языке Python

```
#!/usr/bin/env python  
  
import sys  
  
(last_key, max_val) = (None, -sys.maxint)  
for line in sys.stdin:  
    (key, val) = line.strip().split("\t")
```

¹ В качестве альтернативы Streaming программисты Python могут рассмотреть технологию Dumbo (<http://www.last.fm/dumbo>), с которой работа с интерфейсом MapReduce становится более простой и удобной.

```
if last_key and last_key != key:  
    print "%s\t%s" % (last_key, max_val)  
    (last_key, max_val) = (key, int(val))  
else:  
    (last_key, max_val) = (key, max(max_val, int(val)))  
  
if last_key:  
    print "%s\t%s" % (last_key, max_val)
```

Тестирование программ и запуск задания осуществляются так же, как в примере на языке Ruby. Например, запуск теста выглядит так:

```
% cat input/ncdc/sample.txt | ch02/src/main/python/max_temperature_map.py | \  
sort | ch02/src/main/python/max_temperature_reduce.py  
1949      111  
1950      22
```

Hadoop Pipes

Hadoop Pipes — название интерфейса Hadoop MapReduce для языка C++. В отличие от технологии Streaming, использующей стандартный ввод и вывод для обмена данными с кодом отображения и свертки, Pipes использует сокеты в качестве канала, по которому трекер задачи взаимодействует с процессом, в котором выполняется функция отображения или свертки, написанная на C++. JNI при этом не используется.

Перепишем пример с температурой на языке C++, а потом посмотрим, как запустить его с использованием Pipes. В листинге 2.12 приведен исходный код функций отображения и свертки на языке C++.

Листинг 2.12. Поиск максимальной температуры на языке C++

```
#include <algorithm>  
#include <limits>  
#include <stdint.h>  
#include <string>  
  
#include "hadoop/Pipes.hh"  
#include "hadoop/TemplateFactory.hh"  
#include "hadoop/StringUtils.hh"  
  
class MaxTemperatureMapper : public HadoopPipes::Mapper {  
public:
```

продолжение ⇨

Листинг 2.12 (продолжение)

```

MaxTemperatureMapper(HadoopPipes::TaskContext& context) {
}
void map(HadoopPipes::MapContext& context) {
    std::string line = context.getInputValue();
    std::string year = line.substr(15, 4);
    std::string airTemperature = line.substr(87, 5);
    std::string q = line.substr(92, 1);
    if (airTemperature != "+9999" &&
        (q == "0" || q == "1" || q == "4" || q == "5" || q == "9")) {
        context.emit(year, airTemperature);
    }
}
};

class MapTemperatureReducer : public HadoopPipes::Reducer {
public:
    MapTemperatureReducer(HadoopPipes::TaskContext& context) {
    }
    void reduce(HadoopPipes::ReduceContext& context) {
        int maxValue = INT_MIN;
        while (context.nextValue()) {
            maxValue = std::max(maxValue,
                HadoopUtils::toInt(context.getInputValue()));
        }
        context.emit(context.getInputKey(), HadoopUtils::toString(maxValue));
    }
};

int main(int argc, char *argv[]) {
    return HadoopPipes::runTask(HadoopPipes::TemplateFactory<MaxTemperatureMapper,
                                MapTemperatureReducer>());
}

```

Приложение компонуется с библиотекой C++ Hadoop, которая представляет собой тонкую обертку для взаимодействия с дочерним процессом трекера задачи. Функции отображения и свертки определяются расширением классов `Mapper` и `Reducer`, определенных в пространстве имён `HadoopPipes`, и предоставлением реализаций методов `map()` и `reduce()`. Эти методы получают объект контекста (типа `MapContext` или `ReduceContext`), предоставляющий средства для чтения ввода и записи вывода, а также доступа к конфигурационной информации через класс `JobConf`. Обработка в приведенном примере очень похожа на ее эквивалент в Java.

В отличие от интерфейса Java, ключи и значения в интерфейсе C++ хранятся в байтовых буферах, представленных в виде строк STL (Standard Template

Library). Это упрощает интерфейс, хотя и несколько усложняет задачу разработчика приложения, которому приходится выполнять преобразования к типу уровня предметной области и обратно. Пример встречается в приложении MapTemperatureReducer, в котором нам приходится преобразовывать входное значение в целое число (с использованием вспомогательного метода из HadoopUtils), а затем — найденный максимум обратно в строку перед выводом. В некоторых случаях преобразование можно опустить (как, например, в MaxTemperatureMapper, где значение `airTemperature` не преобразуется в целое число, потому что оно никогда не обрабатывается в числовой форме в методе `map()`).

Метод `main()` является точкой входа приложения. Он вызывает метод `HadoopPipes::runTask`, который связывается с родительским процессом Java и обеспечивает обмен данными с `Mapper` и `Reducer`. Метод `runTask()` получает объект `Factory`, чтобы он мог создавать экземпляры `Mapper` или `Reducer`. Тем, какой именно объект будет создан, управляет родительский процесс Java по подключению через сокет. Существуют перегруженные шаблонные фабричные методы для назначения комбинирующей функции, функции разбиения, функции чтения и функции записи.

Компилирование и запуск

Для компилирования и запуска программы можно воспользоваться make-файлом из листинга 2.13.

Листинг 2.13. Make-файл для программы MapReduce на языке C++

```
CC = g++
CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include

max_temperature: max_temperature.cpp
    $(CC) $(CPPFLAGS) $< -Wall
    -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib -lhadooppipes \
    -lhadooputils -lpthread -g -O2 -o $@
```

Для работы make-файла необходимо задать пару переменных окружения. Кроме переменной `HADOOP_INSTALL`, также необходимо определить переменную `PLATFORM`, которая задает операционную систему, архитектуру и модель данных (32- или 64-разрядную). Я запускал пример в 32-разрядной системе Linux следующими командами:

```
% export PLATFORM=Linux-i386-32
% make
```

При успешном завершении в текущем каталоге появляется исполняемый файл *max_temperature*.

Чтобы запустить задание Pipes, необходимо запустить Hadoop в псевдораспределенном режиме (в котором все демоны выполняются на локальной машине). Интерфейс Pipes не работает в автономном (локальном) режиме, потому что он зависит от механизма распределенного кэша Hadoop, который работает только вместе с HDFS.

После того, как демоны Hadoop успешно заработают, скопируйте исполняемый файл в HDFS, чтобы он стал доступным для трекеров задач при запуске задач отображения и свертки:

```
% hadoop fs -put max_temperature bin/max_temperature.
```

Набор данных также необходимо скопировать из локальной файловой системы в HDFS:

```
% hadoop fs -put input/ncdc/sample.txt sample.txt.
```

Все готово к запуску задания. Для этого мы воспользуемся командой Hadoop *pipes*, передав ей URI исполняемого файла в HDFS в аргументе *-program*:

```
% hadoop pipes \  
  -D hadoop.pipes.java.recordreader=true \  
  -D hadoop.pipes.java.recordwriter=true \  
  -input sample.txt \  
  -output output \  
  -program bin/max_temperature
```

В аргументе *-D* передаются два свойства: *hadoop.pipes.java.recordreader* и *hadoop.pipes.java.recordwriter*. Обоим свойствам присвоено значение *true*; это означает, что мы не задаем функцию чтения или записи C++, а хотим использовать реализации Java по умолчанию (для текстового ввода и вывода). Pipes также позволяет задать функцию отображения, сокращения, разбиения или комбинирующую функцию Java. Собственно, ничто не мешает вам использовать сочетание классов Java и C++ в одном задании.

Программа выводит такой же результат, как и другие версии, приведенные ранее.

3 HDFS

Когда набор данных перерастает емкость одной физической машины, его приходится распределять по нескольким разным машинам. Файловые системы, управляющие хранением данных в сети, называются *распределенными файловыми системами*. Поскольку они работают в сетевой среде, проектировщику приходится учитывать все сложности сетевого программирования, поэтому распределенные файловые системы сложнее обычных дисковых файловых систем. Например, одна из самых серьезных проблем — сделать так, чтобы файловая система переживала сбои отдельных узлов без потери данных.

Hadoop поставляется с распределенной файловой системой, которая называется *HDFS* (Hadoop Distributed Filesystem). Иногда — в старой документации или конфигурациях или в неформальном общении — также встречается сокращение «DFS»; оно означает то же самое. HDFS — основная файловая система Hadoop, которой посвящена эта глава, но в Hadoop также реализована абстракция обобщенной файловой системы, и мы попутно рассмотрим интеграцию Hadoop с другими системами хранения данных (например, локальной файловой системой и Amazon S3).

Строение HDFS

Файловая система HDFS спроектирована для хранения очень больших файлов с потоковой схемой доступа к данным в кластерах обычных машин¹. Рассмотрим это утверждение более подробно.

Очень большие файлы

Под «очень большими» в этом контексте подразумеваются файлы, размер которых составляет сотни мегабайт, гигабайт и терабайт. Сейчас существуют кластеры Hadoop, в которых хранятся петабайты данных².

Потоковый доступ к данным

В основу HDFS заложена концепция однократной записи/многократного чтения как самая эффективная схема обработки данных. Набор данных обычно генерируется или копируется из источника, после чего с ним выполняются различные аналитические операции. В каждой операции задействуется большая часть набора данных (или весь набор), поэтому время чтения всего набора данных важнее задержки чтения первой записи.

Обычное оборудование

Hadoop не требует дорогостоящего оборудования высокой надежности. Система спроектирована для работы на стандартном оборудовании (общедоступное оборудование, которое может быть приобретено у многих фирм)³ с достаточно высокой вероятностью отказа отдельных узлов в кластере (по крайней мере, для больших кластеров). Технология HDFS спроектирована таким образом, чтобы в случае отказа система продолжала работу без сколько-нибудь заметного прерывания.

Также следует выделить области применения, для которых в настоящее время HDFS подходит не лучшим образом (при том, что в будущем ситуация может измениться):

Быстрый доступ к данным

Приложения, требующие доступа к данным с минимальной задержкой (в диапазоне десятков миллисекунд), плохо сочетаются с HDFS. Напомним,

¹ Архитектура HDFS описана в докладе «The Hadoop Distributed File System» Константина Швачко (Konstantin Shvachko), Хайрона Куана (Hairong Kuang), Санджая Радья (Sanjay Radia) и Роберта Ченслера (Robert Chansler) (Proceedings of MSST2010, May 2010, <http://storageconference.org/2010/Papers/MSST/Shvachko.pdf>).

² «Scaling Hadoop to 4000 nodes at Yahoo!», http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html.

³ Типичная спецификация компьютера приведена в главе 9.

что система HDFS оптимизирована для обеспечения высокой пропускной способности передачи данных, за которую приходится расплачиваться замедлением доступа. HBase (глава 13) в настоящее время лучше подходит для организации доступа к данным с минимальной задержкой.

Многочисленные мелкие файлы

Так как узел имен хранит метаданные файловой системы в памяти, предел количества файлов в файловой системе определяется объемом памяти узла имен. Как показывает опыт, каждый файл, каталог и блок занимают около 150 байт. Таким образом, например, если у вас имеется миллион файлов, каждый из которых занимает один блок, для хранения информации потребуется не менее 300 Мбайт памяти. Хранение миллионов файлов еще приемлемо, но миллиарды файлов уже выходят за пределы возможностей современного оборудования¹.

Множественные источники записи, произвольные модификации файлов

Запись в файлы HDFS может выполняться только одним источником. Запись всегда осуществляется в конец файла. Поддержка множественных источников записи или модификации с произвольным смещением в файле отсутствует. (Может быть, эти возможности будут поддерживаться в будущем, но, скорее всего, они будут относительно неэффективными).

Основные концепции HDFS

Блоки

С дисковым устройством связывается размер блока — минимальный объем данных, используемых в операции чтения или записи. Однодисковые файловые системы используют это обстоятельство, возвращая данные в блоках, размер которых кратен размеру дискового блока. Размер блока файловой системы обычно составляет несколько килобайт, тогда как размер дискового блока обычно равен 512 байтам. Как правило, все эти нюансы полностью прозрачны для пользователей файловых систем, которые просто читают или записывают в файлы данные произвольной длины. Однако программы сопровождения файловых систем (такие, как *df* и *fsck*) работают на уровне блоков файловой системы.

¹ За подробным анализом пределов масштабируемости HDFS обращайтесь к публикации Константина Швачко «Scalability of the Hadoop Distributed File System» (http://developer.yahoo.net/blogs/hadoop/2010/05/scalability_of_the_hadoop_dist.html) и сопроводительной статье «HDFS Scalability: The limits to growth» (апрель 2010 г., с. 6–16), <http://www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf>) того же автора.

В HDFS тоже существует концепция блока, но он имеет существенно больший размер — по умолчанию 64 Мбайт. Как и в однодисковых файловых системах, файлы HDFS разбиваются на блочные фрагменты, хранимые независимо друг от друга. В отличие от однодисковых файловых систем, файл HDFS, меньший одного блока, не занимает все пространство, выделенное под блок. Если в тексте не указано обратное, термином «блок» в книге обозначаются блоки HDFS.

ПОЧЕМУ БЛОКИ HDFS ТАКИЕ БОЛЬШИЕ?

Блоки HDFS существенно больше дисковых блоков. Это сделано для того, чтобы свести к минимуму количество операций позиционирования. При достаточно большом размере блока время передачи данных с диска может быть намного больше времени позиционирования к началу блока. Таким образом, время передачи большого файла, состоящего из многих блоков, определяется скоростью передачи данных.

Несложные вычисления показывают, что если время позиционирования составляет около 10 миллисекунд, а скорость передачи составляет 100 Мбайт/с, то, чтобы время позиционирования составляло 1% от времени передачи, размер блока должен составлять примерно 100 Мбайт. По умолчанию используется значение 64 Мбайт, хотя во многих установках HDFS размер блока увеличен до 128 Мбайт. Можно ожидать, что с ростом скорости передачи данных в новых поколениях дисковых накопителей размер блока будет расти.

Впрочем, с увеличением размера блока не стоит заходить слишком далеко. Задачи отображения в MapReduce обычно работают с одним блоком, так что при малом количестве задач (меньшем числа узлов в кластере) ваши задания будут выполняться медленнее, чем могли бы.

Абстракция блоков в распределенной файловой системе имеет несколько преимуществ. Первое преимущество наиболее очевидно: файл может быть больше любого отдельного диска в сети. Блоки файла совершенно необязательно хранить на одном диске; они могут использовать любые диски в кластере. Более того, при хранении файла в кластере HDFS его блоки могут быть распределены по всем дискам кластера (хотя эта ситуация несколько нетипична).

Второе: использование в качестве абстрактной единицы блока вместо файла упрощает подсистему хранения. Простота — желанное свойство любых систем, но она особенно важна в распределенных системах с их разнообразными возможными режимами отказов. Использование блоков в подсистеме хранения данных упрощает хранение (так как блоки имеют фиксированный размер, система может легко вычислить количество блоков, которые могут храниться на заданном диске)

и устраняет проблемы с метаданными (блок — всего лишь фрагмент данных, предназначенный для сохранения, с ним не нужно сохранять метаданные файла — скажем, информацию о разрешениях доступа, которые могут отдельно обрабатываться другой системой).

Кроме того, блоки хорошо вписываются в механизм репликации — они улучшают отказоустойчивость и доступность системы. Для защиты от поврежденных блоков и сбоев дисков/машин, каждый блок реплицируется на небольшом количестве физически разделенных машин (как правило, трех). Если блок становится недоступным, его копия читается из другого места, причем это происходит прозрачно для клиента. Блок, ставший недоступным из-за повреждения данных или сбоя оборудования, копируется из альтернативных хранилищ на другие работоспособные машины, чтобы довести фактор репликации до нормального уровня. (О защите от повреждения данных более подробно рассказано в разделе «Целостность данных» на с. 125). Кроме того, некоторые приложения могут установить высокий фактор репликации для блоков часто запрашиваемого файла, чтобы улучшить распределение нагрузки по чтению в пределах кластера.

Команда HDFS *fsck*, как и ее «родственник» из традиционных файловых систем, работает с информацией о блоках. Например, команда

```
% hadoop fsck / -files -blocks
```

выводит список блоков, из которых состоит каждый файл в файловой системе (также см. «Проверка файловой системы (*fsck*)», с. 445).

Узлы имен и узлы данных

Кластер HDFS состоит из двух типов узлов, работающих по схеме «управляющий-подчиненный»: *узел имен* (управляющий) и несколько *узлов данных* (подчиненные). Узел имен управляет пространством имен файловой системы. Он поддерживает дерево файловой системы и метаданные всех файлов и каталогов в дереве. Информация хранится на локальном диске в виде двух файлов: образа пространства имен и журнала изменений. Узел имен также знает, на каких узлах данных хранятся все блоки заданного файла; однако информация о местонахождении блоков не хранится постоянно, а строится заново по сведениям узлов данных при запуске системы.

Чтобы обратиться к файловой системе по поручению пользователя, клиент связывается с узлом имен и узлами данных. Клиент предоставляет интерфейс файловой системы, сходный с интерфейсом POSIX (Portable Operating System Interface), так что пользовательскому коду не нужно ничего знать об узлах имен и узлах данных.

Узлы данных – основная «рабочая сила» файловой системы. Они читают и записывают блоки (по требованию клиентов или узла имен), а также периодически передают узлу имен список сохраняемых ими блоков.

Без узла имен файловая система становится неработоспособной. Более того, при уничтожении машины, на которой работает узел имен, все файлы в файловой системе будут потеряны, потому что восстановить их по блокам узлов данных будет невозможно. По этой причине важно сделать узел имен устойчивым к возможным сбоям; Hadoop предоставляет два механизма решения этой задачи.

Первый способ – архивация файлов, определяющих устойчивое состояние метаданных файловой системы. Hadoop можно настроить таким образом, чтобы узел имен записывал свое устойчивое состояние в несколько файловых систем. Операции записи выполняются синхронно и атомарно. Обычно в конфигурации настраивается запись на локальный диск и в удаленный том NFS.

Также можно запустить вторичный узел имен, который, несмотря на название, не выполняет функции узла имен. Его основная функция – периодическое слияние образа пространства имен с журналом изменений, чтобы предотвратить чрезмерное разрастание последнего. Вторичный узел имен обычно работает на отдельной физической машине, потому что для выполнения слияния необходимы значительная вычислительная мощность и столько же памяти, сколько на узле имен. Вторичный узел имен хранит копию объединенного образа пространства имен, который может использоваться в случае сбоя основного узла. Тем не менее состояние вторичного узла имен несколько отстает от состояния основного узла, так что в случае полного отказа последнего потеря данных практически неизбежна. Обычно в таких случаях файлы метаданных узла имен копируются из NFS на вторичный узел, который начинает работать как новый основной.

За подробностями обращайтесь к разделу «Образ файловой системы и журнал изменений» на с. 436.

HDFS Federation

Узел имен хранит ссылки на все файлы и блоки файловой системы в памяти; это означает, что в очень больших кластерах с множеством файлов память становится ограничивающим фактором масштабирования (см. раздел «Сколько памяти нужно узлу имен?» на с. 398).

Технология HDFS Federation, появившаяся в серии выпусков 2.x, обеспечивает возможность масштабирования кластера за счет добавления узлов имен, каждый из которых управляет частью пространства имен файловой системы. Например, один узел имен может управлять всеми файлами иерархии `/user`, а второй – файлами иерархии `/share`.

При использовании федерации каждый узел имен управляет *томом пространства имен*, состоящим из метаданных пространства имен и *пула блоков*, содержащего все блоки файлов в пространстве. Тома пространств имен независимы друг от друга; это означает, что узлы имен не обмениваются данными друг с другом, а сбой одного узла имен не влияет на доступность пространств имен, находящихся под управлением других узлов. Вместе с тем, пул блоков разбиению не подвергается, так что узлы данных регистрируются у каждого узла имен в кластере и хранят блоки из разных пулов.

Чтобы обратиться к федеративному кластеру HDFS, клиенты используют таблицы мониторинга для установления соответствия путей к файлам с узлами имен. Эта задача решается на уровне конфигурации с использованием ViewFileSystem и URI *viewfs://*.

Высокая доступность HDFS

Сочетание репликации метаданных узлов имен в нескольких файловых системах и использования вторичного узла имен для создания контрольных точек защищает от потери данных, но не обеспечивает высокой доступности системы. Узел имен по-прежнему остается единой точкой сбоя. Если в этой точке происходит сбой, все клиенты — включая задания MapReduce — теряют возможность чтения, записи или получения списка файлов, потому что узел имен является единственным хранилищем метаданных и информации о соответствии между файлами и блоками. В этом случае вся система Hadoop фактически утрачивает работоспособность, пока не будет запущен новый узел.

Чтобы восстановиться после сбоя узла имен в такой ситуации, администратор запускает новый основной узел имен с одной из реплик метаданных файловой системы и настраивает узлы данных и клиентов на использование нового узла. Новый узел имен не может обслуживать запросы до того, как 1) в память будет загружен образ пространства имен, 2) будет воспроизведен журнал изменений и 3) от узлов данных будет получено достаточно отчетов для выхода из безопасного режима. В больших кластерах с множеством файлов и блоков время запуска узла имен может составлять 30 и более минут.

Долгое восстановление также создает проблемы для повседневного сопровождения. Более того, непредвиденные сбои узлов имен настолько редки, что на практике запланированные простои играют более важную роль.

В серии выпусков Hadoop 2.x эта ситуация была исправлена добавлением поддержки высокой доступности (HA, High Availability) HDFS. В этой реализации используются два узла имен в конфигурации «активный/резервный». В случае сбоя активного узла имен резервный узел берет на себя обязанности по обслуживанию

клиентских запросов без сколько-нибудь заметного перерыва. Чтобы это стало возможно, потребовались некоторые архитектурные изменения:

- Узлы имен должны использовать общее хранилище данных с высокой доступностью для хранения журнала изменений. (В исходной реализации для этой цели используется фильтр NFS, но в будущих версиях появятся дополнительные возможности — например, система на базе BookKeeper, построенная на основе ZooKeeper.) При активизации резервный узел имен читает данные до конца общего журнала изменений, чтобы синхронизировать свое состояние с активным узлом имен, после чего продолжает читать новые записи по мере того, как они будут записываться активным узлом имен.
- Узлы данных должны отправлять блочные отчеты обоим узлам имен, потому что информация о соответствии блоков хранится в памяти узла имен, а не на диске.
- Клиенты должны быть настроены на преодоление сбоев узлов имен с использованием механизма, прозрачного для пользователя.

Если активный узел имен перестает работать, резервный узел может активизироваться очень быстро (за десятые доли секунды), потому что у него в памяти хранится последняя информация состояния: как последние записи журнала изменений, так и актуальная информация соответствия блоков. На практике время преодоления сбоя будет больше (около минуты), потому что решение об отказе активного узла имен должно приниматься системой по возможности консервативно.

В маловероятном случае, когда резервный узел имен оказывается неработоспособным в момент сбоя активного узла, администратор запускает резервный узел «с нуля». Ситуация, по крайней мере, не хуже, чем без высокой доступности; с организационной точки зрения она лучше, потому что процесс представляет собой стандартную процедуру, встроенную в Hadoop.

Преодоление сбоев и изоляция

Переходом от активного узла имен к резервному управляет новый компонент системы, называемый *контроллером преодоления сбоев*. Контроллеры преодоления сбоев заменяют; первая реализация использует ZooKeeper для проверки того, что активен только один узел имен. На каждом узле имен работает облегченный процесс контроллера преодоления сбоев, задача которого состоит в отслеживании сбоев на узле (с использованием простого механизма «проверки пульса») и инициировании преодоления сбоя в случае отказа.

Преодоление сбоев также может инициироваться вручную администратором — например, при регулярном техническом обслуживании. Это называется *корректным*

преодолением сбоев, так как контроллер преодоления сбоев обеспечивает организованное переключение ролей узлов имен.

Однако в случае аварийного преодоления сбоев невозможно быть полностью уверенным в том, что сбойный узел имен перестал работать. Например, медленная работа сети или сетевого раздела может привести к переходу в состояние преодоления сбоев, хотя ранее активный узел имен продолжает работать и полагает, что он все еще является активным. Реализация высокой доступности прикладывает большие усилия, чтобы предотвратить возможные повреждения данных со стороны ранее активного узла имен — этот метод называется *изоляцией*. Система использует различные механизмы изоляции, в числе которых уничтожение процесса узла имен, отзыв его прав доступа к каталогу общего хранения данных (обычно с использованием команды NFS конкретного разработчика) и отключение его сетевого порта командой дистанционного управления. В качестве крайней меры ранее активный узел имен может быть изолирован посредством принудительного отключения питания на управляющем компьютере со специального распределителя электропитания.

Преодоление сбоев на стороне клиента выполняется прозрачно клиентской библиотекой. Простейшая реализация использует для управления преодолением сбоев данные конфигурации на стороне клиента. HDFS URI использует логическое имя хоста, отображаемое на пару адресов узлов имен (в конфигурационном файле), а клиентская библиотека поочередно опробует каждый адрес, пока операция не завершится успехом.

Интерфейс командной строки

В этом разделе мы поближе познакомимся с HDFS, взаимодействуя с системой из командной строки. Существует много других интерфейсов к HDFS, но командная строка остается одним из простейших, а для многих разработчиков — самым знакомым механизмом взаимодействия.

В наших примерах используется установка HDFS на одной машине. Позднее мы покажем, как запустить HDFS в кластере для обеспечения масштабируемости и устойчивости к сбоям.

В пиведораспределенной конфигурации задаются два свойства, заслуживающие дополнительных пояснений. Первому — `fs.default.name` — задается значение `hdfs://localhost/`, определяющее файловую систему Hadoop по умолчанию. Файловые системы задаются идентификаторами URI; в данном случае URI `hdfs` приказывает Hadoop использовать HDFS по умолчанию. Демоны HDFS используют это свойство для определения хоста и порта узла имен HDFS. Мы запускаем его на

хосте *localhost*, по умолчанию используется порт HDFS 8020. Клиенты HDFS при помощи этого свойства определяют местонахождение узла имен для подключения к нему.

Второму свойству — `dfs.replication` — задается значение 1, чтобы система HDFS не использовала репликацию блоков файловой системы со стандартным коэффициентом 3. С одним узлом данных HDFS не сможет реплицировать блоки на трех узлах данных, поэтому система будет постоянно жаловаться на недостаточный уровень репликации. Свойство решает эту проблему.

Основные операции файловой системы

Файловая система готова к использованию. С ней можно выполнять все обычные операции файловых систем — чтение файлов, создание каталогов, перемещение файлов, удаление данных и чтение содержимого каталогов. Для получения подробной справки по каждой команде используйте команду `hadoop fs -help`.

Начнем с копирования файла из локальной файловой системы в HDFS:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt hdfs://localhost/user/tom/  
quangle.txt
```

Эта команда запускает команду оболочки файловой системы Hadoop `fs`, которая поддерживает несколько подкоманд — в нашем примере выполняется команда `-copyFromLocal`. Локальный файл *quangle.txt* копируется в файл */user/tom/quangle.txt* в экземпляре HDFS, работающем на хосте *localhost*. Вообще говоря, схему и хост в URI можно было не указывать и использовать значение по умолчанию *hdfs://localhost*, заданное в файле *core-site.xml*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

Также можно использовать относительный путь и скопировать файл в домашний каталог HDFS, которым в нашем примере является каталог */user/tom*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

Скопируем файл обратно в локальную файловую систему и убедимся в том, что он не изменился:

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt  
% md5 input/docs/quangle.txt quangle.copy.txt  
MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9  
MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

Контрольные суммы MD5 совпадают; это означает, что файл успешно выдержал копирование в HDFS и обратно.

В завершение опробуем возможность получения списка файлов HDFS. Сначала мы создадим каталог просто для того, чтобы проверить его присутствие в списке:

```
% hadoop fs -mkdir books  
% hadoop fs -ls .  
Found 2 items  
drwxr-xr-x - tom supergroup 0 2009-04-02 22:41 /user/tom/books  
-rw-r--r-- 1 tom supergroup 118 2009-04-02 22:29 /user/tom/quangle.txt
```

Полученная информация очень похожа на результат выполнения команды Unix *ls -l* с небольшими различиями. В первом столбце выводится режим доступа к файлу. Во втором столбце выводится коэффициент репликации файла (в традиционной файловой системе Unix его нет). Вспомните, что в конфигурации мы задали коэффициент репликации по умолчанию равным 1; это значение и указано в листинге. Для каталогов столбец остается пустым, потому что концепция репликации на них не распространяется — каталоги интерпретируются как метаданные и сохраняются узлом имен, но не узлами данных. В третьем и четвертом столбцах указаны владелец файла и группа. Пятый столбец содержит размер файла в байтах или нуль для каталогов. В шестом и седьмом столбцах находится дата и время последнего изменения. Наконец, восьмом столбце выводится абсолютное имя файла или каталога.

РАЗРЕШЕНИЯ В HDFS

Модель разрешений, используемая в HDFS при обращениях к файлам и каталогам, имеет много общего с POSIX. Разрешения делятся на три типа: разрешения чтения (r), разрешения записи (w) и разрешения исполнения (x). Разрешение чтения необходимо для чтения файлов или получения содержимого каталога. Разрешение записи необходимо для записи в файлы, а для каталогов — для создания или удаления содержащихся в них файлов или каталогов. Разрешение исполнения для файлов игнорируется, потому что в HDFS (в отличие от POSIX) выполнить файл невозможно, а для каталогов это разрешение необходимо для обращения к их потомкам.

С каждым файлом или каталогом связывается владелец, группа и режим доступа. Режим состоит из набора разрешений для пользователя, являющегося владельцем, разрешений пользователей, являющихся членами группы, и разрешений для остальных пользователей (то есть не являющихся владельцем и не входящих в группу).

По умолчанию разрешения клиента определяются именем пользователя и группой процесса, в котором он работает. Так как клиенты работают удаленно, это позволяет работать в качестве любого пользователя, просто создав учетную запись с нужным именем в удаленной системе. Таким образом, механизм разрешений должен использоваться только в сообществе пользователей для организации общего доступа к ресурсам файловой системы и предотвращения случайной потери данных, а не для защиты ресурсов в агрессивной среде. (Впрочем, в последних версиях Hadoop поддерживается аутентификация Kerberos, снимающая эти ограничения; см. врезку «Безопасность» на с. 419.) Несмотря на ограничения, проверку разрешений все же желательно включить (как это делается по умолчанию; см. описание свойства `dfs.permissions`), чтобы предотвратить случайное изменение или удаление важных частей файловой системы — как пользователями, так и автоматизированными инструментами или программами.

При включенной проверке разрешения владельца проверяются в том случае, если имя пользователя-клиента совпадает с именем владельца, а разрешения группы — в том случае, если клиент является членом группы; в противном случае проверяются разрешения для остальных пользователей. Также существует концепция суперпользователя, которому соответствует процесс узла имен. Для суперпользователя проверка разрешений не выполняется.

Файловые системы Hadoop

В Hadoop существует абстрактное представление файловой системы; HDFS — всего лишь одна из ее возможных реализаций. Абстрактный класс Java `org.apache.hadoop.fs.FileSystem` представляет файловую систему Hadoop. Несколько конкретных реализаций описаны в табл. 3.1.

Таблица 3.1. Файловые системы Hadoop

Файловая система	Схема URI	Реализация Java (из <code>org.apache.hadoop</code>)	Описание
Локальная	file	<code>fs.LocalFileSystem</code>	Файловая система для локально подключенных дисков с контрольными суммами на стороне клиента. Для локальных файловых систем без контрольных сумм используется <code>RawLocalFileSystem</code> . См. « <code>LocalFileSystem</code> », с. 127

Файловая система	Схема URI	Реализация Java (из org.apache.hadoop)	Описание
HDFS	hdfs	hdfs. DistributedFileSystem	Распределенная файловая система Hadoop. Система HDFS спроектирована для эффективной работы в сочетании с MapReduce
HFTP	hftp	hdfs.HftpFileSystem	Файловая система, предоставляющая доступ только для чтения к HDFS по протоколу HTTP (несмотря на название, HFTP не имеет никакого отношения к FTP). Часто используется с distcp (см. «Параллельное копирование с использованием distcp», с. 118) для копирования данных между кластерами HDFS с разными версиями
HSFTP	hsftp	hdfs.HsftpFileSystem	Файловая система, предоставляющая доступ только для чтения к HDFS по протоколу HTTP (также не имеет никакого отношения к FTP)
WebHDFS	webhdfs	hdfs.web.WebHdfsFile System	Файловая система, предоставляющая защищенный доступ для чтения/записи к HDFS по протоколу HTTP. Система WebHDFS проектировалась как замена для HFTP и HSFTP
HAR	har	fs.HarFileSystem	Файловая система, работающая поверх другой файловой системы, и предназначенная для архивации файлов. HAR обычно применяется для архивации файлов в HDFS для сокращения затрат памяти узла имен. См. «HAR», с. 121
KFS (Cloud- Store)	kfs	fs.kfs. KosmosFileSystem	CloudStore (ранее Kosmos) — распределенная файловая система, сходная с HDFS или Google GFS, написанная на C++. За дополнительной информацией обращайтесь по адресу http://code.google.com/p/kosmosfs/
FTP	ftp	fs.ftp.FTPFileSystem	Файловая система на базе сервера FTP
S3 (исход- ная)	s3n	fs.s3native. NativeS3FileSystem	Файловая система на базе Amazon S3. См. http://wiki.apache.org/hadoop/AmazonS3

Таблица 3.1 (продолжение)

Файловая система	Схема URI	Реализация Java (из org.apache.hadoop)	Описание
S3 (блочная)	s3	fs.s3.S3FileSystem	Файловая система на базе Amazon S3, хранящая файлы в блоках (по аналогии с HDFS) для преодоления ограничений размера файла в 5 Гбайт, действующих в S3
Distributed RAID	hdfs	hdfs.DistributedRaidFileSystem	«RAID»-версия системы HDFS, предназначенная для архивного хранения данных. Для каждого файла в HDFS создается (небольшой) файл контроля четности что позволяет сократить коэффициент репликации HDFS с 3 до 2, что сокращает затраты дискового пространства на 25–30% без повышения вероятности потери данных. Для работы Distributed RAID в кластере должен работать демон RaidNode
View	viewfs	viewfs.ViewFileSystem	Таблица монтирования на стороне клиента для других файловых систем Hadoop. Обычно используется для создания точек монтирования федеративных узлов имен (см. «HDFS Federation», с. 84)

Hadoop предоставляет несколько интерфейсов к своим файловым системам. Обычно для выбора правильного экземпляра файловой системы используется схема URI. Например, оболочка файловой системы, которую мы видели в предыдущем разделе, работает со всеми файловыми системами Hadoop. Чтобы получить список файлов в корневом каталоге локальной файловой системы, введите следующую команду:

```
% hadoop fs -ls file:///
```

Запуск программ MapReduce, работающих со всеми файловыми системами, возможен (а иногда даже очень удобен), но при обработке больших объемов данных следует выбирать распределенную файловую систему с оптимизацией локальности данных — в первую очередь HDFS (см. «MapReduce в перспективе», с. 63).

Интерфейсы

Реализация Hadoop написана на Java, и все взаимодействия с файловой системой Hadoop осуществляются через Java API. Например, оболочка файловой системы представляет собой Java-приложение, использующее класс Java `FileSystem` для выполнения операций. В этом разделе кратко описаны другие интерфейсы файловых систем. Чаще всего они используются с HDFS, так как для других файловых систем Hadoop обычно уже имеются готовые инструменты (FTP-клиенты для FTP, инструментарий S3 для S3 и т. д.), но многие из них работают с любой файловой системой Hadoop.

HTTP

Существует два способа обращения к HDFS по протоколу HTTP: прямой, когда демоны HDFS обслуживают запросы HTTP, и опосредованный, когда обращения к HDFS по поручению клиента осуществляются через стандартный API `DistributedFileSystem`. Эти два способа наглядно изображены на рис. 3.1.

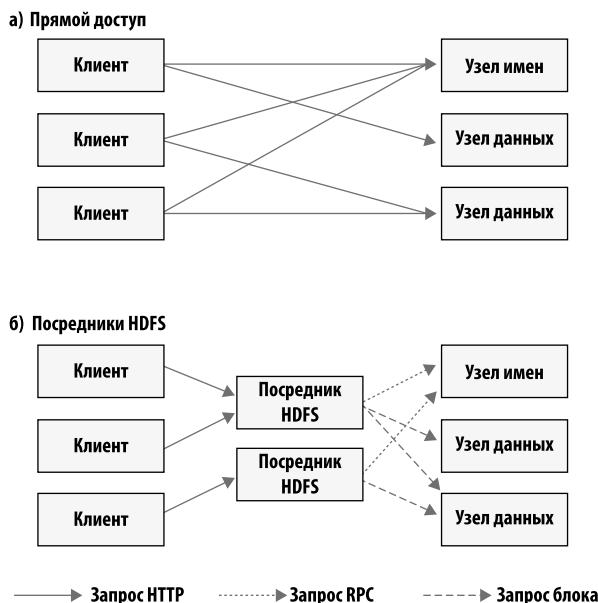


Рис. 3.1. Обращение к HDFS: напрямую через HTTP (а) и через посредников HDFS (б)

В первом случае списки содержимого каталогов предоставляются встроенным веб-сервером узла имен (порт 50070) в формате XML или JSON, а данные файлов передаются узлами данных их веб-серверами (порт 50075).

Исходный прямой интерфейс HTTP (HFTP и HSFTP) был доступен только для чтения, но новая реализация WebHDFS поддерживает все операции с файловыми системами, включая аутентификацию Kerberos. Чтобы включить поддержку WebHDFS, следует задать свойству `dfs.webhdfs.enabled` значение `true`; это позволит вам использовать URI `webhdfs`.

Второй механизм обращения к HDFS через HTTP использует один или несколько автономных серверов-посредников. (Посредники не поддерживают состояния, поэтому они могут запускаться за стандартными распределителями нагрузки.) Весь трафик к кластеру проходит через посредника, что позволяет установить более жесткую политику защитной фильтрации и ограничения нагрузки. Посредники обычно используются для передачи данных между кластерами Hadoop, расположенным в разных центрах обработки данных.

Исходный HDFS-посредник (в `src/contrib/hdfsproxy`) поддерживал доступ только для чтения, а обращаться к нему могли только клиенты, использующие `FileSystem`-реализацию HSFTP (URI `hsftp`). В версии 1.0.0. появился новый посредник HttpFS, который поддерживает чтение и запись и предоставляет тот же интерфейс HTTP, что и WebHDFS, так что клиенты могут обращаться к обоим интерфейсам через URI `webhdfs`.

HTTP REST API, предоставляемый WebHDFS, формально определен в спецификации. Ожидается, что со временем появятся клиенты, работающие с ним напрямую и написанные на других языках, отличных от Java.

C

Hadoop предоставляет библиотеку `libhdfs`, моделирующую интерфейс Java `FileSystem` (библиотека была написана на С для работы с HDFS, но, несмотря на имя, может использоваться для работы с любыми файловыми системами Hadoop). Для обращений к Java-клиенту файловой системы используется интерфейс JNI (Java Native Interface).

С API очень похож на Java API, но обычно его разработка отстает от разработки Java, поэтому новые функции в нем могут не поддерживаться. Сгенерированная документация для C API хранится в каталоге `libhdfs/docs/api` дистрибутива Hadoop. В комплект поставки Hadoop включаются готовые двоичные файлы `libhdfs` для 32-разрядной версии Linux; для других платформ вам придется построить их самостоятельно (инструкции доступны по адресу <http://wiki.apache.org/hadoop/LibHDFS>).

FUSE

FUSE (Filesystem in Userspace) обеспечивает возможность интеграции файловых систем, построенных в пользовательском пространстве, в качестве файловой

системы Unix. Модуль Hadoop Fuse-DFS contrib позволяет смонтировать любую файловую систему Hadoop (как правило, это HDFS) как стандартную файловую систему. Для взаимодействия с файловой системой можно использовать средства Unix (такие, как *ls* и *cat*), а также обращаться к ней из любого языка программирования с использованием библиотек POSIX.

Реализация Fuse-DFS написана на C, при этом в качестве интерфейса к HDFS используется *libhdfs*. Документация по компилированию и запуску Fuse-DFS находится в каталоге *src/contrib/fuse-dfs* дистрибутива Hadoop.

Интерфейс Java

В этом разделе мы подробнее рассмотрим класс Hadoop `FileSystem` — API для взаимодействия с одной из файловых систем Hadoop¹. Хотя основное внимание будет уделено на реализации HDFS `DistributedFileSystem`, в общем случае рекомендуется программировать для абстрактного класса `FileSystem`, чтобы код портировался для разных файловых систем. Например, это очень полезно при тестировании, потому что вы можете быстро выполнить тесты с использованием данных, хранящихся в локальной файловой системе.

Чтение данных Hadoop по URL-адресу

Один из простых способов чтения файла из файловой системы Hadoop — открытие потока, из которого будут читаться данные, с использованием объекта `java.net.URL`. Общая идиома выглядит так:

```
InputStream in = null;
try {
    in = new URL("hdfs://host/path").openStream();
    // Обработка in
} finally {
    IOUtils.closeStream(in);
}
```

Чтобы Java-программа распознавала схему URL-адресов Hadoop *hdfs*, необходимо дополнитель но потрудиться. Для этого следует вызвать для URL-адреса метод `setURLStreamHandlerFactory` с экземпляром `FsUrlStreamHandlerFactory`. Метод

¹ В версиях после 1.x появился новый интерфейс файловой системы `FileContext` с улучшенной поддержкой файловых систем (например, один объект `FileContext` может интерпретировать несколько схем файловых систем) и более стройным и последовательным интерфейсом.

может вызываться только один раз для каждой виртуальной машины, поэтому он обычно выполняется в статическом блоке. Из этого ограничения следует, что, если какая-то другая часть вашей программы (скажем, сторонний компонент, который вами не контролируется) вызывает `setURLStreamHandlerFactory`, вы не сможете использовать этот подход для чтения данных Hadoop. В следующем разделе рассматривается альтернативное решение.

В листинге 3.1 приведена программа, выводящая файлы из файловых систем Hadoop в стандартном выводе (по аналогии с командой Unix `cat`).

Листинг 3.1. Отображение файлов из файловой системы Hadoop в стандартном выводе с использованием URLStreamHandler

```
public class URLCat {  
    static {  
        URL.setURLStreamHandlerFactory(new FsURLStreamHandlerFactory());  
    }  
  
    public static void main(String[] args) throws Exception {  
        InputStream in = null;  
        try {  
            in = new URL(args[0]).openStream();  
            IOUtils.copyBytes(in, System.out, 4096, false);  
        } finally {  
            IOUtils.closeStream(in);  
        }  
    }  
}
```

Мы используем удобный класс `IOUtils`, поставляемый с Hadoop, для закрытия потока в секции `finally`, а также для копирования байтов между входным и выходным потоками (`System.out` в нашем случае). В последних двух аргументах метода `copyBytes` передается размер буфера, используемого для копирования, и флаг закрытия потока при завершении копирования. Мы закрываем входной поток самостоятельно, поэтому `System.out` закрывать не нужно.

Пример выполнения¹:

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt  
On the top of the Crumpetty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.
```

¹ Отрывок из стихотворения Эдварда Лира (Edward Lear) «The Quangle Wangle's Hat».

Чтение данных с использованием Filesystem API

Как объяснялось в предыдущем разделе, иногда вариант с вызовом `setURLStreamHandlerFactory` в приложении невозможен. В таком случае приходится открывать входной поток для файла с использованием `FileSystem`.

Файл в файловой системе Hadoop представлен объектом Hadoop `Path` (а не объектом `java.io.File`, поскольку семантика последнего слишком тесно связана с локальной файловой системой). Объект `Path` можно представить себе как URI в файловой системе Hadoop (например, `hdfs://localhost/user/tom/quangle.txt`).

`FileSystem` – обобщенный API файловой системы, поэтому первым шагом должно стать получение экземпляра файловой системы, которую мы хотим использовать – HDFS в нашем случае. Существует несколько статических фабричных методов для получения экземпляра `FileSystem`:

```
public static FileSystem get(Configuration conf) throws IOException  
public static FileSystem get(URI uri, Configuration conf) throws IOException  
public static FileSystem get(URI uri, Configuration conf, String user)  
    throws IOException
```

В объекте `Configuration` инкапсулируется конфигурация клиента или сервера, определяемая в конфигурационных файлах (например, `conf/core-site.xml`). Первый метод возвращает файловую систему по умолчанию (указанную в файле `conf/core-site.xml`, или локальную файловую систему по умолчанию, если она не задана). Второй метод определяет файловую систему по схеме и полномочиям (`authority`) заданного URI; если схема в URI не указана, используется файловая система по умолчанию. Третий метод получает объект файловой системы от имени заданного пользователя, что важно в контексте безопасности (см. «Безопасность», с. 419).

В некоторых ситуациях требуется получить экземпляр локальной файловой системы. В этом случае можно воспользоваться вспомогательным методом `getLocal()`:

```
public static LocalFileSystem getLocal(Configuration conf) throws IOException.
```

Получив экземпляр `FileSystem`, мы можем получить входной поток для файла вызовом метода `open()`:

```
public FSDataInputStream open(Path f) throws IOException  
public abstract FSDataInputStream open(Path f, int bufferSize)  
    throws IOException
```

Первый метод использует размер буфера по умолчанию, равный 4 Кбайт.

С учетом всего сказанного мы можем переписать программу из листинга 3.1 так, как показано в листинге 3.2.

Листинг 3.2. Отображение файлов из файловой системы Hadoop в стандартном выводе с использованием FileSystem

```
public class FileSystemCat {

    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        InputStream in = null;
        try {
            in = fs.open(new Path(uri));
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

Результат выполнения программы:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream

Метод `open()` объекта `FileSystem` вместо стандартного класса `java.io` возвращает объект класса `FSDataInputStream`. Этот класс представляет собой специализацию `java.io.DataInputStream` с поддержкой произвольного доступа, что позволяет читать данные из любой части потока:

```
package org.apache.hadoop.fs;
public class FSDataInputStream extends DataInputStream
    implements Seekable, PositionedReadable {
    // ...
}
```

Интерфейс `Seekable` обеспечивает позиционирование и содержит метод для получения текущего смещения от начала файла (`getPos()`):

```
public interface Seekable {
    void seek(long pos) throws IOException;
```

```
    long getPos() throws IOException;  
}
```

При вызове `seek()` с позицией, превышающей длину файла, происходит исключение `IOException`. В отличие от метода `skip()` класса `java.io.InputStream`, переводящего поток в точку за текущей позицией, метод `seek()` позволяет перейти к произвольной абсолютной позиции в файле.

В листинге 3.3 приведено расширение листинга 3.2, в котором файл направляется в стандартный вывод дважды, с позиционированием в начало файла после первого раза.

Листинг 3.3. Возврат к началу потока с использованием seek

```
public class FileSystemDoubleCat {  
  
    public static void main(String[] args) throws Exception {  
        String uri = args[0];  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(URI.create(uri), conf);  
        FSDataInputStream in = null;  
        try {  
            in = fs.open(new Path(uri));  
            IOUtils.copyBytes(in, System.out, 4096, false);  
            in.seek(0); // Возврат к началу файла  
            IOUtils.copyBytes(in, System.out, 4096, false);  
        } finally {  
            IOUtils.closeStream(in);  
        }  
    }  
}
```

Результат выполнения программы для небольшого файла:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt  
On the top of the Crumpty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.  
On the top of the Crumpty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.
```

Класс `FSDataInputStream` также реализует интерфейс `PositionedReadable` для чтения частей файла с заданным смещением:

```
public interface PositionedReadable {  
    public int read(long position, byte[] buffer, int offset, int length)  
        throws IOException;  
  
    public void readFully(long position, byte[] buffer, int offset, int length)  
        throws IOException;  
  
    public void readFully(long position, byte[] buffer) throws IOException;  
}
```

Метод `read()` читает до `length` байтов из заданной позиции файла в позицию со смещением `offset` в буфере `buffer`. Возвращаемое значение содержит количество фактически прочитанных байтов; при вызове следует проверить полученное значение, потому что оно может быть меньше `length`. Методы `readFully()` читают в буфер `length` байтов (или `buffer.length` для версии, получающей байтовый массив `buffer`); если при этом будет достигнут конец файла, выдается исключение `EOFException`.

Все эти методы сохраняют текущее смещение в файле и безопасны по отношению к программным потокам. Они предоставляют удобный механизм обращения к другим частям файла (например, метаданным) во время чтения основного содержимого файла.

Наконец, следует помнить, что вызов `seek()` является относительно высокозатратной операцией, поэтому использовать его следует осмотрительно. Модель доступа к данным в приложении должна базироваться на потоковой обработке данных (например, с использованием MapReduce), а не на многочисленных операциях позиционирования.

Запись данных

Класс `FileSystem` содержит несколько методов для создания файлов. Простейший метод получает объект `Path` создаваемого файла и возвращает выходной поток, в который записываются данные:

```
public FSDataOutputStream create(Path f) throws IOException.
```

Перегруженные версии этого метода позволяют задать признак принудительной замены существующих файлов, коэффициент репликации файла, размер буфера записи, размер блока файла и разрешения доступа.



Методы `create()` создают все родительские каталоги файла, не существующие на момент вызова. Такое поведение удобно, но иногда оно оказывается неожиданным. Если операция записи должна завершаться неудачей, если родительский каталог не существует, заранее проверьте его существование методом `exists()`.

Также существует перегруженный метод, которому передается интерфейс обратного вызова `Progressable`, чтобы ваше приложение оповещалось о ходе записи данных в узлы данных:

```
package org.apache.hadoop.util;
public interface Progressable {
    public void progress();
}
```

Вместо создания нового файла можно присоединить данные к существующему файлу методом `append()` (также существующему в нескольких перегруженных версиях):

```
public FSDataOutputStream append(Path f) throws IOException.
```

Операция `append` позволяет одному источнику записи изменить уже записанный файл; для этого файл открывается, а данные записываются с максимальным смещением. С таким API приложения, создающие файлы неограниченной длины (например, файлы журналов), могут дописывать данные в существующий файл после его закрытия. Операция присоединения реализована не всеми файловыми системами Hadoop. Например, HDFS поддерживает присоединение¹, а файловые системы S3 его не поддерживают.

В листинге 3.4 показано, как выполняется копирование файла в файловую систему Hadoop. Чтобы продемонстрировать ход выполнения операции, мы выводим точку каждый раз, когда Hadoop вызывает метод `progress()` — это происходит после записи каждого 64-килобайтного пакета данных в канал узла данных. (Учтите, что это конкретное поведение не определено в API, так что оно может измениться в будущих версиях Hadoop).

Листинг 3.4. Копирование локального файла в файловую систему Hadoop

```
public class FileCopyWithProgress {
    public static void main(String[] args) throws Exception {
        String localSrc = args[0];
        продолжение ↗
    }
}
```

¹ В Hadoop 1.x у операции присоединения возникают проблемы с надежностью, так что обычно рекомендуется использовать ее только в последующих версиях с новой реализацией.

Листинг 3.4 (продолжение)

```

String dst = args[1];

InputStream in = new BufferedInputStream(new FileInputStream(localSrc));

Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(dst), conf);
OutputStream out = fs.create(new Path(dst), new Progressable() {
    public void progress() {
        System.out.print(".");
    }
});

IOUtils.copyBytes(in, out, 4096, true);
}
}

```

Типичный пример использования:

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt hdfs://localhost/user/tom/
1400-8.txt
.....
```

Все остальные файловые системы Hadoop в настоящее время не вызывают `progress()` во время записи. Как вы узнаете далее, получение информации о ходе операции играет важную роль в приложениях MapReduce.

FSDataOutputStream

Метод `create()` объекта `FileSystem` возвращает объект `FSDataOutputStream`, который, как и `FSDataInputStream`, содержит метод для получения текущей позиции в файле:

```

package org.apache.hadoop.fs;
public class FSDataOutputStream extends DataOutputStream implements Syncable {
    public long getPos() throws IOException {
        // ...
    }

    // ...
}

```

Однако в отличие от `FSDataInputStream`, `FSDataOutputStream` не поддерживает позиционирования. Это объясняется тем, что HDFS поддерживает только последовательную запись в открытый файл или присоединение данных к уже записанному

файлу. Другими словами, запись поддерживается только в конце файла, так что возможность позиционирования при записи все равно бесполезна.

Каталоги

`FileSystem` содержит метод для создания каталогов:

```
public boolean mkdirs(Path f) throws IOException.
```

Метод создает все необходимые родительские каталоги, если они не существуют (по аналогии с методом `mkdir()` класса `java.io.File`). Если каталог (вместе со всеми родительскими каталогами) был создан успешно, метод возвращает `true`.

Часто явное создание каталогов оказывается излишним, потому что при записи в файл вызовом `create()` все родительские каталоги создаются автоматически.

Получение информации от файловой системы

Метаданные файла: `FileStatus`

Любая файловая система должна поддерживать возможность обхода структуры каталогов и получения информации о хранящихся в ней файлах и каталогах. Класс `FileStatus` инкапсулирует метаданные файловой системы о файлах и каталогах, включая длину файла, размер блока, коэффициент репликации, время модификации, данные владельца и разрешения доступа. Метод `getFileStatus()` класса `FileSystem` предоставляет способ получения объекта `FileStatus` для одного файла или каталога. Пример использования приведен в листинге 3.5.

Листинг 3.5. Использование информации `FileStatus`

```
public class ShowFileStatusTest {  
  
    private MiniDFSCluster cluster; // Использование внутрипроцессного  
                                   // кластера HDFS для тестирования  
    private FileSystem fs;  
  
    @Before  
    public void setUp() throws IOException {  
        Configuration conf = new Configuration();  
        if (System.getProperty("test.build.data") == null) {  
            System.setProperty("test.build.data", "/tmp");  
        }  
    }
```

продолжение ⇨

Листинг 3.5 (продолжение)

```
cluster = new MiniDFSCluster(conf, 1, true, null);
fs = cluster.getFileSystem();
OutputStream out = fs.create(new Path("/dir/file"));
out.write("content".getBytes("UTF-8"));
out.close();
}

{@After
public void tearDown() throws IOException {
    if (fs != null) { fs.close(); }
    if (cluster != null) { cluster.shutdown(); }
}

@Test(expected = FileNotFoundException.class)
public void throwsFileNotFoundExceptionForNonExistentFile() throws IOException {
    fs.getFileStatus(new Path("no-such-file"));
}

@Test
public void fileStatusForFile() throws IOException {
    Path file = new Path("/dir/file");
    FileStatus stat = fs.getFileStatus(file);
    assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
    assertThat(stat.isDir(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(64 * 1024 * 1024L));
    assertThat(stat.getOwner(), is("tom"));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rw-r--r--"));
}

@Test
public void fileStatusForDirectory() throws IOException {
    Path dir = new Path("/dir");
    FileStatus stat = fs.getFileStatus(dir);
    assertThat(stat.getPath().toUri().getPath(), is("/dir"));
    assertThat(stat.isDir(), is(true));
    assertThat(stat.getLen(), is(0L));
    assertThat(stat.getModificationTime(),
```

```
        isLessThanOrEqualTo(System.currentTimeMillis())));
assertThat(stat.getReplication(), is((short) 0));
assertThat(stat.getBlockSize(), is(0L));
assertThat(stat.getOwner(), is("tom"));
assertThat(stat.getGroup(), is("supergroup"));
assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
}
}
```

Если файл или каталог не существует, выдается исключение `FileNotFoundException`. Если вам нужно только проверить существование файла или каталога, удобнее воспользоваться методом `exists()` класса `FileSystem`:

```
public boolean exists(Path f) throws IOException
```

Получение списка файлов

Получение информации об отдельном файле или каталоге — полезная возможность, но часто требуется получить информацию о содержимом каталога. Для этого используются методы `listStatus()` класса `FileSystem`:

```
public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
public FileStatus[] listStatus(Path[] files) throws IOException
public FileStatus[] listStatus(Path[] files, PathFilter filter) throws
    IOException
```

Если аргумент определяет файл, простейшая версия метода возвращает массив объектов `FileStatus` длины 1. Если аргумент определяет каталог, возвращается нуль и более объектов `FileStatus`, представляющих файлы и каталоги, содержащиеся в каталоге.

Перегруженным версиям может передаваться объект `PathFilter`, определяющий фильтр для отбора файлов и каталогов. Пример представлен в разделе «`PathFilter`», с. 108. Наконец, при передаче массива путей результат представляет собой упрощенную запись для вызова эквивалентного «однопутевого» метода `listStatus` последовательно для каждого пути с накоплением массивов объектов `FileStatus` в одном массиве. Например, эта возможность может пригодиться для построения списков файлов из разных частей дерева файловой системы. В листинге 3.6 приведена несложная демонстрация этой идеи. Обратите внимание на использование метода `stat2Paths()` класса `FileUtil` для преобразования массива объектов `FileStatus` в массив объектов `Path`.

Листинг 3.6. Вывод информации о файлах для набора путей в файловой системе Hadoop

```
public class ListStatus {

    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);

        Path[] paths = new Path[args.length];
        for (int i = 0; i < paths.length; i++) {
            paths[i] = new Path(args[i]);
        }

        FileStatus[] status = fs.listStatus(paths);
        Path[] listedPaths = FileUtil.stat2Paths(status);
        for (Path p : listedPaths) {
            System.out.println(p);
        }
    }
}
```

Пример использования программы для получения объединенного списка содержимого каталогов:

```
% hadoop ListStatus hdfs://localhost/ hdfs://localhost/user/tom
hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

Шаблоны имен файлов

Обработка набора файлов за одну операцию — достаточно распространенная задача. Например, задание MapReduce может анализировать накопившиеся за месяц журнальные файлы, хранящиеся в нескольких каталогах. Вместо того, чтобы перебирать каждый файл и каждый каталог, удобно использовать метасимволы для описания нескольких файлов в одном выражении. Hadoop предоставляет два метода `FileSystem` для обработки шаблонов имен файлов:

```
public FileStatus[] globStatus(Path pathPattern) throws IOException
public FileStatus[] globStatus(Path pathPattern, PathFilter filter) throws
    IOException
```

Метод `globStatus()` возвращает массив объектов `FileStatus`, пути которых соответствуют заданному шаблону; элементы массива сортируются по путям.

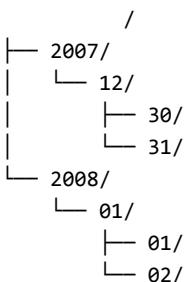
Дополнительный объект `PathFilter` обеспечивает дополнительную фильтрацию результатов.

Hadoop поддерживает те же метасимволы шаблонов имен файлов, что и *bash* (табл. 3.2).

Таблица 3.2. Метасимволы шаблонов имен файлов и их значения

Метасимвол	Название	Совпадение
*	звездочка	Нуль и более символов
?	вопросительный знак	Один символ
[ab]	символьный класс	Один символ из множества {a, b}
[^ab]	инвертированный символьный класс	Один символ, не входящий в множество {a, b}
[a-b]	диапазон символов	Один символ из (замкнутого) множества {a, b}, где a в лексикографическом отношении меньше либо равно b
[^a-b]	инвертированный диапазон символов	Один символ, не входящий в (замкнутое) множество {a, b}, где a в лексикографическом отношении меньше либо равно b
{a, b}	альтернатива	Выражение a или выражение b
\c	экранированный символ	Символ c, который без префикса интерпретируется как метасимвол

Предположим, журналы хранятся в структуре каталогов, упорядоченной иерархически по дате. Так, файлы за последний день 2007 года хранятся в каталоге `/2007/12/31`. Полная иерархия выглядит так:



Несколько шаблонов имен файлов и их смысл:

Шаблон	Расширение
/*	/2007 /2008
/*/*	/2007/12 /2008/01
/*/12/*	/2007/12/30 /2007/12/31
/200?	/2007 /2008
/200[78]	/2007 /2008
/200[7-8]	/2007 /2008
/200[^01234569]	/2007 /2008
/*/*/{31,01}	/2007/12/31 /2008/01/01
/*/*/{3{0,1}}	/2007/12/30 /2007/12/31
/*/{12/31,01/01}	/2007/12/31 /2008/01/01

PathFilter

Возможности шаблонов имен файлов порой оказываются недостаточными для описания всех файлов, с которыми вы хотите работать. Например, при использовании шаблонов в общем случае невозможно исключить из обработки конкретный файл. Методы `listStatus()` и `globStatus()` класса `FileSystem` получают дополнительный объект `PathFilter`, который предоставляет возможность программного управления процессом поиска совпадений:

```
package org.apache.hadoop.fs;
public interface PathFilter {
    boolean accept(Path path);
}
```

`PathFilter` представляет собой эквивалент класса `java.io.FileFilter` для объектов `Path` (вместо объектов `File`).

В листинге 3.7 приведена реализация `PathFilter` для исключения путей, совпадающих с регулярным выражением.

Листинг 3.7. Реализация `PathFilter` для исключения путей, совпадающих с регулярным выражением

```
public class RegexExcludePathFilter implements PathFilter {

    private final String regex;
    public RegexExcludePathFilter(String regex) {
```

```
    this.regex = regex;
}
public boolean accept(Path path) {
    return !path.toString().matches(regex);
}
}
```

Фильтр пропускает только те файлы, которые не совпадают с регулярным выражением. После того, как шаблон отбирает исходный набор включаемых файлов, фильтр используется для уточнения результатов. Например, фильтр

```
fs.globStatus(new Path("/2007/*/*"), new RegexExcludeFilter("^.*/2007/12/31$"))
```

расширяется в */2007/12/30*.

Фильтры могут работать только с именами файлов, представленными `Path`. Они не могут использовать для фильтрации свойства файла — например, время создания. Тем не менее они могут выполнять отбор файлов так, как этого не могут делать ни шаблоны имен файлов, ни регулярные выражения. Например, если файлы хранятся в структуре каталогов, упорядоченных по дате (как в предыдущем разделе), можно написать реализацию `PathFilter`, которая отбирает файлы из заданного диапазона дат.

Удаление данных

Для удаления файлов или каталогов используется метод `delete()` класса `FileSystem`:

```
public boolean delete(Path f, boolean recursive) throws IOException.
```

Если `f` — файл или пустой каталог, значение `recursive` игнорируется. Непустой каталог удаляется вместе с содержимым только в том случае, если параметр `recursive` равен `true` (в противном случае инициируется исключение `IOException`).

Поток данных

Чтение файла

Чтобы получить представление о том, как организована передача данных между клиентом, взаимодействующим с HDFS, узлом имен и узлами данных, взгляните на рис. 3.2. На нем изображена основная последовательность событий при чтении файла.

Клиент открывает файл, из которого он хочет прочитать данные, вызывая метод `open()` объекта `FileSystem`, который для HDFS представляет собой экземпляр `DistributedFileSystem` (шаг 1 на рис. 3.2). `DistributedFileSystem` обращается к узлу имен, используя RPC, для определения местонахождения нескольких начальных блоков файла (шаг 2). Для каждого блока узел имен возвращает адреса узлов данных, содержащих копию блока. Кроме того, узлы данных сортируются по степени близости к клиенту (в соответствии с топологией сети кластера; см. «Сетевая топология», с. 387). Если клиент сам является узлом данных (например, в случае задач MapReduce), он читает данные из локального узла данных, если последний содержит копию блока (также см. рис. 2.2).



Рис. 3.2. Чтение данных клиентом из HDFS

Объект `DistributedFileSystem` возвращает объект `FSDataInputStream` (входной поток с поддержкой позиционирования), из которого клиент читает данные. В свою очередь, `FSDataInputStream` инкапсулирует объект `DFSInputStream`, управляющий вводом/выводом узлов данных и узла имен.

Затем клиент вызывает для потока `read()` (шаг 3). Объект `DFSInputStream`, располагающий адресами узлов данных нескольких начальных блоков файла, подключается к первому (ближайшему) узлу данных для получения первого блока. Данные передаются с узла данных клиенту, который многократно вызывает `read()` для потока (шаг 4). При достижении конца блока `DFSInputStream` закрывает подключение к узлу данных, а затем находит лучший узел данных для следующего блока (шаг 5). Все происходящее прозрачно для клиента, с точки зрения которого данные просто читаются из непрерывного потока.

Блоки читаются по порядку, объект `DFSInputStream` открывает новые подключения к узлам данных по мере чтения. При этом он обращается к узлу имен за информацией о местонахождении узлов данных для следующей группы блоков, когда в них возникнет необходимость. Завершив чтение, клиент вызывает `close()` для `FSDataInputStream` (шаг 6).

Если в процессе чтения при взаимодействии с узлом данных происходит ошибка, `DFSInputStream` пытается использовать следующий ближайший узел данных для этого блока. Он также запоминает сбойные узлы данных и не пытается снова использовать их для последующих блоков. `DFSInputStream` также проверяет контрольные суммы данных, полученных им от узла данных. Обнаружив поврежденный блок, он сообщает о нем узлу имен, прежде чем пытаться читать копию блока с другого узла данных. Важная особенность этой архитектуры заключается в том, что клиент напрямую обращается к узлам данных для получения данных, а узел имен помогает ему подобрать лучший узел данных для каждого блока. Такая архитектура обеспечивает масштабирование HDFS для большого количества одновременно обслуживаемых клиентов, потому что трафик данных по всем узлам данных в кластере. При этом узел имен только обслуживает запросы местонахождения блоков (делая это чрезвычайно эффективно, поскольку информация хранится в памяти) и не пытается предоставлять данные, чтобы не создавать «узких мест» при увеличении количества клиентов.

СЕТЕВАЯ ТОПОЛОГИЯ И HADOOP

Что следует понимать под «близостью» двух узлов локальной сети? В контексте крупномасштабной обработки данных ограничивающим фактором является скорость передачи данных между узлами. Пропускная способность канала может использоваться как метрика расстояния между двумя узлами.

Но вместо измерения пропускной способности, что бывает нелегко сделать на практике (для этого нужен свободный кластер, а количество пар узлов в кластере растет в квадратичной зависимости от количества узлов), Hadoop использует более простое решение: сеть представляется в виде дерева, а расстояние между двумя узлами оценивается как сумма их расстояний до ближайшего общего предка. Уровни не определяются заранее, но обычно в дерево включаются уровни, соответствующие центру обработки данных, сегменту и узлу, на котором работает процесс. Выбор обусловлен последовательным снижением доступной пропускной способности в следующих сценариях:

- Процессы на одном узле.
- Разные узлы одного сегмента.

- Узлы разных сегментов одного центра обработки данных.
- Узлы разных центров обработки данных.

Для примера возьмем узел n1 в сегменте r1 центра обработки данных d1, для представления которого используется запись /d1/r1/n1. Примеры расстояний для четырех сценариев:

- $\text{distance}(/d1/r1/n1, /d1/r1/n1) = 0$ (процессы на одном узле);
- $\text{distance}(/d1/r1/n1, /d1/r1/n2) = 2$ (разные узлы одного сегмента);
- $\text{distance}(/d1/r1/n1, /d1/r2/n3) = 4$ (узлы разных сегментов одного центра обработки данных);
- $\text{distance}(/d1/r1/n1, /d2/r3/n4) = 6$ (узлы разных центров обработки данных).

Сценарии схематично представлены на рис. 3.3.

Наконец, важно понимать, что Hadoop не сможет определить сетевую топологию без вашей помощи. О том, как настраивается топология, рассказано в разделе «Сетевая топология» на с. 387. По умолчанию считается, что сеть является плоской (имеет одноуровневую иерархию), то есть все узлы находятся в одном сегменте одного центра обработки данных. В малых кластерах это может быть действительно так, в этом случае дополнительная настройка не понадобится.

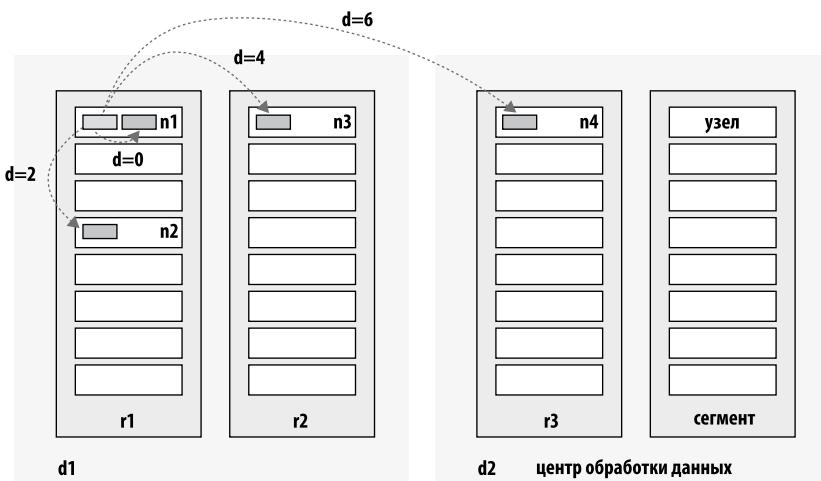
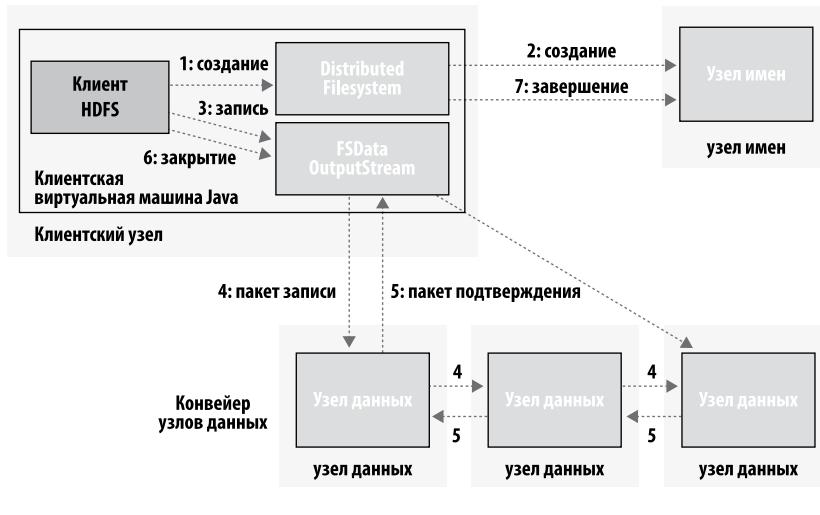


Рис. 3.3. Расстояния в Hadoop

Запись в файлы

Перейдем к рассмотрению записи в файлы HDFS. Мы рассмотрим ситуацию с созданием нового файла, записью данных в него и закрытием (рис. 3.4).



4

Рис. 3.4. Запись данных клиентом из HDFS

Клиент создает файл, вызывая метод `create()` объекта `DistributedFileSystem` (шаг 1 на рис. 3.4). `DistributedFileSystem` обращается с вызовом RPC к узлу имен для создания в пространстве имен файловой системы нового файла, с которым не связан ни один блок (шаг 2). Узел имен выполняет различные проверки: он убеждается в том, что файл не существует, а клиент обладает необходимыми разрешениями для его создания. Если все проверки проходят успешно, узел имен сохраняет информацию о новом файле; в противном случае операция создания файла завершается неудачей и у клиента происходит исключение `IOException`. `DistributedFileSystem` возвращает клиенту объект `FSDataOutputStream`, в который можно записывать данные. Как и при чтении, `FSDataOutputStream` содержит объект `DFSOutputStream`, который обеспечивает все взаимодействия с узлами данных и узлом имен.

Во время записи данных клиентом (шаг 3) `DFSOutputStream` разбивает их на пакеты, которые ставятся во внутреннюю очередь, называемую *очередью данных*. Очередь данных обслуживается объектом `DataStreamer`, который обращается к узлу имен с запросом на выделение новых блоков; для этого узел имен должен

выбрать узлы данных, подходящие для хранения реплик. Список узлов данных образует конвейер. Будем считать, что коэффициент репликации равен 3, так что конвейер состоит из трех узлов. `DataStreamer` отправляет пакеты первому узлу данных в конвейере, тот сохраняет пакеты и пересыпает их второму узлу данных в конвейере. Аналогичным образом второй узел данных сохраняет пакеты и пересыпает их третьему (и последнему) узлу данных в конвейере (шаг 4).

`DFSOutputStream` также поддерживает внутреннюю очередь пакетов, ожидающих подтверждения со стороны узлов данных; она называется *очередью подтверждения*. Пакет удаляется из очереди подтверждения только после того, как он будет подтвержден всеми узлами данных в конвейере (шаг 5).

Если в узле данных произойдет сбой во время записи данных, выполняются следующие действия (полностью прозрачные для клиента, записывающего данные): прежде всего, конвейер закрывается, а все пакеты в очереди подтверждения добавляются в начало очереди данных, чтобы избежать потери пакетов узлами данных за сбийным узлом. Текущий блок на работоспособных узлах данных получает новую идентификационную информацию, которая передается узлу имен, так что частично записанный блок на сбийном узле данных будет удален при последующем восстановлении работоспособности узла. Сбийный узел данных удаляется из конвейера, а оставшаяся часть данных блока записывается в два работоспособных узла данных в конвейере. Узел имен замечает, что количество реплик блока недостаточно, и организует создание дополнительной реплики на другом узле. После этого все последующие блоки обрабатываются по обычной схеме.

Во время записи блока теоретически возможен (хотя и маловероятен) сбой сразу нескольких узлов данных. При условии записи `dfs.replication.min` реплик (по умолчанию 1) операция завершается успешно, а блок асинхронно реплицируется в кластере до достижения целевого коэффициента репликации (`dfs.replication`, по умолчанию используется значение 3).

Завершив запись данных, клиент вызывает для потока `close()` (шаг 6). Все оставшиеся пакеты направляются в конвейер узлов данных, а связь с узлом имен осуществляется после подтверждения завершения записи файла (шаг 7). Узел имен уже знает, из каких блоков состоит файл (информация передается `DataStreamer` при запросах на распределение блоков); остается лишь дождаться минимальной репликации блоков, после чего возвращается признак успешного завершения операции.

РАЗМЕЩЕНИЕ РЕПЛИК

Каким образом узел имен выбирает узлы данных, на которых должны храниться реплики? Решение требует компромисса между надежностью и затратами пропускной способности операций чтения/записи. Например, размещение всех реплик на одном узле означает минимальные затраты пропускной способности записи, так как канал репликации работает на одном узле, но и реальной избыточности в этом случае не будет (в случае сбоя узла все данные блока будут потеряны). Кроме того, внесегментное чтение создаст повышенную нагрузку на канал связи. С другой стороны, размещение реплик в разных центрах обработки данных обеспечит максимальную избыточность, но за нее придется расплачиваться затратами пропускной способности. Даже в пределах одного центра обработки данных (а все кластеры Hadoop на момент написания книги работают именно в такой конфигурации) существуют разные стратегии размещения. В Hadoop версии 0.17.0 была выбрана новая стратегия размещения, обеспечивающая относительно равномерное распределение блоков в кластере (за информацией о распределении нагрузки в кластерах обращайтесь к разделу «Balancer», с. 449). А в версиях после 1.x политика размещения блоков определяется заменяемыми модулями.

По умолчанию Hadoop размещает первую реплику в одном узле с клиентом (для клиентов, работающих за пределами кластера, узел выбирается случайным образом, хотя система старается не выбирать слишком загруженные или заполненные узлы). Вторая реплика размещается в случайно выбранном сегменте, отличном от первого. Третья реплика размещается в одном сегменте со второй, но на другом, случайно выбираемом узле. Остальные реплики размещаются на случайных узлах кластера, хотя система по возможности избегает размещения слишком большого количества реплик в одном сегменте.

После выбора местонахождения реплик система переходит к построению конвейера с учетом топологии сети. С коэффициентом репликации 3 конвейер выглядит примерно так, как показано на рис. 3.5. В целом эта стратегия обеспечивает хороший баланс между надежностью (блоки хранятся в двух сегментах), затратами пропускной способности при записи (операция записи проходит через один сетевой коммутатор), производительностью чтения (данные могут читаться из двух сегментов на выбор) и распределением блоков по кластеру (клиенты записывают только один блок в локальном сегменте).

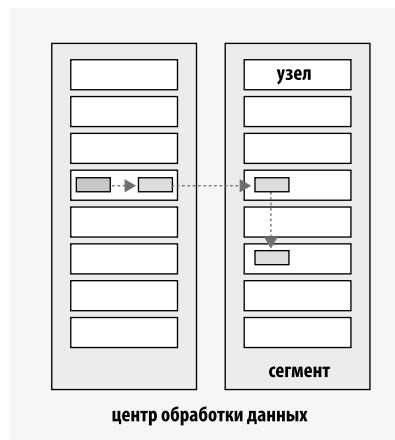


Рис. 3.5. Типичный конвейер репликации

Модель целостности

Модель целостности файловой системы описывает видимость данных при операциях чтения и записи. HDFS нарушает некоторые требования POSIX в пользу производительности, поэтому поведение некоторых операций может отличаться от того, к которому вы привыкли.

Созданный файл, как и следует ожидать, виден в пространстве имен файловой системы:

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```

Однако видимость содержимого, записанного в файл, не гарантирована — даже после сброса потока. Все выглядит так, словно файл имеет нулевую длину:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

После того, как объем записанных данных достигнет размера блока, первый блок становится видимым для новых читателей. Это правило распространяется и на все последующие блоки: текущий записываемый блок невидим для других операций чтения.

HDFS предоставляет возможность принудительной синхронизации всех буферов по узлам данных — для этого следует вызвать метод `sync()` класса `FSDataOutputStream`. HDFS гарантирует, что после успешного возвращения управления `sync()` данные, записанные до указанной позиции файла, достигли всех узлов данных в конвейере записи и стали видимыми для всех новых операций чтения¹:

```
Path p = new Path("p");
FSDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
out.sync();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

Такое поведение напоминает системный вызов `fsync` в POSIX, закрепляющий буферизованные данные для файлового дескриптора. Например, при использовании стандартного Java API для записи локального файла записанные данные гарантированно станут видны только после сброса потока и синхронизации:

```
FileOutputStream out = new FileOutputStream(localFile);
out.write("content".getBytes("UTF-8"));
out.flush(); // Сброс потока
out.getFD().sync(); // Синхронизация с диском
assertThat(localFile.length(), is(((long) "content".length())));
```

При закрытии файла в HDFS также выполняется неявный вызов `sync()`:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.close();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

Последствия для проектирования приложений

Модель целостности влияет на проектирование приложений. Если в приложении не будут включены вызовы `sync()`, приготовьтесь к возможной потере блока данных в случае сбоя клиента или системы. Во многих приложениях такая потеря недопустима; в таких случаях следует вставить вызовы `sync()` в подходящие точки (например, после вывода определенного количества записей или байтов). Хотя

¹ В Hadoop версий 1.x и выше метод `sync()` считается устаревшим; вместо него следует использовать метод `hflush()`. Также был добавлен метод `hsync()`, который дает более надежные гарантии того, что операционная система сбросила информацию на диски узлов данных (по аналогии с POSIX `fsync`). Однако на момент написания книги метод еще не был реализован и ограничивался простым вызовом `hflush()`.

операция `sync()` спроектирована так, чтобы не создавать лишней нагрузки на HDFS, она все же сопряжена с некоторыми затратами, так что вы снова сталкиваетесь с компромиссом между надежностью и нагрузкой на канал связи. Какое именно решение можно считать приемлемым — зависит от приложения. Чтобы выбрать оптимальное соотношение, измерьте производительность вашего приложения с разной частотой `sync()`.

Перемещение данных: Flume и Sqoop

Не торопитесь писать собственное приложение для перемещения данных в HDFS. Для начала рассмотрите возможность применения готовых инструментов, потому что в них учтены многие распространенные требования.

Apache Flume (<http://incubator.apache.org/flume/>) — система перемещения больших объемов потоковых данных в HDFS. Очень типичный пример использования — сбор журнальных данных в одной системе (например, банка веб-серверов) и их накопление в HDFS для последующего анализа. Flume поддерживает разнообразные источники данных, включая *tail* (данные, записываемые в локальный файл, направляются в Flume — как и при использовании *tail* в Unix), *syslog* и Apache *log4j* (Java-приложения могут записывать события в файлы HDFS через Flume).

Узлы Flume могут образовывать произвольные топологии. Обычно на каждом исходном компьютере (например, на каждом веб-сервере) работает узел; узлы объединяются в уровни, через которые проходят данные на своем пути в HDFS.

Flume поддерживает разные степени надежности доставки, от максимально доступного (любые сбои узлов Flume считаются недопустимыми) до сквозного (доставка гарантируется даже в случае сбоя нескольких узлов Flume между источником и HDFS).

Проект Apache Sqoop (<http://sqoop.apache.org/>) разрабатывался для выполнения массового импортирования данных из структурированных хранилищ (например, реляционных баз данных) в HDFS. Например, Sqoop уместно использовать в организации, которая каждую ночь импортирует данные за день из основной базы данных в склад данных Hive для анализа. Технология Sqoop рассматривается в главе 15.

Параллельное копирование с использованием distcp

Модели обращений к HDFS, рассматривавшиеся до сих пор, ориентировались на однопоточную модель. Работа с наборами файлов (например, с указанием

шаблонов имен файлов) возможна, но для эффективной параллельной обработки таких файлов вам придется написать программу самостоятельно. В комплект поставки Hadoop входит полезная программа *distcp*, предназначенная для параллельного копирования больших объемов данных в файловые системы Hadoop и из них.

Классический пример использования *distcp* — передача данных между двумя кластерами HDFS. Если в кластерах выполняются идентичные версии Hadoop, наиболее подходящей будет схема *hdfs*:

```
% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

Каталог */foo* (со всем содержимым) копируется из первого кластера в каталог */bar* во втором кластере, так что во втором кластере появляется структура каталогов */bar/foo*. Если каталог */bar* не существует, он будет предварительно создан. Если в командной строке *distcp* указано несколько исходных путей, все они будут скопированы в приемник. Исходные пути должны быть абсолютными.



По умолчанию *distcp* пропускает файлы, уже существующие в приемнике; чтобы такие файлы заменялись, следует указать параметр *-overwrite*. Также можно обновлять только изменившиеся файлы (параметр *-update*).

Использование параметра *-overwrite* или *-update* (или обоих) изменяет интерпретацию путей источника и приемника. Сказанное лучше всего пояснить на примере. Если изменить файл в подкаталоге */foo* первого кластера из предыдущего примера, синхронизация изменений со вторым кластером может быть выполнена следующей командой:

```
% hadoop distcp -update hdfs://namenode1/foo hdfs://namenode2/bar/foo.
```

Дополнительный завершающий подкаталог */foo* необходим, потому что в этом случае содержимое каталога-источника копируется в содержимое каталога-приемника.

Если вы не уверены в том, к каким последствиям приведет выполнение операции *distcp*, сначала опробуйте ее на небольшом тестовом дереве каталогов.

Поведением *distcp* управляют многочисленные параметры, которые, в частности, позволяют сохранять атрибуты файлов, игнорировать сбои и ограничивать количество копируемых файлов или общий объем данных. Чтобы получить справку по параметрам, запустите программу без параметров.

Команда *distcp* реализована как задание MapReduce, а фактическая работа по копированию выполняется отображениями, параллельно работающими в кластере

(без сверток). Каждый файл копируется одним отображением; при этом *distcp* старается предоставить каждому отображению примерно одинаковый объем данных.

Количество отображений определяется следующим образом. Так как каждому отображению для копирования желательно предоставить разумный объем данных, чтобы свести к минимуму дополнительные затраты ресурсов при создании задач, каждое отображение копирует не менее 256 Мбайт (если только общий размер входных данных не меньше этой величины; в этом случае одно отображение выполняет всю работу). Например, для файлов общим размером 1 Гбайт будут созданы четыре задачи отображения. При очень большом размере данных становится необходимо ограничить количество отображений, чтобы свести к минимуму нагрузку на каналы связи и кластеры. По умолчанию максимальное количество отображений равно 20 на узел кластера. Например, при копировании 1000 Гбайт в кластер из 100 узлов будет создано 2000 отображений (по 20 на узел), так что каждое скопирует в среднем 512 Мбайт. Количество задач можно сократить, включив в командную строку *distcp* параметр *-m*. Например, с параметром *-m 1000* будет создано 1000 отображений, каждое из которых скопирует в среднем 1 Гбайт.

При использовании *distcp* между двумя кластерами HDFS, в которых работают разные версии, попытка использования протокола *hdfs* завершится неудачей из-за несовместимости систем RPC. Проблему можно решить использованием для чтения из источника файловой системы HFTP на базе HTTP. Задание должно выполняться в кластере-приемнике, чтобы версии RPC были совместимы. С использованием HFTP предыдущий пример выглядит так:

```
% hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar.
```

Обратите внимание на необходимость указания веб-порта узла имен в исходном URI. Значение определяется свойством *dfs.http.address*, которое по умолчанию равно 50070.

Новый протокол *webhdfs* (заменивший *hftp*) может использоваться для обоих кластеров, источника и приемника, без каких-либо проблем с совместимостью:

```
% hadoop distcp webhdfs://namenode1:50070/foo webhdfs://namenode2:50070/bar.
```

Также можно использовать в качестве источника или приемника *distcp* HTTP-посредника HDFS; к преимуществам этого решения относится возможность управления фильтрацией трафика и пропускной способностью (см. «HTTP», с. 93).

Сбалансированность кластеров HDFS

При копировании данных в HDFS важно учитывать сбалансированность кластера. HDFS лучше всего работает при равномерном распределении файловых блоков по кластеру, поэтому желательно позаботиться о том, чтобы при выполнении *distcp* это распределение не нарушалось. Вернемся к примеру с копированием 1000 Гбайт: с параметром *-m 1* копирование будет выполнено одним отображением, при котором — помимо низкой скорости и неэффективного использования ресурсов кластера — первая реплика каждого блока будет находиться в узле, в котором работает отображение (до заполнения диска). Вторая и третья реплики будут распределяться по кластеру, но один узел станет несбалансированным. Если количество отображений будет больше количества узлов в кластере, проблема исчезнет. По этой причине лучше начать с запуска *distcp* с 20 отображениями на узел (значение по умолчанию).

Впрочем, предотвратить разбалансирование кластера удается не всегда. Например, количество отображений может ограничиваться для того, чтобы некоторые узлы могли использоваться другими заданиями. В таких ситуациях для равномерного распределения блоков в кластере приходится использовать программу балансировки (см. «Balancer», с. 449).

HAR

В HDFS малые файлы хранятся неэффективно, поскольку каждый файл занимает блок, а метаданные блока должны храниться в памяти узла имен. Таким образом, многочисленные мелкие файлы поглощают слишком много памяти на узле имен. (Однако следует помнить о том, что затраты дискового пространства для хранения малых файлов не превышают того, что необходимо для хранения непосредственного содержимого файла; например, 1-мегабайтный файл с размером блока 128 Мбайт использует 1 Мбайт дискового пространства, а не 128.)

HAR (Hadoop ARchives) — инструмент архивации файлов, повышающий эффективность упаковки файлов в блоки HDFS, тем самым сокращающий затраты памяти без потери прозрачности доступа к файлам. В частности, Hadoop Archives может использоваться в качестве входного источника для MapReduce.

Использование HAR

Архив Hadoop создается из набора файлов программой *archive*. Программа запускает задание MapReduce для параллельной обработки входных файлов,

соответственно, для ее запуска необходим работающий кластер MapReduce. Допустим, мы хотим заархивировать несколько файлов в HDFS:

```
% hadoop fs -lsr /my/files
-rw-r--r-- 1 tom supergroup      1 2009-04-09 19:13 /my/files/a
drwxr-xr-x - tom supergroup    0 2009-04-09 19:13 /my/files/dir
-rw-r--r-- 1 tom supergroup      1 2009-04-09 19:13 /my/files/dir/b
```

Для этого выполняется следующая команда *archive*:

```
% hadoop archive -archiveName files.har /my/files /my
```

В первом параметре передается имя архива (*files.har* в данном случае). Файлы HAR всегда имеют расширение *.har*; вскоре вы увидите, почему оно является обязательным. Далее перечисляются файлы, включаемые в архив. В нашем примере архивируется только одно исходное дерево — файлы из каталога */my/files* в HDFS, но программа позволяет задать несколько деревьев. В последнем аргументе задается выходной каталог для файла HAR. Посмотрим, что создаст программа *archive*:

```
% hadoop fs -ls /my
Found 2 items
drwxr-xr-x - tom supergroup      0 2009-04-09 19:13 /my/files
drwxr-xr-x - tom supergroup      0 2009-04-09 19:13 /my/files.har
% hadoop fs -ls /my/files.har
Found 3 items
-rw-r--r-- 10 tom supergroup     165 2009-04-09 19:13 /my/files.har/_index
-rw-r--r-- 10 tom supergroup      23 2009-04-09 19:13
                               /my/files.har/_masterindex
-rw-r--r-- 1 tom supergroup       2 2009-04-09 19:13 /my/files.har/part-0
```

Из полученного результата видно, что файл HAR состоит из двух индексных файлов и набора файлов частей (в нашем примере такой файл всего один). В файлах частей находится сцепленное содержимое исходных файлов, а в индексе хранится информация о файле части, в котором хранится заархивированный файл, смещении и длине. Впрочем, вся эта информация остается скрытой от приложения, которое использует для взаимодействия с файлами архивов URI-схему *har*; при этом используется файловая система HAR, работающая поверх базовой файловой системы (HDFS в нашем случае). Следующая команда выводит рекурсивный список содержимого архива:

```
% hadoop fs -lsr har:///my/files.har
drw-r--r-- - tom supergroup      0 2009-04-09 19:13 /my/files.har/my
drw-r--r-- - tom supergroup      0 2009-04-09 19:13 /my/files.har/my/files
-rw-r--r-- 10 tom supergroup      1 2009-04-09 19:13 /my/files.har/my/files/a
```

```
drw-r--r-- - tom supergroup      0 2009-04-09 19:13  
                  /my/files.har/my/files/dir  
-rw-r--r-- 10 tom supergroup     1 2009-04-09 19:13  
                  /my/files.har/my/files/dir/b
```

Если файл HAR находится в файловой системе по умолчанию, все просто. С другой стороны, если вы хотите обратиться к файлу HAR в другой файловой системе, придется использовать другую форму путевого URI-адреса. Например, следующие две команды приводят к одинаковым результатам:

```
% hadoop fs -lsr har:///my/files.har/my/files/dir  
% hadoop fs -lsr har://localhost:8020/my/files.har/my/files/dir
```

Обратите внимание: во второй форме для обозначения файловой системы HAR также используется схема *har*, но в части полномочий указана базовая система *hdfs*, за ней следуют дефис, хост HDFS (*localhost*) и порт (8020). Теперь понятно, почему файлы HAR должны иметь расширение *.har*. Чтобы преобразовать URI *har* в URI базовой файловой системы, файловая система HAR анализирует полномочия и путь до компонента с расширением *.har* включительно. В нашем примере это часть *hdfs://localhost:8020/my/files.har*. Оставшаяся часть пути определяет путь файла в архиве: */my/files/dir*.

Для удаления файла HAR используется рекурсивная форма *delete*, потому что с точки зрения базовой файловой системы файл HAR представляет собой каталог:

```
% hadoop fs -rmr /my/files.har
```

Ограничения

При работе с файлами HAR необходимо учитывать некоторые ограничения. При создании архива создаются копии исходных файлов, поэтому для хранения архива потребуется столько же дискового пространства, как для архивируемых файлов (хотя после того, как архив будет создан, оригиналы можно удалить). В настоящее время сжатие архивов не поддерживается, хотя файлы, включаемые в архив, могут быть сжатыми (файлы HAR в этом отношении похожи на файлы *tar*).

Архив не может изменяться после создания. Чтобы добавить или удалить файлы, архив придется создать заново. На практике это ограничение не создает проблем для файлов, не изменяющихся после записи, так как они могут регулярно архивироваться в пакетном режиме — например, ежедневно или еженедельно.

Как упоминалось ранее, файлы HAR могут использоваться в качестве входных данных для MapReduce. Однако не существует версии *InputFormat* с поддержкой

архивов, способной упаковывать несколько файлов в один сплит MapReduce, поэтому обработка большого количества мелких файлов даже в виде файла HAR все равно может оказаться неэффективной. В разделе «Малые файлы и CombineFileInputFormat» на с. 315 рассматривается другое решение этой проблемы.

Наконец, если даже после всех мер по сокращению количества мелких файлов в системе вы сталкиваетесь с проблемой нехватки памяти на узле имен, рассмотрите возможность применения HDFS Federation для разбиения пространства имен (см. «HDFS Federation», с. 84).

4 Ввод/вывод в Hadoop

Hadoop поставляется с набором примитивов для ввода/вывода данных. Некоторые из используемых механизмов (например, целостность данных и сжатие) имеют более широкое применение помимо Hadoop, но заслуживают особого внимания при работе с терабайтными наборами данных. Другие — инструменты Hadoop или API, образующие структурные элементы для разработки распределенных систем (например, среды сериализации и дисковые структуры данных).

Целостность данных

Пользователи Hadoop с полным правом ожидают, что данные не будут повреждены или потеряны при хранении или обработке. Но поскольку каждая операция ввода/вывода на диске или в сети сопряжена с небольшой вероятностью появления ошибок в читаемых или записываемых данных, при прохождении через систему таких огромных объемов данных вероятность их повреждения достаточно велика.

Стандартный механизм обнаружения поврежденных данных основан на вычислении контрольной суммы при поступлении данных в систему и при всех последующих передачах по ненадежному каналу, способному повредить данные. Если заново сгенерированная контрольная сумма отличается от исходной, данные считаются поврежденными. Механизм контрольных сумм не предоставляет возможности исправления данных — он только обнаруживает ошибки. Учтите,

что поврежденной может оказаться контрольная сумма, а не сами данные, но эта ситуация крайне маловероятна, так как контрольная сумма существенно меньше данных.

Для обнаружения ошибок чаще всего применяется код CRC-32 (Cyclic Redundancy Check), вычисляющий 32-разрядную целочисленную контрольную сумму для входных данных произвольного размера.

Целостность данных в HDFS

В HDFS организовано прозрачное вычисление контрольных сумм всех записываемых данных. По умолчанию контрольная сумма проверяется при чтении данных. Отдельная контрольная сумма генерируется для каждого `io.bytes.per.checksum` байт данных — по умолчанию 512 байт. Длина контрольной суммы CRC-32 равна 4 байтам, и затраты на ее хранение составляют менее 1%.

Узлы данных отвечают за проверку получаемых данных перед сохранением самих данных и их контрольных сумм. Это относится к данным, полученным от клиентов и других узлов данных в ходе репликации. Клиент, записывающий данные, отправляет их в конвейер узлов данных (см. главу 3); последний узел данных в конвейере проверяет контрольную сумму. При обнаружении ошибки клиент получает исключение `ChecksumException` (субкласс `IOException`), которое обрабатывается в соответствии со спецификой приложения — например, попыткой повторного выполнения операции.

Когда клиенты читают данные из узла данных, они также проверяют контрольные суммы, сравнивая их с суммами, хранящимися в узле данных. Каждый узел данных ведет журнал проверок контрольных сумм, поэтому он знает время последней проверки каждого блока. Если проверка блока клиентом проходит успешно, клиент сообщает об этом узлу данных, и тот обновляет свой журнал. Накопленная статистика помогает в выявлении сбоев на дисках.

Помимо проверки блоков при чтении данных клиентом, на каждом узле данных в фоновом режиме работает процесс `DataBlockScanner`, который периодически проверяет все блоки, хранящиеся на узле данных. Эта мера защищает от физических дефектов носителей информации. О том, как получить доступ к отчетам сканера, рассказано в разделе «Сканирование блоков на узлах данных», с. 447.

Так как HDFS хранит реплики блоков, поврежденные блоки можно «восстановить» копированием одной из неповрежденных реплик. Если клиент обнаруживает ошибку при чтении блока, он передает узлу имен информацию о поврежденном блоке и узле данных, с которого он пытался читать, перед выдачей исключения

`ChecksumException`. Узел имен помечает реплику блока как поврежденную, чтобы клиенты не обращались к ней и не пытались скопировать ее на другой узел данных. Затем он планирует копирование блока на другой узел данных, чтобы коэффициент репликации вернулся к заданному уровню. Когда все это будет сделано, поврежденная реплика удаляется.

Проверку контрольных сумм можно отключить, вызвав метод `setVerifyChecksum()` класса `FileSystem` с параметром `false` перед вызовом метода `open()` для чтения файла. Чтобы добиться того же результата в управляющей оболочке, используйте параметр `-ignoreCrc` с `-get` или эквивалентной командой `-copyToLocal`. Данная возможность будет полезна, если у вас имеется поврежденный файл и вы хотите просмотреть его, чтобы решить, что с ним делать дальше. Например, вы можете проверить, нельзя ли восстановить какие-либо данные из файла.

LocalFileSystem

Класс Hadoop `LocalFileSystem` выполняет проверку контрольных сумм на стороне клиента. Это означает, что при записи файла с именем `filename` клиент файловой системы в прозрачном режиме создает в том же каталоге скрытый файл `filename.crc` с контрольными суммами для каждого фрагмента файла. Размер фрагмента, как и в HDFS, определяется свойством `io.bytes.per.checksum`, по умолчанию равным 512 байтам. Размер фрагмента хранится в виде метаданных в файле `.crc`, так что файл может быть правильно прочитан даже после изменения размера фрагмента. Контрольные суммы проверяются при чтении файла, и при обнаружении ошибки `LocalFileSystem` выдает исключение `ChecksumException`.

Вычисление контрольных сумм обходится относительно дешево (в Java оно реализовано в платформенном коде) — как правило, время чтения или записи увеличивается на несколько процентов. В большинстве приложений это невысокая цена за целостность данных. Однако при необходимости контрольные суммы можно отключить — например, если их вычисление встроено в базовую файловую систему. Для этого следует использовать `RawLocalFileSystem` вместо `LocalFileSystem`. Чтобы провести замену глобально в масштабах приложения, достаточно переключить реализацию URI `file`, задав свойству `fs.file.impl` значение `org.apache.hadoop.fs.RawLocalFileSystem`. Также можно напрямую создать экземпляр `RawLocalFileSystem`, что может быть удобно при отключении проверки контрольных сумм только для отдельных операций чтения, например:

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs.initialize(null, conf);
```

ChecksumFileSystem

В работе `LocalFileSystem` используется класс `ChecksumFileSystem`. Он позволяет легко добавить проверку контрольных сумм в другие файловые системы, в которых такая поддержка отсутствует, так как `ChecksumFileSystem` представляет собой простую обертку для `FileSystem`. Общая идома использования выглядит так:

```
FileSystem rawFs = ...
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

Объект базовой файловой системы может быть получен вызовом метода `getRawFileSystem()` для `ChecksumFileSystem`. `ChecksumFileSystem` содержит и другие, более полезные методы для работы с контрольными суммами — например, метод `getChecksumFile()`, возвращающий путь к файлу контрольных сумм для любого файла. За информацией о других методах обращайтесь к документации.

Если во время чтения файла `ChecksumFileSystem` обнаруживает ошибку, вызывается метод `reportChecksumFailure()`. Реализация по умолчанию не делает ничего, но `LocalFileSystem` перемещает поврежденный файл и его контрольные суммы в служебный каталог *bad_files* на том же устройстве. Администратор должен периодически проверять наличие поврежденных файлов и что-то делать с ними.

Сжатие

Сжатие файлов обладает двумя основными преимуществами: оно сокращает затраты дискового пространства на хранение файлов и ускоряет передачу данных по сети, на диск или с него. При больших объемах данных оба вида экономии достаточно важны, поэтому вы должны тщательно продумать применение сжатия в Hadoop.

Существует много разных форматов, инструментов и алгоритмов сжатия, обладающих разными характеристиками. В табл. 4.1 перечислены наиболее распространенные форматы, используемые с Hadoop.

Таблица 4.1. Сводка форматов сжатия

Формат сжатия	Программа	Алгоритм	Расширение	Поддержка разбиения
DEFLATE ^a	—	DEFLATE	.deflate	Нет
gzip	gzip	DEFLATE	.gz	Нет
bzip2	bzip2	bzip2	.bz2	Да

Формат сжатия	Программа	Алгоритм	Расширение	Поддержка разбиения
LZO	LZOp	LZO	.LZO	Нет ^b
LZ4	–	LZ4	.lz4	Нет
Snappy	–	Snappy	.snappy	Нет

^a DEFLATE — алгоритм сжатия, стандартной реализацией которого является zlib. Распространенной программы командной строки для создания файлов в формате DEFLATE не существует, на практике чаще используется формат gzip (DEFLATE с дополнительными заголовками и завершителем). Расширение .deflate входит в систему правил Hadoop.

^b Файлы LZO могут разбиваться, если они были предварительно проиндексированы. См. с. 135

Любой алгоритм сжатия экономит место за счет скорости обработки: при быстром сжатии и восстановлении файл обычно занимает больше места. Программные инструменты, перечисленные в табл. 4.1, обычно позволяют в некоторой степени управлять этим соотношением. Они поддерживают девять разных параметров: -1 — оптимизация по скорости, а -9 — оптимизация размера файла. Следующая команда создает сжатый файл file.gz с использованием самого быстрого метода сжатия:

```
gzip -1 file
```

Программы сильно отличаются по характеристикам сжатия. Gzip — программа общего назначения со средними характеристиками. Bzip2 сжимает файлы более эффективно, чем gzip, но работает медленнее. Скорость восстановления данных bzip2 выше скорости сжатия, но она все равно работает медленнее других форматов. С другой стороны, LZO, LZ4 и Snappy оптимизированы по скорости: они работают на порядок быстрее gzip, но сжатие производится с меньшей эффективностью. По скорости восстановления сжатых данных Snappy и LZ4 также значительно превосходят LZO¹.

В столбце «Поддержка разбиения» табл. 4.1 указано, поддерживает ли формат разбиение, то есть можно ли выполнить позиционирование к произвольной точке потока и начать чтение с некоторой позиции, следующей за ней. Форматы сжатия

¹ По адресу <https://github.com/ning/jvm-compressor-benchmark> находится хороший справочник по JVM-совместимым библиотекам (включая некоторые платформенные библиотеки). Инструменты командной строки описаны у Джекфа Гилкрист (Jeff Gilchrist) в статье «Archive Comparison Test» по адресу <http://compression.ca/act/act-summary.html>.

с поддержкой разбиения особенно хорошо подходят для MapReduce; за дополнительной информацией обращайтесь к разделу «Сжатие и разбиение входных данных», с. 135.

Кодеки

Кодек представляет собой реализацию алгоритма сжатия/восстановления. В Hadoop кодек представляется реализацией интерфейса `CompressionCodec`. Например, `GzipCodec` инкапсулирует алгоритм сжатия и восстановления gzip. В табл. 4.2 перечислены кодеки, доступные для Hadoop.

Таблица 4.2. Кодеки сжатия Hadoop

Формат сжатия	Реализация <code>CompressionCodec</code>
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

Библиотеки LZO распространяются на условиях лицензии GPL и не могут включаться в дистрибутивы Apache, поэтому кодеки Hadoop необходимо загружать отдельно по адресу <http://code.google.com/p/hadoop-gpl-compression/> (или <http://github.com/kevinweil/hadoop-lzo>, с исправлениями ошибок и расширенным инструментарием). Кодек `LzopCodec` совместим с программой lzop; фактически это формат LZO с дополнительными заголовками, который обычно используется на практике. Также существует кодек `LzoCodec` для «чистого» формата LZO, для которого используется расширение `.lzo_deflate` (по аналогии с форматом DEFLATE, который представляет собой формат gzip без заголовков).

Сжатие и восстановление потоков с использованием `CompressionCodec`

Интерфейс `CompressionCodec` содержит два метода, которые позволяют легко выполнять сжатие или восстановление данных. Чтобы сжать данные, выводимые в выходной поток, используйте метод `createOutputStream(OutputStream out)` для создания объекта `CompressionOutputStream`. Несжатые данные,

выводимые в этот поток, записываются в сжатой форме в базовый поток. И наоборот, для восстановления прочитанных данных из входного потока вызовите `createInputStream(InputStream in)` для получения объекта `CompressionInputStream`, который используется для чтения несжатых данных из базового потока.

`CompressionOutputStream` и `CompressionInputStream` в целом похожи на `java.util.zip.DeflaterOutputStream` и `java.util.zip.DeflaterInputStream` — кроме того, что в них предусмотрена возможность сброса потока сжатия или восстановления. Данная возможность играет важную роль в приложениях, сжимающих части потока данных в отдельные блоки (как, например, `SequenceFile` — см. «`SequenceFile`», с. 186).

Листинг 4.1 показывает, как использовать API для сжатия данных, прочитанных из стандартного ввода, и записи их в стандартный вывод.

Листинг 4.1. Программа для сжатия данных, прочитанных из стандартного ввода, и записи их в стандартный вывод

```
public class StreamCompressor {  
    public static void main(String[] args) throws Exception {  
        String codecclassname = args[0];  
        Class<?> codecClass = Class.forName(codecclassname);  
        Configuration conf = new Configuration();  
        CompressionCodec codec = (CompressionCodec)  
            ReflectionUtils.newInstance(codecClass, conf);  
  
        CompressionOutputStream out = codec.createOutputStream(System.out);  
        IOUtils.copyBytes(System.in, out, 4096, false);  
        out.finish();  
    }  
}
```

Приложение ожидает, что в первом аргументе командной строки передается полностью уточненное имя реализации `CompressionCodec`. Мы при помощи `ReflectionUtils` конструируем новый экземпляр кодека, а затем получаем обертку сжатия для `System.out`. Наконец, вызов `finish()` для `CompressionOutputStream` сообщает о завершении записи в сжатый поток, но поток при этом не закрывается. Для проверки можно воспользоваться приведенной ниже командной строкой, которая сжимает строку «Text», используя `StreamCompressor` с `GzipCodec`, а затем восстанавливает ее из стандартного ввода при помощи `gunzip`:

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec  
\  
| gunzip -  
Text
```

Определение кодеков с использованием CompressionCodecFactory

Если вы читаете сжатый файл, используемый кодек обычно определяется по расширению файла. Файл с расширением `.gz` читается кодеком `GzipCodec`, и так далее. Расширения разных форматов сжатия перечислены в табл. 4.1.

Метод `getCodec()` класса `CompressionCodecFactory` позволяет связать расширение файла с объектом `CompressionCodec`. Метод получает объект `Path` для соответствующего файла. В листинге 4.2 представлено приложение, использующее данную возможность для восстановления сжатого файла.

Листинг 4.2. Программа для восстановления сжатого файла с использованием кодека, определенного по расширению файла

```
public class FileDecompressor {
    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);

        Path inputPath = new Path(uri);
        CompressionCodecFactory factory = new CompressionCodecFactory(conf);
        CompressionCodec codec = factory.getCodec(inputPath);
        if (codec == null) {
            System.err.println("No codec found for " + uri);
            System.exit(1);
        }
        String outputUri =
            CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());

        InputStream in = null;
        OutputStream out = null;
        try {
            in = codec.createInputStream(fs.open(inputPath));
            out = fs.create(new Path(outputUri));
            IOUtils.copyBytes(in, out, conf);
        } finally {
            IOUtils.closeStream(in);
            IOUtils.closeStream(out);
        }
    }
}
```

После определения кодека суффикс файла отделяется для формирования имени выходного файла (при этом используется статический метод `removeSuffix()`

класса `CompressionCodecFactory`). В нашем примере команда запуска программы для восстановления файла с именем `file.gz` выглядит так:

```
% hadoop FileDecompressor file.gz
```

`CompressionCodecFactory` находит кодеки в списке, определяемом конфигурационным свойством `io.compression.codecs`. По умолчанию в список включаются все кодеки, предоставляемые Hadoop (табл. 4.3), так что придется изменять его только при регистрации дополнительных кодеков (внешних кодеков LZO). Для каждого кодека известно его расширение по умолчанию, что позволяет `CompressionCodecFactory` проводить поиск расширений среди зарегистрированных кодеков.

Таблица 4.3. Зарегистрированные кодеки

Имя свойства	Тип	Значение по умолчанию	Описание
io.compression.codecs	Имена классов, разделенные запятыми	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec	Список классов <code>CompressionCodec</code> для сжатия/восстановления файлов

Платформенные библиотеки

По соображениям быстродействия для выполнения операций сжатия и восстановления рекомендуется использовать платформенные библиотеки. Например, в одном тесте платформенные библиотеки `gzip` сократили время восстановления на 50%, а время сжатия — на 10% (по сравнению с встроенной реализацией Java). В табл. 4.4 содержится информация о доступности реализаций Java и платформенных реализаций для каждого формата сжатия. Некоторые форматы не имеют платформенной реализации (например, `bzip2`), а другие существуют только в платформенном варианте (как LZO).

Таблица 4.4. Реализации библиотек сжатия

Формат сжатия	Реализация на Java	Платформенная реализация
DEFLATE	Да	Да
gzip	Да	Да

продолжение ↗

Таблица 4.4 (продолжение)

Формат сжатия	Реализация на Java	Платформенная реализация
bzip2	Да	Нет
LZO	Нет	Да
LZ4	Нет	Да
Snappy	Нет	Да

Hadoop поставляется с построенными платформенными библиотеками сжатия для 32- и 64-разрядных версий Linux. Для других платформ вам придется откомпилировать библиотеки самостоятельно; инструкции приведены в вики Hadoop по адресу <http://wiki.apache.org/hadoop/NativeHadoop>.

Платформенные библиотеки выбираются при помощи системного свойства Java `java.library.path`. Сценарий `hadoop` в каталоге `bin` задает это свойство за вас; если вы не используете этот сценарий, вам придется задать свойство в своем приложении.

По умолчанию Hadoop ищет платформенные библиотеки для платформы, на которой работает, и если поиск успешен — автоматически загружает их. Это означает, что для использования платформенных библиотек не нужно изменять никакие настройки конфигурации. Однако в некоторых ситуациях требуется запретить использовать платформенные библиотеки (скажем, во время отладки ошибок, связанных со сжатием). Для этого можно задать свойству `hadoop.native.lib` значение `false`, чтобы использовались встроенные эквиваленты Java (если они доступны).

CodecPool

Если вы используете платформенную библиотеку и в приложении выполняются многочисленные операции сжатия и восстановления, рассмотрите возможность использования пула кодеков `CodecPool`.

Это позволит вам повторно использовать объекты сжатия и восстановления, избавившись от затрат на их создание.

Пример использования пула приведен в листинге 4.3 (хотя в этой программе, создающей только один объект `Compressor`, необходимости в создании пула нет).

Листинг 4.3. Программа для сжатия данных, прочитанных из стандартного ввода, и записи их в стандартный вывод с использованием пула объектов сжатия

```
public class PooledStreamCompressor {

    public static void main(String[] args) throws Exception {
```

```
String codecclassname = args[0];
Class<?> codecClass = Class.forName(codecclassname);
Configuration conf = new Configuration();
CompressionCodec codec = (CompressionCodec)
    ReflectionUtils.newInstance(codecClass, conf);
Compressor compressor = null;
try {
    compressor = CodecPool.getCompressor(codec);
    CompressionOutputStream out =
        codec.createOutputStream(System.out, compressor);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
} finally {
    CodecPool.returnCompressor(compressor);
}
}
```

Мы получаем экземпляр `Compressor` из пула для заданного объекта `Compression-Codec` и используем его в перегруженном методе `createOutputStream()` кодека. Блок `finally` гарантирует, что объект `Compressor` будет возвращен в пул даже при возникновении исключения `IOException` в процессе копирования байтов между потоками.

Сжатие и разбиение входных данных

Рассматривая возможность сжатия данных, обрабатываемых MapReduce, важно понимать, поддерживает ли формат сжатия разбиение. Допустим, в HDFS хранится несжатый файл размером 1 Гбайт. При 64-мегабайтном размере блока HDFS файл будет храниться в 16 блоках, а задание MapReduce, использующее этот файл в качестве входных данных, создаст 16 сплитов, каждый из которых обрабатывается независимо как входные данные отдельной задачи отображения.

Теперь представьте, что файл сжат в формате gzip и размер сжатого файла составляет 1 Гбайт. Как и прежде, HDFS хранит файл в 16 блоках. Однако создать сплит для каждого блока не удастся, потому что начать чтение с произвольной точки потока gzip невозможно, а следовательно, задача отображения не сможет прочитать свой сплит независимо от других. Формат gzip использует для хранения сжатых данных формат DEFLATE, в котором данные хранятся в виде серии сжатых блоков. Проблема в том, что начало блока не имеет никаких отличительных признаков, которые позволили бы переместить позицию из произвольной точки потока к началу следующего блока. По этой причине gzip не поддерживает разбиение.

В данном примере MapReduce поступит правильно и не будет пытаться разбивать сжатый файл — известно, что входные данные сжаты в формате gzip (это определяется по расширению файла), а формат gzip не поддерживает разбиение. Задания будут работать, но с потерей локальности: одно отображение будет обрабатывать 16 блоков HDFS, многие из которых не будут локальными для него. Кроме того, с уменьшением количества отображений теряется гранулярность, а следовательно, выполнение задания может занять больше времени.

Если бы файл из нашего гипотетического примера был сжат в формате LZO, возникла бы та же проблема: базовый формат сжатия не предоставляет средств синхронизации чтения с потоком. Однако файлы в формате LZO могут обрабатываться программой индексирования, входящей в поставку LZO-библиотек Hadoop (список сайтов приведен в разделе «Кодеки» на с. 130). Программа строит индекс точек разбиения, вследствие чего появляется возможность разбиения файла при использовании соответствующего входного формата MapReduce.

С другой стороны, в файлах формата bzip2 между блоками размещаются синхронизационные маркеры (48-разрядное приближение числа π), поэтому разбиение для них возможно. (В табл. 4.1 указано, поддерживает ли разбиение тот или иной формат сжатия.)

КАКОЙ ФОРМАТ СЖАТИЯ ИСПОЛЬЗОВАТЬ?

Приложения Hadoop обрабатывают большие наборы данных, поэтому вам стоит по возможности использовать сжатие. Выбор формата зависит от многих факторов: размер файла, формат и программные инструменты, используемые для обработки. Ниже приведены некоторые решения, расположенные по убыванию эффективности.

- Используйте контейнерные форматы — такие, как SequenceFile (с. 186), RCFile (с. 553) и Avro (с. 161). Все они поддерживают как сжатие, так и разбиение. Форматы с быстрым сжатием (LZO, LZ4, Snappy) обычно являются хорошими кандидатами.
- Используйте формат сжатия с поддержкой разбиения (например, bzip2, хотя он работает относительно медленно) или формат, который может индексироваться для поддержки разбиения — например, LZO.
- Разбейте файл на фрагменты в приложении и сжимайте фрагменты по отдельности, используя любые поддерживаемые форматы сжатия (в этом случае неважно, поддерживают они разбиение или нет). В этом случае размер фрагмента выбирается таким образом, чтобы размер сжатых фрагментов был приблизительно равен размеру блока HDFS.

- Храните файлы в несжатом виде.

Для больших файлов не рекомендуется использовать формат сжатия, не поддерживающий разбиение на уровне файла, так как в этом случае теряется локальность данных, а приложения MapReduce становятся крайне неэффективными.

Использование сжатия в MapReduce

Как описано в разделе «Определение кодеков с использованием CompressionCodecFactory» на с. 132, сжатые входные файлы автоматически восстанавливаются при чтении в MapReduce, при этом кодек определяется по расширению файла.

Чтобы применить сжатие к выходным данным задания MapReduce, задайте в конфигурации задания свойству `mapred.output.compress` значение `true`, а свойству `mapred.output.compression.codec` — имя класса кодека сжатия. Также для задания этих свойств можно воспользоваться статическими вспомогательными методами `FileOutputFormat`; пример приведен в листинге 4.4.

Листинг 4.4. Приложение для выполнения задания со сжатием выходных данных

```
public class MaxTemperatureWithCompression {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperatureWithCompression <input path> " +
                "<output path>");
            System.exit(-1);
        }
        Job job = new Job();
        job.setJarByClass(MaxTemperature.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

        FileOutputFormat.setCompressOutput(job, true);
        FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setCombinerClass(MaxTemperatureReducer.class);
```

продолжение ↗

Листинг 4.4 (продолжение)

```

job.setReducerClass(MaxTemperatureReducer.class);

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

```

Программа запускается для сжатых входных данных (формат сжатия которых не обязан совпадать с форматом сжатия вывода, хотя в нашем примере они совпадают) следующим образом:

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output
```

Каждая часть выходных данных подвергается сжатию; в нашем случае часть всего одна:

```

% gunzip -c output/part-r-00000.gz
1949    111
1950    22

```

Если в вывод направляются последовательные файлы (sequence files), вы можете задать свойство `mapred.output.compression.type` для управления типом сжатия. По умолчанию используется значение `RECORD`, при котором сжимаются отдельные записи. Рекомендуется заменить его значением `BLOCK`, при котором сжимаются группы записей, потому что оно обеспечивает более эффективное сжатие (см. «SequenceFile», с. 186).

Также для задания этого свойства можно воспользоваться статическим вспомогательным методом `setOutputCompressionType()` класса `SequenceFileOutputFormat`.

Конфигурационные свойства, определяющие формат сжатия выходных данных заданий MapReduce, описаны в табл. 4.5. Если ваша управляющая программа MapReduce использует интерфейс Tool (см. «GenericOptionsParser, Tool и ToolRunner», с. 211), любые из этих свойств могут передаваться программе в командной строке. Обычно это удобнее, чем изменение программного кода при жестком кодировании свойств сжатия.

Таблица 4.5. Свойства сжатия MapReduce

Имя свойства	Тип	Значение по умолчанию	Описание
<code>mapred.output.compress</code>	<code>boolean</code>	<code>false</code>	Сжатие выходных данных

Имя свойства	Тип	Значение по умолчанию	Описание
mapred.output.compression.codec	Classname	org.apache.hadoop.io.compress.DefaultCodec	Кодек, используемый для сжатия выходных данных
mapred.output.compression.type	String	RECORD	Тип сжатия, используемый для выходных данных SequenceFile: NONE, RECORD или BLOCK

Сжатие выходных данных отображений

Даже если приложение MapReduce читает и записывает несжатые данные, сжатие промежуточного вывода в фазе отображения может принести пользу. Так как выходные данные отображений записываются на диск и передаются по сети узлам свертки, применение быстрого формата сжатия (LZO, LZ4, Snappy) обеспечит прирост производительности просто за счет сокращения объема передаваемых данных. Свойства конфигурации для включения сжатия выходных данных отображений и настройки формата сжатия перечислены в табл. 4.6.

Таблица 4.6. Свойства сжатия выходных данных отображений

Имя свойства	Тип	Значение по умолчанию	Описание
mapred.compress.map.output	boolean	false	Сжатие выходных данных
mapred.map.output.compression.codec	Class	org.apache.hadoop.io.compress.DefaultCodec	Кодек, используемый для сжатия выходных данных отображений

Чтобы включить сжатие выходных данных отображений в формате gzip (с новым API), включите в свое задание следующий фрагмент:

```
Configuration conf = new Configuration();
conf.setBoolean("mapred.compress.map.output", true);
conf.setClass("mapred.map.output.compression.codec", GzipCodec.class,
    CompressionCodec.class);
Job job = new Job(conf);
```

В старом API та же задача решалась вспомогательными методами объекта JobConf:

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

Сериализация

Сериализацией называется процесс преобразования структурированных объектов в поток байтов для передачи по сети или долгосрочного хранения. *Десериализация* – обратный процесс преобразования потока байтов в набор структурированных объектов.

Сериализация встречается в двух достаточно разных областях распределенной обработки данных: в межпроцессных взаимодействиях и в долгосрочном хранении.

В Hadoop межпроцессные взаимодействия между узлами системы реализуются с использованием механизма RPC (Remote Procedure Calls). Протокол RPC использует сериализацию для преобразования сообщения в двоичный поток, который передается удаленному узлу, который десериализует двоичный поток в исходное сообщение. В общем случае формат сериализации RPC должен обладать следующими свойствами:

Компактность

Компактный формат обеспечивает наиболее эффективное использование пропускной способности сети – самого дефицитного ресурса в центрах обработки данных.

Быстрота

Межпроцессные взаимодействия образуют основу работы распределенной системы, поэтому очень важно, чтобы процессы сериализации и десериализации выполнялись с минимальными затратами ресурсов.

Расширяемость

Со временем протоколы изменяются из-за появления новых требований, поэтому их развитие не должно создавать проблем для клиентов и серверов. Например, если к вызову метода добавляется новый аргумент, новые серверы должны принимать сообщения в старом формате (без нового аргумента) от старых клиентов.

Совместимость

В некоторых системах клиенты пишутся на языках, отличных от языка сервера. Формат должен быть спроектирован так, чтобы достичь совместимости.

На первый взгляд требования к формату данных, выбираемому для долгосрочного хранения, отличаются от требований к среде сериализации. В конце концов, жизненный цикл RPC длится меньше секунды, а долгосрочные данные могут быть прочитаны через несколько лет после того, как они были записаны. Но оказывается, четыре свойства формата сериализации RPC также чрезвычайно важны для формата долгосрочного хранения данных. Формат хранения должен быть компактным (для эффективного использования пространства), быстрым (чтобы затраты на чтение и запись терабайтов данных были минимальными), расширяемым (чтобы операции чтения и записи данных могли выполняться на разных языках).

Hadoop использует собственный формат сериализации `Writable` – безусловно компактный и быстрый, но не такой простой в расширении или использовании из языков, отличных от Java. Так как формат `Writable` играет важную роль в Hadoop (большинство программ MapReduce используют его для своих типов ключей и значений), мы рассмотрим его в следующих трех разделах, прежде чем переходить к более подробному рассмотрению сред сериализации и Avro (система сериализации, спроектированная для преодоления некоторых ограничений `Writable`).

Интерфейс `Writable`

Интерфейс `Writable` определяет два метода: для записи своего состояния в двоичный поток `DataOutput` и для чтения состояния из двоичного потока `DataInput`.

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
    void write(DataOutput out) throws IOException;
    void readFields(DataInput in) throws IOException;
}
```

Рассмотрим конкретную реализацию `Writable`, чтобы понять, как с ней работать. Мы будем использовать `IntWritable` – обертку для типа Java `int`. Для создания объекта и задания его значения можно вызвать метод `set()`:

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Также можно воспользоваться конструктором, получающим целое значение:

```
IntWritable writable = new IntWritable(163);
```

Чтобы проанализировать сериализованную форму `IntWritable`, мы написали небольшой вспомогательный метод, который упаковывает `java.io.ByteArrayOutputStream` в `java.io.DataOutputStream` (реализация `java.io.DataOutput`) для сохранения байтов сериализованного потока:

```
public static byte[] serialize(Writable writable) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(out);
    writable.write(dataOut);
    dataOut.close();
    return out.toByteArray();
}
```

Целое число записывается в виде четырех байтов (для проверки используются `assert`-условия JUnit 4):

```
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
```

Байты записываются в обратном порядке (то есть сначала в поток записывается старший байт, как того требует интерфейс `java.io.DataOutput`). Для просмотра шестнадцатеричного представления используется метод Hadoop `StringUtils`:

```
assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

Теперь проверим десериализацию. Как и в предыдущем случае, мы создаем вспомогательный метод для чтения объекта `Writable` из массива байтов:

```
public static byte[] deserialize(Writable writable, byte[] bytes)
    throws IOException {
    ByteArrayInputStream in = new ByteArrayInputStream(bytes);
    DataInputStream dataIn = new DataInputStream(in);
    writable.readFields(dataIn);
    dataIn.close();
    return bytes;
}
```

Мы конструируем новый пустой объект `IntWritable`, вызываем `deserialize()` для чтения только что записанных данных и убеждаемся в том, что метод `get()` возвращает исходное значение 163:

```
IntWritable newWritable = new IntWritable();
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163));
```

WritableComparable и сравнения

`IntWritable` реализует интерфейс `WritableComparable`, который представляет собой субинтерфейс `Writable` и `java.lang.Comparable`:

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable, Comparable<T> { }
```

Сравнение типов играет важнейшую роль в MapReduce, так как в фазе сортировки выполняется сравнение ключей. Одной из оптимизаций, предоставляемых Hadoop, является расширение `RawComparator` интерфейса Java `Comparator`:

```
package org.apache.hadoop.io;

import java.util.Comparator;
public interface RawComparator<T> extends Comparator<T> {

    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

}
```

Интерфейс позволяет реализациям сравнивать записи, прочитанные из потока, без их десериализации в объекты; тем самым он избавляется от затрат на создание объектов. Например, для `IntWritable` метод `compare()` реализуется чтением целого числа из байтовых массивов `b1` и `b2` и их прямым сравнением по заданным начальными позициями (`s1` и `s2`) и длинам (`l1` и `l2`).

`WritableComparator` — реализация общего назначения `RawComparator` для классов `WritableComparable`. Интерфейс предоставляет две основные функции. Во-первых, он предоставляет реализацию по умолчанию низкоуровневого метода `compare()`, которая десериализует сравниваемые объекты из потока и вызывает метод `compare()` объектов. Во-вторых, он действует как фабрика экземпляров `RawComparator` (зарегистрированных реализациями `Writable`).

Например, чтобы получить объект сравнения для `IntWritable`, достаточно использовать следующую конструкцию:

```
RawComparator<IntWritable> comparator =
        WritableComparator.get(IntWritable.class);
```

Полученный объект может использоваться для сравнения двух объектов `IntWritable`:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

или их сериализованных представлений:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
greaterThan(0));
```

Классы Writable

Hadoop поставляется с большой подборкой классов `Writable` в пакете `org.apache.hadoop.io.package`. Классы образуют иерархию, изображенную на рис. 4.1.

Обертки для примитивов Java

Обертки `Writable` имеются для всех примитивных типов Java (табл. 4.7), кроме типа `char` (который может храниться в `IntWritable`). Все они содержат методы `get()` и `set()` для чтения и сохранения упакованного значения.

Таблица 4.7. Классы-обертки `Writable` для примитивов Java

Примитив Java	Реализация <code>Writable</code>	Размер сериализации (в байтах)
<code>boolean</code>	<code>BooleanWritable</code>	1
<code>byte</code>	<code>ByteWritable</code>	1
<code>short</code>	<code>ShortWritable</code>	2
<code>int</code>	<code>IntWritable</code>	4
	<code>VIntWritable</code>	1–5
<code>float</code>	<code>FloatWritable</code>	4
<code>long</code>	<code>LongWritable</code>	8
	<code>VLongWritable</code>	1–9
<code>double</code>	<code>DoubleWritable</code>	8

При кодировании целых чисел появляется выбор между форматами фиксированной длины (`IntWritable` и `LongWritable`) и форматами переменной длины (`VIntWritable` и `VLongWritable`). Форматы переменной длины используют всего один байт для кодирования значения, если оно достаточно мало (от –112 до 127 включительно); в противном случае первый байт определяет знак числа и количество последующих байтов. Например, для представления числа 163 используются два байта:

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

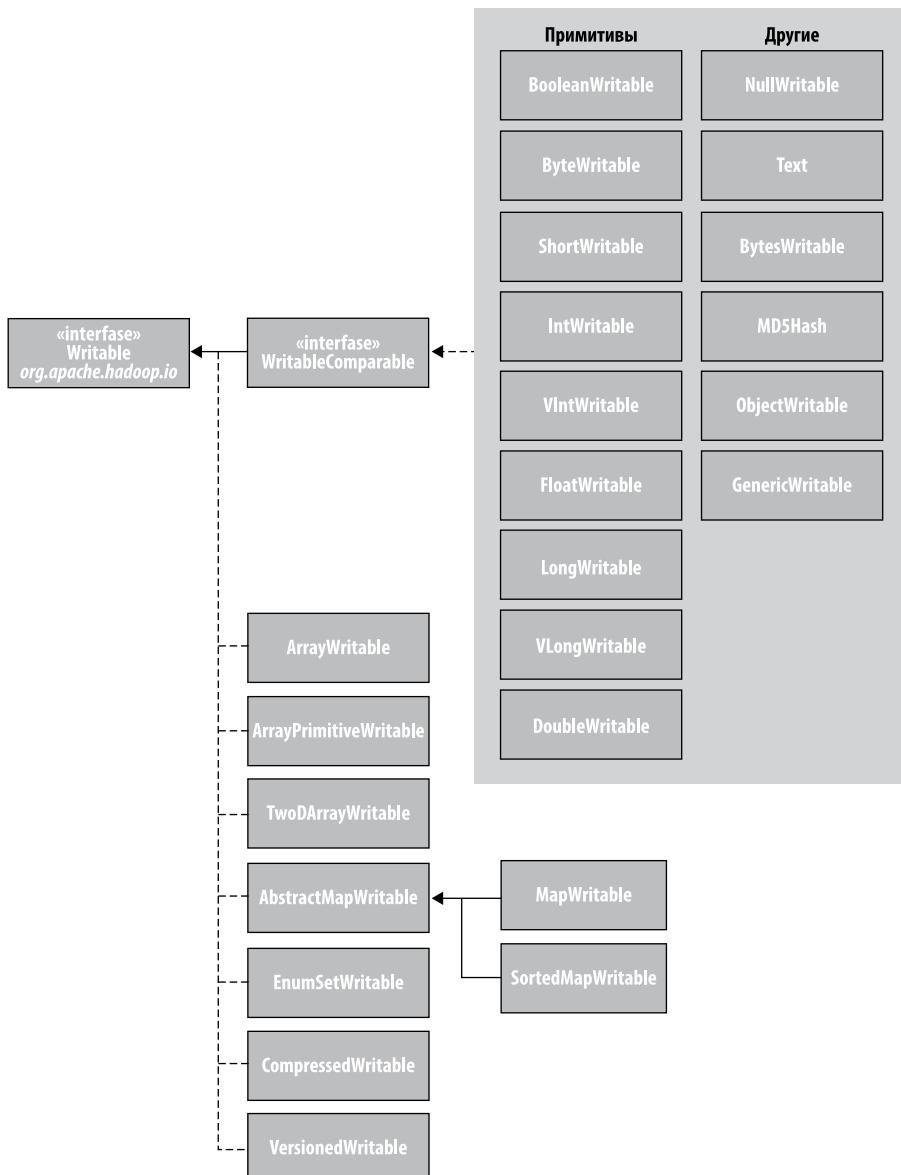


Рис. 4.1. Иерархия классов `Writable`

Как выбрать между фиксированной и переменной длиной кодирования? Фиксированная длина хорошо подходит при относительно равномерном распределении значений по пространству значений — например, при использовании (хорошо спроектированной) хеш-функции. Чаще числовые переменные распределяются

неравномерно, и статистически переменная длина кодирования сэкономит место. У кодирования переменной длины есть и другое преимущество: простота переключения с `VIntWritable` на `VLongWritable`, потому что их кодировки фактически совпадают. Таким образом, выбор представления с переменной длиной оставляет возможности для расширения; ведь вам не приходится изначально связывать себя 8-разрядным представлением.

Text

`Text` — `Writable`-обертка для последовательностей символов UTF-8; считайте, что это `Writable`-эквивалент `java.lang.String`. Класс `Text` является заменой для класса `UTF8`, который использовать не рекомендуется, потому что он не поддерживает строки, кодировка которых занимает более 32 767 байт, и использует модифицированную версию UTF-8 языка Java.

Класс `Text` хранит количество байтов в строке в формате `int` (с кодированием переменной длины), поэтому максимальное значение равно 2 Гбайт. Кроме того, `Text` использует стандартную кодировку UTF-8, что упрощает потенциальные взаимодействия с другими программами, поддерживающими UTF-8.

Индексирование

Из-за ориентированности `Text` на стандартный вариант UTF-8 между ним и классом Java `String` существуют некоторые различия. Индексирование класса `Text` осуществляется по позиции в кодированной последовательности байтов, а не по символам Юникода в строке или кодовым блокам Java `char` (как в случае с `String`). Для ASCII-строк эти три концепции позиции индексирования совпадают. Следующий пример демонстрирует использование метода `charAt()`:

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));

assertThat(t.charAt(2), is((int) 'd'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

Обратите внимание: `charAt()` возвращает значение `int`, представляющее кодовый пункт Юникода, — в отличие от версии для `String`, возвращающей `char`. Класс `Text` также содержит метод `find()`, аналогичный методу `indexOf()` класса `String`:

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

Юникод

Различия между `Text` и `String` наглядно проявляются при работе с символами, кодировка которых занимает более одного байта. Возьмем символы Юникода, перечисленные в табл. 4.8¹.

Таблица 4.8. Символы Юникода

Кодовый пункт	U+0041	U+00DF	U+6771	U+10400
Имя	LATIN CAPITAL LETTER A	LATIN SMALL LETTER SHARP S	– (иероглиф)	DESERET CAPITAL LETTER LONG I
Кодовые блоки UTF-8	41	c3 9f	e6 9d b1	f0 90 90 80
Представление в Java	\u0041	\u00DF	\u6771	\u00D801\uD0C00

Все символы в таблице, кроме последнего (U+10400), могут быть выражены одним значением Java `char`. U+10400 — дополнительный символ, представляемый двумя символами Java (называемыми *суррогатной парой*). Тесты в листинге 4.5 демонстрируют различия между `String` и `Text` при обработке строки из четырех символов из табл. 4.8.

Листинг 4.5. Тесты, демонстрирующие различия между классами String и Text

```
public class StringTextComparisonTest {
    @Test
    public void string() throws UnsupportedEncodingException {

        String s = "\u0041\u00DF\u6771\u00D801\u00D0C00";
        assertThat(s.length(), is(5));
        assertThat(s.getBytes("UTF-8").length, is(10));

        assertThat(s.indexOf("\u0041"), is(0));
        assertThat(s.indexOf("\u00DF"), is(1));
        assertThat(s.indexOf("\u6771"), is(2));
        assertThat(s.indexOf("\u00D801\u00D0C00"), is(3));

        assertThat(s.charAt(0), is('\u0041'));
    }
}
```

продолжение ↗

¹ Пример позаимствован из статьи «Supplementary Characters in the Java Platform».

Листинг 4.5 (продолжение)

```
assertThat(s.charAt(1), is('\u00DF'));
assertThat(s.charAt(2), is('\u6771'));
assertThat(s.charAt(3), is('\uD801'));
assertThat(s.charAt(4), is('\uDC00'));

assertThat(s.codePointAt(0), is(0x0041));
assertThat(s.codePointAt(1), is(0x00DF));
assertThat(s.codePointAt(2), is(0x6771));
assertThat(s.codePointAt(3), is(0x10400));
}

@Test
public void test() {

    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
    assertThat(t.getLength(), is(10));

    assertThat(t.find("\u0041"), is(0));
    assertThat(t.find("\u00DF"), is(1));
    assertThat(t.find("\u6771"), is(3));
    assertThat(t.find("\uD801\uDC00"), is(6));

    assertThat(t.charAt(0), is(0x0041));
    assertThat(t.charAt(1), is(0x00DF));
    assertThat(t.charAt(3), is(0x6771));
    assertThat(t.charAt(6), is(0x10400));
}
}
```

Тест подтверждает, что длина объекта `String` равна количеству содержащихся в нем кодовых блоков `char` (5 — по одному для каждого из первых трех символов и суррогатная пара в конце), тогда как длина объекта `Text` равна количеству байтов в его кодированном представлении UTF-8 ($10 = 1+2+3+4$). Аналогичным образом метод `indexOf()` класса `String` возвращает индекс в кодовых блоках `char`, а метод `find()` класса `Text` возвращает смещение в байтах.

Метод `charAt()` класса `String` возвращает кодовый блок `char` для заданного индекса, который в случае суррогатной пары не соответствует полному символу Юникода. Для получения одного символа Юникода, представленного в формате `int`, необходим метод `codePointAt()`, индексируемый по кодовым блокам `char`. Пожалуй, метод `charAt()` класса `Text` похож на метод `codePointAt()` больше, чем его «тезка» из `String`. Единственное различие — индексирование по смещению в байтах.

Перебор

Использование байтового смещения для индексирования усложняет перебор символов Юникода в `Text`, так как вы не можете просто увеличить индекс на 1. Идиома перебора выглядит довольно запутанно (листинг 4.6): объект `Text` преобразуется в `java.nio.ByteBuffer`, после чего статический метод `bytesToCodePoint()` класса `Text` последовательно вызывается с полученным буфером. Метод получает следующий кодовый пункт в формате `int` и обновляет позицию в буфере. Конец строки определяется по возвращению вызовом `bytesToCodePoint()` значения `-1`.

Листинг 4.6. Перебор символов в объекте `Text`

```
public class TextIterator {  
  
    public static void main(String[] args) {  
        Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");  
  
        ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());  
        int cp;  
        while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) {  
            System.out.println(Integer.toHexString(cp));  
        }  
    }  
}
```

Программа выводит кодовые пункты четырех символов строки:

```
% hadoop TextIterator  
41  
df  
6771  
10400
```

Изменяемость

Между `String` и `Text` существует еще одно различие: объект `Text` может изменяться (как и все реализации `Writable` в Hadoop, кроме объекта `NullWritable`, существующего только в одном экземпляре). Чтобы повторно использовать экземпляр `Text`, вызовите для него один из методов `set()`. Пример:

```
Text t = new Text("hadoop");  
t.set("pig");  
assertThat(t.getLength(), is(3));  
assertThat(t.getBytes().length, is(3));
```



В некоторых ситуациях длина массива байтов, возвращаемого методом `getBytes()`, может превышать длину, возвращаемую `getLength()`:

```
Text t = new Text("hadoop");
t.set(new Text("pig"));
assertThat(t.getLength(), is(3));
assertThat("Byte length not shortened", t.getBytes().length,
           is(6));
```

Этот пример показывает, почему при вызове `getBytes()` всегда следует вызывать `getLength()`, чтобы вы всегда знали, какую часть массива занимают действительные данные.

Преобразование в String

Класс `Text` не имеет такого богатого API для обработки строк, как `java.lang.String`, так что во многих случаях объект `Text` приходится преобразовывать в `String`. Преобразование осуществляется стандартным способом — с использованием метода `toString()`:

```
assertThat(new Text("hadoop").toString(), is("hadoop"));
```

BytesWritable

`BytesWritable` — обертка для массива двоичных данных. Сериализованный формат состоит из целочисленного поля (4 байта), определяющего количество байтов данных, за которым следуют сами байты. Например, массив байтов длины 2 со значениями 3 и 5 сериализуется в 4-байтовое целое число (00000002), за которым следуют два байта из массива (03 и 05):

```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

`BytesWritable` — изменяемый объект; его значение изменяется вызовом метода `set()`. Как и в случае с `Text`, размер массива байтов, возвращаемого методом `getBytes()` объекта `BytesWritable` может не соответствовать фактическому размеру данных, хранимых в `BytesWritable`. Размер `BytesWritable` определяется вызовом `getLength()`, например:

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

NullWritable

`NullWritable` — особая разновидность `Writable` с нулевой длиной сериализации. Байты не записываются в поток и не читаются из потока. Объект используется как заполнитель; например, в MapReduce ключ или значение можно объявить с типом `NullWritable`, если вы не собираетесь использовать его значение. Фактически тем самым вы сохраняете пустое значение-константу. `NullWritable` также может использоваться как ключ `SequenceFile` при сохранении списка значений (вместо пар «ключ-значение»). Объект не может изменяться и существует в единственном экземпляре, который может быть получен вызовом `NullWritable.get()`.

ObjectWritable и GenericWritable

`ObjectWritable` — обертка общего назначения для примитивов Java, `String`, перечислений, `Writable`, `null` или массивов любых из перечисленных типов. Тип используется в Hadoop RPC для передачи аргументов методов и возвращаемых типов.

Обертка `ObjectWritable` удобна в тех случаях, когда поле может иметь более одного типа. Например, если значение в `SequenceFile` может содержать разные типы, вы объявляете тип значения `ObjectWritable` и упаковываете каждый тип в `ObjectWritable`. За такую универсальность приходится расплачиваться размерами, потому что при каждой сериализации записывается имя упакованного класса. Если количество типов невелико и известно заранее, для экономии места можно создать статический массив типов и использовать индекс массива в качестве сериализованной ссылки на тип. Именно такой подход используется классом `GenericWritable`; чтобы указать поддерживаемые типы, субклассируйте его.

Коллекции Writable

В пакете `org.apache.hadoop.io` определяются шесть типов коллекций `Writable`: `ArrayWritable`, `ArrayPrimitiveWritable`, `TwoDArrayWritable`, `MapWritable`, `SortedMapWritable` и `EnumSetWritable`.

`ArrayWritable` и `TwoDArrayWritable` — реализации `Writable` для массивов и двумерных массивов экземпляров `Writable`. Все элементы `ArrayWritable` или `TwoDArrayWritable` должны быть экземплярами одного класса, который задается при конструировании следующим образом:

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

В контекстах, где `Writable` определяется типом (например, в ключах, или значениях `SequenceFile`, или входных данных MapReduce вообще), необходимо

субклассировать `ArrayWritable` (или `TwoDArrayWritable`) для статического задания типа. Например:

```
public class TextArrayWritable extends ArrayWritable {  
    public TextArrayWritable() {  
        super(Text.class);  
    }  
}
```

Оба класса — `ArrayWritable` и `TwoDArrayWritable` — содержат методы `get()` и `set()`, а также метод `toArray()`, который создает поверхностную копию массива (или двумерного массива).

`ArrayPrimitiveWritable` — обертка для массивов примитивов Java. Тип компонента определяется при вызове `set()`, применять субклассирование для задания типа не нужно.

`MapWritable` и `SortedMapWritable` — реализации `java.util.Map<Writable, Writable>` и `java.util.SortedMap<WritableComparable, Writable>` соответственно. Тип каждого поля ключа и значения является частью формата сериализации этого поля. Тип хранится в виде одного байта, который используется как индекс в массиве типов. Массив заполняется стандартными типами в пакете `org.apache.hadoop.io`, но нестандартные типы `Writable` тоже поддерживаются — для этого записывается заголовок, кодирующий массив типов для нестандартных типов. В своей реализации `MapWritable` и `SortedMapWritable` используются для нестандартных типов положительные значения, поэтому в любом конкретном экземпляре `MapWritable` или `SortedMapWritable` могут использоваться до 127 разных нестандартных классов `Writable`. Пример использования `MapWritable` с разными типами ключей и значений:

```
MapWritable src = new MapWritable();  
src.put(new IntWritable(1), new Text("cat"));  
src.put(new VIntWritable(2), new LongWritable(163));  
  
MapWritable dest = new MapWritable();  
WritableUtils.cloneInto(dest, src);  
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));  
assertThat((LongWritable) dest.get(new VIntWritable(2)), is(new  
    LongWritable(163)));
```

Бросается в глаза отсутствие реализаций коллекций `Writable` для множеств и списков. Обобщенное множество может эмулироваться использованием `MapWritable` (или `SortedMapWritable` для отсортированного множества) со значениями `NullWritable`. Также имеется тип `EnumSetWritable` для множеств перечисляемых типов. Для списков одного типа `Writable` подойдет `ArrayWritable`, но для хранения разных типов `Writable` в одном списке можно воспользоваться `GenericWritable`.

для упаковки элементов в `ArrayWritable`. Также можно написать обобщенную реализацию `ListWritable`, используя идеи из `MapWritable`.

Пользовательские реализации `Writable`

Hadoop содержит немало полезных реализаций `Writable`, удовлетворяющих большинство потребностей; однако время от времени возникает необходимость в написании вашей собственной, специализированной реализации. Пользовательская реализация `Writable` позволяет полностью контролировать двоичное представление и порядок сортировки. Так как интерфейс `Writable` занимает центральное место в тракте данных MapReduce, оптимизация двоичного представления может существенно повлиять на производительность. Готовые реализации `Writable`, входящие в поставку Hadoop, хорошо оптимизированы, но для более сложных структур часто бывает предпочтительнее создать новый тип `Writable`, чем заниматься объединением готовых типов.

Чтобы продемонстрировать процесс создания пользовательских реализаций `Writable`, мы напишем реализацию `TextPair` для представления пары строк. Основа реализации приведена в листинге 4.7.

Листинг 4.7. Реализация `Writable` для сохранения пары объектов `Text`

```
import java.io.*;
import org.apache.hadoop.io.*;
public class TextPair implements WritableComparable<TextPair> {
    private Text first;
    private Text second;

    public TextPair() {
        set(new Text(), new Text());
    }

    public TextPair(String first, String second) {
        set(new Text(first), new Text(second));
    }

    public TextPair(Text first, Text second) {
        set(first, second);
    }

    public void set(Text first, Text second) {
        this.first = first;
```

продолжение ↗

Листинг 4.7 (продолжение)

```
    this.second = second;
}

public Text getFirst() {
    return first;
}

public Text getSecond() {
    return second;
}

@Override
public void write(DataOutput out) throws IOException {
    first.write(out);
    second.write(out);
}

@Override
public void readFields(DataInput in) throws IOException {
    first.readFields(in);
    second.readFields(in);
}

@Override
public int hashCode() {
    return first.hashCode() * 163 + second.hashCode();
}

@Override
public boolean equals(Object o) {
    if (o instanceof TextPair) {
        TextPair tp = (TextPair) o;
        return first.equals(tp.first) && second.equals(tp.second);
    }
    return false;
}

@Override
public String toString() {
    return first + "\t" + second;
}

@Override
public int compareTo(TextPair tp) {
    int cmp = first.compareTo(tp.first);
    if (cmp != 0) {
```

```
        return cmp;
    }
    return second.compareTo(tp.second);
}
}
```

Первая часть реализации вполне прямолинейна: в ней определяются две переменные экземпляров типа `Text` с именами `first` и `second`, а также соответствующие конструкторы, `get`- и `set`-методы. Все реализации `Writable` должны иметь конструктор по умолчанию, чтобы среда MapReduce могла создать их экземпляры, а затем заполнить их поля вызовом `readFields()`. Экземпляры `Writable` часто используются повторно, поэтому постарайтесь избежать создания объектов в методе `write()` или `readFields()`.

Метод `write()` класса `TextPair` последовательно сериализует каждый объект `Text` в выходной поток, делегируя выполнение операции самим объектам `Text`. Аналогичным образом `readFields()` десериализует байты из входного потока, делегируя выполнение операции каждому объекту `Text`. Интерфейсы `DataOutput` и `DataInput` содержат богатый набор методов для сериализации и десериализации примитивов Java, так что в общем случае вы полностью контролируете формат объекта `Writable`.

Для каждого объекта-значения, который вы пишете на Java, следует переопределить методы `hashCode()`, `equals()` и `toString()` класса `java.lang.Object`. Метод `hashCode()` использует `HashPartitioner` (реализация разделения по умолчанию в MapReduce) для выбора раздела свертки; хеш-функция должна обеспечивать приблизительно равные размеры разделов.



Если вы когда-либо захотите написать собственную реализацию `Writable` с `TextOutputFormat`, то вы должны реализовать метод `toString()`. `TextOutputFormat` вызывает `toString()` для ключей и значений, чтобы получить их выходное представление. Для `TextPair` мы записываем внутренние объекты `Text` в виде строк, разделенных символом табуляции.

Класс `TextPair` реализует `WritableComparable`, поэтому он предоставляет ожидаемую реализацию метода `compareTo()`: сортировка осуществляется сначала по первой строке, а затем по второй. Обратите внимание на отличие `TextPair` от `TextArrayWritable` из предыдущего раздела (помимо количества сохраняемых объектов `Text`): `TextArrayWritable` реализует только `Writable`, а не `WritableComparable`.

Реализация `RawComparator`

Код `TextPair` работает в том виде, в котором он приведен в листинге 4.7; однако есть еще одна оптимизация. Как объясняется в разделе «`WritableComparable`

и сравнения» на с. 143, при использовании `TextPair` в качестве ключа MapReduce, для вызова метода `compareTo()` необходимо выполнить десериализацию. А нельзя ли сравнить два объекта `TextPair` просто по их сериализованным представлениям?

Оказывается, можно, потому что `TextPair` является конкатенацией двух объектов `Text`, а двоичное представление объекта `Text` состоит из количества байтов в представлении строки в UTF-8, за которым следуют сами байты UTF-8. Прочитав исходную длину, мы узнаем длину представления первого объекта `Text` в байтах; после этого можно воспользоваться интерфейсом `RawComparator` объекта `Text` и вызвать его с соответствующими смещениями для первой или второй строки. Подробности приведены в листинге 4.8 (этот код включается в класс `TextPair`).

Листинг 4.8. Реализация `RawComparator` для сравнения байтовых представлений `TextPair`

```
public static class Comparator extends WritableComparator {  
  
    private static final Text.Comparator TEXT_COMPARATOR =  
        new Text.Comparator();  
  
    public Comparator() {  
        super(TextPair.class);  
    }  
    @Override  
    public int compare(byte[] b1, int s1, int l1,  
                       byte[] b2, int s2, int l2) {  
  
        try {  
            int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);  
            int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);  
            int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);  
            if (cmp != 0) {  
                return cmp;  
            }  
            return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,  
                                           b2, s2 + firstL2, l2 - firstL2);  
        } catch (IOException e) {  
            throw new IllegalArgumentException(e);  
        }  
    }  
  
    static {  
        WritableComparator.define(TextPair.class, new Comparator());  
    }  
}
```

Мы используем субклассирование `WritableComparator` вместо прямой реализации `RawComparator`, чтобы получить доступ к вспомогательным методам и реализациям по умолчанию. Неочевидной частью кода является вычисление `firstL1` и `firstL2` — длины первого поля `Text` в каждом из байтовых потоков. Каждое значение складывается из размера целого поля переменной длины (возвращаемого вызовом `decodeVIntSize()` для `WritableUtils`) и кодируемого значения (возвращаемого `readVInt()`).

Статический блок регистрирует низкоуровневый объект сравнения, чтобы он использовался по умолчанию каждый раз, когда MapReduce встречает класс `TextPair`.

Пользовательские реализации сравнения

Как мы видим в примере с `TextPair`, написание низкоуровневых реализаций сравнения требует определенной осторожности, так как вам придется работать с информацией на уровне байтов. Если вам понадобится написать собственную реализацию, стоит ознакомиться с примерами реализаций `Writable` из пакета `org.apache.hadoop.io`. Кроме того, вспомогательные методы из `WritableUtils` очень удобны.

Пользовательские реализации сравнения также следует по возможности писать для `RawComparator`. Они реализуют порядок сортировки, отличный от естественного порядка, определяемого по умолчанию. В листинге 4.9 приведен `FirstComparator` — класс сравнения для `TextPair`, который учитывает только первую строку пары. Обратите внимание: мы переопределили метод `compare()`, который получает объекты, так что оба метода `compare()` обладают одинаковой семантикой.

Мы вернемся к этому классу сравнения в главе 8, когда будем рассматривать соединения и вторичную сортировку в MapReduce (см. «Соединения», с. 368).

Листинг 4.9. Пользовательская реализация `RawComparator` для сравнения первого поля байтовых представлений `TextPair`

```
public static class FirstComparator extends WritableComparator {  
  
    private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();  
  
    public FirstComparator() {  
        super(TextPair.class);  
    }  
    @Override  
    public int compare(byte[] b1, int s1, int l1,  
                      byte[] b2, int s2, int l2) {  
  
        try {  
            продолжение ↗  
        }  
    }  
}
```

Листинг 4.9 (продолжение)

```
int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
} catch (IOException e) {
    throw new IllegalArgumentException(e);
}
}

@Override
public int compare(WritableComparable a, WritableComparable b) {
    if (a instanceof TextPair && b instanceof TextPair) {
        return ((TextPair) a).first.compareTo(((TextPair) b).first);
    }
    return super.compare(a, b);
}
}
```

Программные среды сериализации

Большинство программ MapReduce использует типы ключей и значений `Writable`, но это не является обязательным требованием MapReduce API. Собственно, использовать можно любой тип; необходимо лишь наличие механизма преобразования каждого типа в двоичное представление и обратно.

Для обеспечения этого требования в Hadoop предусмотрен API для подключения программных сред сериализации. Программные среды представлены реализацией интерфейса `Serialization` (в пакете `org.apache.hadoop.io.serializer`). `WritableSerialization`, например, является реализацией `Serialization` для типов `Writable`.

Интерфейс `Serialization` определяет соответствие между типами и экземплярами `Serializer` (для преобразования объекта в поток байтов) и экземплярами `Deserializer` (для преобразования потока байтов в объект).

Чтобы зарегистрировать реализации `Serialization`, задайте свойству `io.serializations` разделенный запятыми список имен классов. Значение по умолчанию включает в себя `org.apache.hadoop.io.serializer.WritableSerialization`, а также специализации для Avro и рефлексивной сериализации. Это означает, что в исходном состоянии сериализация/десериализация возможна только для объектов `Writable` или `Avro`.

Hadoop включает в себя класс `JavaSerialization`, использующий механизм Java Object Serialization. Хотя этот механизм упрощает использование стандартных

типов Java (таких, как `Integer` и `String`) в программах MapReduce, по эффективности он уступает `Writable`, поэтому идти на этот компромисс не рекомендуется.

ПОЧЕМУ НЕ СЛЕДУЕТ ИСПОЛЬЗОВАТЬ JAVA OBJECT SERIALIZATION?

В Java существует собственный механизм сериализации, называемый Java Object Serialization, плотно интегрированный с языком. Естественно спросить, почему он не был использован в Hadoop? Вот как Дуг Каттинг ответил на этот вопрос:

«...Почему я не использовал механизм `Serialization`, когда мы начали работать над Hadoop? Потому что он выглядел громоздким и неуклюжим, а нам нужно было что-то быстрое и компактное, обеспечивающее полный контроль над чтением и записью объектов — все это очень важно для Hadoop. `Serialization` предоставляет некоторую степень контроля, но за нее нужно сражаться.»

Отказ от RMI объяснялся теми же причинами. Эффективные, высокопроизводительные межпроцессные взаимодействия очень важны для Hadoop. Я считал, что нам понадобится полный контроль за подключениями, тайм-аутами и буферами, а RMI такой возможности не предоставляет.»

Проблема в том, что Java Object Serializtion не соответствует критериям формата сериализации, приведенным выше: компактность, быстрота, расширяемость и совместимость.

Механизм Java Object Serialization не компактен. Классы, реализующие `java.io.Serializable` или `java.io.Externalizable`, записывают свое имя класса и представление объекта в поток. Последующие экземпляры того же класса записывают ссылку на первый экземпляр, занимающую только 5 байтов. Однако такие ссылки плохо сочетаются с произвольным доступом, потому что класс, к которому относится ссылка, может находиться в любой точке предшествующего потока. Что еще хуже, ссылки вмешиваются в процедуру сортировки записей сериализованного потока, потому что первая запись класса должна рассматриваться по особым правилам.

Таким образом, запись имени класса в поток создает слишком много проблем. `Writable` идет по другому пути: предполагается, что клиент знает нужный тип. В результате формат получается существенно более компактным по сравнению с Java Object Serialization, а произвольный доступ и сортировка работают нормально, потому что каждая запись существует независимо от других.

Java Object Serialization — механизм общего назначения для сериализации графов объектов, поэтому он неизбежно требует затрат на сериализацию и десериализацию объектов. Кроме того, процедура десериализации создает новый экземпляр для каждого объекта, десериализованного из потока. С другой стороны, объекты Writable могут использоваться (и часто используются) повторно. Для задания MapReduce, которое сериализует и десериализует миллиарды записей всего нескольких разных типов, отказ от создания новых объектов приводит к значительной экономии.

Что касается расширяемости, Java Object Serialization в определенной степени поддерживает эволюцию типов, но этот механизм слишком непрочен и его трудно использовать эффективно (у Writable такой поддержки нет; программисту приходится делать все самостоятельно).

Вообще говоря, другие языки могут интерпретировать протокол потока Java Object Serialization (определенный в спецификации), но на практике распространенных реализаций для других языков не существует, поэтому данное решение ориентировано только на Java. То же самое можно сказать и о Writable.

IDL

Существуют и другие среды сериализации, которые используют другой подход: вместо определения типов в коде используется декларативное, независящее от языка определение IDL (Interface Description Language). В этом случае система генерирует типы для разных языков, что хорошо по соображениям совместимости. Также при этом обычно определяются схемы контроля версий, упрощающие эволюцию типов.

Механизм Hadoop Record I/O (пакет `org.apache.hadoop.record`) поддерживает определения IDL, компилируемые в объекты `Writable`, что упрощает генерирование типов, совместимых с MapReduce. Однако по каким-то причинам механизм Hadoop Record I/O не получил широкого использования, и сейчас вместо него обычно используется Avro.

Apache Thrift и Google Protocol Buffers — популярные среды сериализации, широко используемые в качестве формата для долгосрочного сохранения двоичных данных. Имеется ограниченная поддержка для их использования в качестве форматов MapReduce¹, однако используются они во внутренней реализации компонентов Hadoop для RPC и обмена данными.

¹ Проект Twitter Elephant Bird (<http://github.com/kevinweil/elephant-bird>) включает инструменты для работы с Thrift и Protocol Buffers в Hadoop.

В следующем разделе рассматривается Avro — среда сериализации на базе IDL, спроектированная для работы в условиях крупномасштабной обработки данных в Hadoop.

Avro

Apache Avro — система сериализации данных, не зависящая от языка программирования¹. Проект был создан Дугом Каттингом (создатель Hadoop) для преодоления основного недостатка Hadoop *Writable*: отсутствия портируемости. Наличие формата данных, который может обрабатываться на многих языках (в настоящее время C, C++, C#, Java, PHP, Python и Ruby), упрощает совместное использование данных более широкой аудиторией, чем в случае привязки к конкретному языку. Кроме того, технология не потеряет актуальности со временем, так как данные могут пережить язык, используемый для их чтения и записи.

Но почему нужно изобретать новую систему сериализации данных? Avro обладает функциональными возможностями, которые в совокупности отличают ее от других систем — таких, как Apache Thrift или Google Protocol Buffers². Как и в этих, а также многих других системах, данные Avro описываются в виде схемы, независимой от языка. Однако в отличие от других систем, генерирование кода в Avro не является обязательным; это означает, что вы можете читать и записывать данные, соответствующие заданной схеме, даже если ранее ваш код еще не «видел» эту схему. Для этого Avro предполагает, что схема присутствует всегда — как во время чтения, так и во время записи, что обеспечивает чрезвычайно компактное кодирование, так как закодированные значения не нужно помечать идентификаторами полей.

Схемы Avro обычно записываются в JSON, а данные обычно кодируются в двоичном формате, но существуют и другие варианты. Схемы могут записываться на С-подобном языке высокого уровня Avro IDL. Также имеется кодировщик данных на базе JSON, удобный для создания прототипов и отладки данных Avro.

Спецификация Avro точно определяет двоичный формат, который должен поддерживаться всеми реализациями. В нем также указаны многие функциональные возможности Avro, которые должны поддерживаться реализациями. Однако при этом реализации обладают полной свободой в отношении API, предоставляемых

¹ Название позаимствовано у самолетостроительной фирмы из Великобритании, существовавшей в XX веке.

² Кроме того, как показывают эталонные тесты по адресу <http://code.google.com/p/thrift-protobuf-compare/>, Avro также превосходит другие библиотеки сериализации по производительности.

для работы с данными Avro, потому что API неизбежно привязаны к языковой специфике. Факт существования только одного двоичного формата очень важен, потому что он упрощает создание реализации для новых языков и обходит проблему комбинаторного роста численности языков и форматов, которая отрицательно бы отразилась на совместимости.

Avro также обладает широкими возможностями по *преобразованию схем*. В некоторых, тщательно определенных ограничениях схема, использованная для чтения данных, может отличаться от схемы, использованной для их записи. Именно этот механизм лежит в основе поддержки эволюции схем в Avro.

Например, в запись можно добавить новое необязательное поле, объявив его в схеме чтения старых данных. И новые, и старые клиенты смогут читать старые данные, а новые клиенты смогут записывать новые данные с использованием нового поля. И наоборот, если старый клиент видит данные в новой кодировке, он корректно игнорирует старое поле и продолжает обработку так, как если бы он имел дело со старыми данными.

Avro определяет контейнерный формат для последовательностей объектов — по аналогии с последовательными файлами Hadoop. Файл данных Avro содержит раздел метаданных, в котором хранится схема. Файлы данных Avro поддерживают сжатие и разбиение, что очень важно для формата входных данных MapReduce. Более того, поскольку технология Avro проектировалась с расчетом на MapReduce, в будущем появится возможность использования Avro для внедрения передовых MapReduce API (то есть API, обладающих более широкими возможностями, чем Streaming — например, Java API или C++ Pipes) в языки, поддерживающие Avro.

Avro также может использоваться для реализации RPC, но эта возможность здесь не рассматривается. За дополнительной информацией обращайтесь к спецификации.

Типы данных и схемы Avro

Avro определяет небольшой набор примитивных типов данных, которые используются для построения схемных описаний структур данных конкретного приложения. Для обеспечения совместимости реализации должны поддерживать все типы Avro.

Примитивные типы Avro перечислены в табл. 4.9. Каждый примитивный тип также может определяться в расширенной форме с использованием атрибута `type`, например:

```
{ "type": "null" }
```

Таблица 4.9. Примитивные типы Avro

Тип	Описание	Схема
null	Значение отсутствует	"null"
boolean	Двоичное значение	"boolean"
int	32-разрядное целое со знаком	"int"
long	64-разрядное целое со знаком	"long"
float	Вещественное число одинарной точности (32-разрядное) в формате IEEE 754	"float"
double	Вещественное число двойной точности (64-разрядное) в формате IEEE 754	"double"
bytes	Последовательность 8-разрядных байтов без знака	"bytes"
string	Последовательность символов Юникода	"string"

Avro также определяет составные типы, перечисленные в табл. 4.10 (с характерными примерами схем каждого типа).

Таблица 4.10. Составные типы Avro

Тип	Описание	Пример схемы
array	Упорядоченный набор объектов. Все объекты массива должны иметь одинаковую схему	{ "type": "array", "items": "long" }
map	Неупорядоченный набор пар «ключ-значение». Ключи — строки, значения имеют произвольный тип (одинаковый для всех значений набора)	{ "type": "map", "values": "string" }
record	Набор именованных полей произвольных типов	{ "type": "record", "name": "WeatherRecord", "doc": "A weather reading.", "fields": [{"name": "year", "type": "int"}, {"name": "temperature", "type": "int"}, {"name": "stationId", "type": "string"}] }

продолжение ↗

Таблица 4.10 (продолжение)

Тип	Описание	Пример схемы
enum	Набор именованных значений	{ "type": "enum", "name": "Cutlery", "doc": "An eating utensil.", "symbols": ["KNIFE", "FORK", "SPOON"] }
fixed	Фиксированное количество 8-разрядных байтов без знака	{ "type": "fixed", "name": "Md5Hash", "size": 16 }
union	Объединение схем. Представляется массивом JSON, в котором каждый элемент соответствует схеме. Данные, представляемые объединением, должны соответствовать одной из схем	["null", "string", {"type": "map", "values": "string"}]

В каждом языковом Avro API существует представление для каждого типа Avro, специфическое для конкретного языка. Например, типу Avro `double` в C, C++ и Java соответствует тип `double`, в Python — `float`, в Ruby — `Float`.

Более того, в одном языке таких представлений (или *привязок*) может быть несколько. Все языки поддерживают динамическую привязку, которая может использоваться даже в том случае, если схема неизвестна до момента выполнения. В Java такие привязки называются *обобщенными* (*generic*).

При этом реализации Java и C++ могут генерировать код для представления данных в схемах Avro. Генерирование кода, которое называется *конкретной привязкой* в языке Java, представляет собой оптимизацию, которая может быть полезной, если копия схемы доступна до момента чтения или записи данных. Сгенерированные классы также предоставляют API для пользовательского кода, в большей степени ориентированный на предметную область, чем при обобщенной привязке.

В Java также существует третья разновидность привязок — рефлексивные привязки, связывающие типы Avro с существующими типами Java посредством рефлексии (*reflection*). По скорости этот механизм уступает обобщенным и конкретным привязкам, и использовать его в новых приложениях обычно не рекомендуется.

Привязки типов для языка Java перечислены в табл. 4.11. Как видно из таблицы, конкретные привязки обычно совпадают с обобщенными, кроме тех случаев, где явно указано обратное (а рефлексивные привязки, также за немногочисленными исключениями, совпадают с конкретными). Конкретные привязки отличаются от обобщенных только для составных типов `record`, `enum` и `fixed`, для которых генерируются классы (имена которых определяются атрибутом `name` и необязательным атрибутом `namespace`).



Типу Avro `string` может соответствовать как тип Java `String`, так и тип Java `Utf8`. `Utf8` выбирается по соображениям эффективности: так как объекты этого типа являются изменяемыми, один экземпляр `Utf8` может повторно использоваться для чтения или записи серии значений. Кроме того, тип Java `String` декодирует `Utf8` в момент конструирования объекта, а тип Avro `Utf8` делает это в отложенном режиме, что в некоторых случаях приводит к повышению производительности.

`Utf8` реализует интерфейс Java `java.lang(CharSequence)`, обеспечивающий некоторую совместимость с библиотеками Java. В других случаях приходится преобразовывать экземпляры `Utf8` в объекты `String`, вызывая их метод `toString()`.

В Avro 1.6.0 и выше можно заставить Avro всегда выполнять преобразование в `String`. Это можно сделать двумя способами. Во-первых, свойству `avro.java.string` в схеме можно задать значение `String`:

```
{ "type": "string", "avro.java.string": "String" }
```

Во-вторых, для конкретной привязки можно сгенерировать классы с `get`- и `set`-методами на базе `String`. При использовании плагина Avro Maven для этого конфигурационному свойству `stringType` задается значение `String` (способ продемонстрирован в примерах кода, прилагаемых к книге).

Наконец, следует заметить, что в рефлексивной привязке Java всегда используются объекты `String`, так как эта разновидность привязок проектировалась для обеспечения совместимости с Java, а не для производительности.

Таблица 4.11. Привязки типов Avro в Java

Тип Avro	Обобщенная привязка Java	Конкретная привязка Java	Рефлексивная привязка Java
null	null		
boolean	boolean		
int	int		short или int
long	long		
float	float		
double	double		
bytes	java.nio.ByteBuffer		Массив byte
string	org.apache.avro.util.Utf8 или java.lang.String		java.lang.String
array	org.apache.avro.generic.GenericArray		Массив или java.util.Collection
map	java.util.Map		
record	org.apache.avro.generic.GenericRecord	Сгенерированный класс, реализующий org.apache.avro.specific.SpecificRecord	Произвольный пользовательский класс с безаргументным конструктором. Используются все унаследованные непереходные (nontransient) поля экземпляров
enum	java.lang.String	Сгенерированное перечисление Java	Произвольное перечисление Java
fixed	org.apache.avro.generic.GenericFixed	Сгенерированный класс, реализующий org.apache.avro.specific.SpecificFixed	org.apache.avro.generic.GenericFixed
union	java.lang.Object		

Сериализация и десериализация в памяти

Avro предоставляет API для сериализации и десериализации, полезные для интеграции Avro в существующие системы — например, в систему передачи сообщений, в которой уже определен формат кадров. В других случаях стоит рассмотреть возможность использования формата файлов данных Avro.

Напишем простую Java-программу для чтения и записи данных Avro в потоки. Начнем с простой схемы Avro для представления пары строк в виде записи:

```
{  
    "type": "record",  
    "name": "StringPair",  
    "doc": "A pair of strings.",  
    "fields": [  
        {"name": "left", "type": "string"},  
        {"name": "right", "type": "string"}  
    ]  
}
```

Если сохранить эту схему в файле с именем *StringPair.avsc* (.avsc — стандартное расширение для схем Avro), то для ее загрузки достаточно всего двух строк кода:

```
Schema.Parser parser = new Schema.Parser();  
Schema schema =  
    parser.parse(getClass().getResourceAsStream("StringPair.avsc"));
```

Экземпляр записи Avro в обобщенном API создается следующим образом:

```
GenericRecord datum = new GenericData.Record(schema);  
datum.put("left", "L");  
datum.put("right", "R");
```

Далее запись сериализуется в выходной поток:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();  
DatumWriter<GenericRecord> writer =  
    new GenericDatumWriter<GenericRecord>(schema);  
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);  
writer.write(datum, encoder);  
encoder.flush();  
out.close();
```

В этом фрагменте встречаются два важных объекта: **DatumWriter** и **Encoder**. Объект **DatumWriter** преобразует объекты данных в типы, воспринимаемые **Encoder**, которые последний записывает в выходной поток. В приведенном примере используется реализация **GenericDatumWriter**, которая передает **Encoder** поля **GenericRecord**. Фабрике **EncoderFactory** передается **null**, потому что мы не пытаемся повторно использовать ранее сконструированный объект **Encoder**.

В этом примере в поток записывается всего один объект, но при желании перед закрытием потока можно вызвать **write()** с другими объектами.

Объекту **GenericDatumWriter** необходимо передать схему, по которой он определяет, какие значения из объектов данных должны записываться в поток. После

вызыва метода `write()` мы выполняем сброс `Encoder` и закрываем выходной поток.

Чтение объекта из байтового буфера выполняется в обратном порядке:

```
DatumReader<GenericRecord> reader =
        new GenericDatumReader<GenericRecord>(schema);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(), null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
```

При вызовах `binaryDecoder()` и `read()` передаются значения `null`, потому что объекты (декодер и запись соответственно) не используются повторно.

Объекты, возвращаемые `result.get("left")` и `result.get("right")`, относятся к типу `Utf8`, поэтому мы преобразуем их в объекты Java `String`, вызывая их методы `toString()`.

Конкретный API

Посмотрим, как будет выглядеть эквивалентный код в конкретном API. Класс `StringPair` генерируется по файлу схемы при помощи плагина Avro Maven. Ниже приведена соответствующая часть модели POM (Project Object Model) Maven:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro-maven-plugin</artifactId>
        <version>${avro.version}</version>
        <executions>
          <execution>
            <id>schemas</id>
            <phase>generate-sources</phase>
            <goals>
              <goal>schema</goal>
            </goals>
            <configuration>
              <includes>
                <include>StringPair.avsc</include>
              </includes>
              <sourceDirectory>src/main/resources</sourceDirectory>
            
```

```
<outputDirectory>${project.build.directory}/generated-sources/java
    </outputDirectory>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
...
</project>
```

Вместо Maven для генерирования Java-кода схемы также можно воспользоваться Ant-задачей Avro `org.apache.avro.specific.SchemaTask` или средствами командной строки Avro¹.

В коде сериализации и десериализации вместо `GenericRecord` мы конструируем экземпляр `StringPair`, который записывается в поток с использованием `SpecificDatumWriter`, а затем читается с использованием `SpecificDatumReader`:

```
StringPair datum = new StringPair();
datum.left = "L";
datum.right = "R";
ByteArrayOutputStream out = new ByteArrayOutputStream();
DatumWriter<StringPair> writer =
    new SpecificDatumWriter<StringPair>(StringPair.class);
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();

DatumReader<StringPair> reader =
    new SpecificDatumReader<StringPair>(StringPair.class);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(), null);
StringPair result = reader.read(null, decoder);
assertThat(result.left.toString(), is("L"));
assertThat(result.right.toString(), is("R"));
```

Начиная с Avro 1.6.0, в генерируемый код Java включаются `get-` и `set-`методы, так что вместо приведенных конструкций можно использовать `datum.setLeft("L")` и `result.getLeft()`.

¹ Avro можно загрузить как в виде исходного кода, так и в двоичной форме по адресу <http://avro.apache.org/releases.html>. Чтобы получить инструкции по использованию инструментов Avro, введите команду `java -jar avro-tools-*jar`.

Файлы данных Avro

Формат контейнерных файлов Avro предназначен для хранения серий объектов Avro. По своей структуре он близок к формату последовательных файлов Hadoop, описанных в разделе «SequenceFile», с. 186. Главное различие заключается в том, что файлы данных Avro проектировались с расчетом на межъязыковую портируемость — то есть, например, вы можете записать файл на языке Python и прочитать его на C (именно это будет сделано в следующем разделе).

Файл данных содержит заголовок с метаданными, включая схему Avro и синхронизационный маркер, за которыми следует серия (возможно, сжатых) блоков с сериализованными объектами Avro. Блоки разделяются синхронизационным маркером, уникальным для файла (маркер конкретного файла находится в заголовке), что открывает возможность быстрой синхронизации по границе блока после перехода к произвольной позиции в файле (например, границе блока HDFS). Таким образом, файлы данных Avro поддерживают разбиение, что повышает эффективность их обработки в MapReduce.

Процесс записи объектов Avro в файл данных напоминает процесс записи в поток. Мы, как и прежде, используем `DatumWriter`, но вместо `Encoder` создается экземпляр `DataFileWriter`. Затем создается новый файл данных (по соглашению имеющий расширение `.avro`), к которому присоединяются объекты:

```
File file = new File("data.avro");
DatumWriter<GenericRecord> writer = new GenericDatumWriter<GenericRecord>(schema);
DataFileWriter<GenericRecord> dataFileWriter =
    new DataFileWriter<GenericRecord>(writer);
dataFileWriter.create(schema, file);
dataFileWriter.append(datum);
dataFileWriter.close();
```

Объекты, записываемые в файл данных, должны соответствовать его схеме; в противном случае при вызове `append()` произойдет исключение.

В примере используется локальный файл (`java.io.File` в предыдущем фрагменте), но запись также может осуществляться в произвольный поток `java.io.OutputStream`, для чего используется перегруженный метод `create()` объекта `DataFileWriter`. Например, чтобы записать файл в HDFS, следует получить объект `OutputStream`, вызвав `create()` для `FileSystem` (см. «Запись данных», с. 100).

Чтение объектов из файла данных происходит примерно так же, как в приведенном ранее примере чтения объектов из потока в памяти, с одним важным различием: указывать схему не обязательно, так как она читается из метаданных файла. Вместо этого мы получаем схему из экземпляра `DataFileReader` при помощи

`getSchema()` и убеждаемся в том, что она не отличается от схемы, использованной при записи исходного объекта:

```
DatumReader<GenericRecord> reader = new GenericDatumReader<GenericRecord>();
DataFileReader<GenericRecord> dataFileReader =
    new DataFileReader<GenericRecord>(file, reader);
assertThat("Schema is the same", schema, is(dataFileReader.getSchema()));
```

`DataFileReader` — обычный итератор Java, что позволяет нам перебирать объекты данных стандартными вызовами его методов `hasNext()` и `next()`. Следующий фрагмент проверяет, что в наборе осталась только одна запись и она содержит ожидаемые значения полей:

```
assertThat(dataFileReader.hasNext(), is(true));
GenericRecord result = dataFileReader.next();
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
assertThat(dataFileReader.hasNext(), is(false));
```

Вместо обычного метода `next()` лучше использовать перегруженную форму, получающую экземпляр возвращаемого объекта (в данном случае `GenericRecord`), потому что она заново использует объект и избавляет от затрат на создание и уборку мусора для файлов, содержащих многочисленные объекты. Стандартная идиома выглядит так:

```
GenericRecord record = null;
while (dataFileReader.hasNext()) {
    record = dataFileReader.next(record);
    // Обработка записи
}
```

Если повторное использование объектов для вас несущественно, используйте укороченную запись:

```
for (GenericRecord record : dataFileReader) {
    // Обработка записи
}
```

Для общего случая чтения файла из файловой системы Hadoop следует использовать класс Avro `FSInput` для определения входного файла с использованием объекта Hadoop `Path`. Вообще говоря, `DataFileReader` обеспечивает произвольный доступ к файлам данных Avro (с использованием методов `seek()` и `sync()`); тем не менее во многих случаях хватает последовательного потокового доступа, для которого следует использовать `DataFileStream`. Объект `DataFileStream` может выполнять чтение из любого потока Java `InputStream`.

Совместимость

Чтобы продемонстрировать языковую совместимость Avro, мы запишем файл данных на одном языке (Python), а прочитаем его на другом (C).

Python API

Программа из листинга 4.10 читает строки, разделенные запятыми, из стандартного ввода и записывает их в виде записей `StringPair` в файл данных Avro. Как и в коде записи файла данных на языке Java, мы создаем объекты `DatumWriter` и `DataFileWriter`. Схема Avro встроена в программный код, хотя с таким же успехом ее можно было загрузить из файла.

В Python записи Avro представлены словарями (dictionaries); каждая строка, прочитанная из стандартного ввода, преобразуется в объект словаря и присоединяется к `DataFileWriter`.

Листинг 4.10. Программа Python для записи пар в файл данных

```
import os
import string
import sys

from avro import schema
from avro import io
from avro import datafile

if __name__ == '__main__':
    if len(sys.argv) != 2:
        sys.exit('Usage: %s <data_file>' % sys.argv[0])
    avro_file = sys.argv[1]
    writer = open(avro_file, 'wb')
    datum_writer = io.DatumWriter()
    schema_object = schema.parse("\
{ \"type\": \"record\",
  \"name\": \"StringPair\",
  \"doc\": \"A pair of strings.\",
  \"fields\": [
    {\"name\": \"left\", \"type\": \"string\"},
    {\"name\": \"right\", \"type\": \"string\"}
  ]
}")
    dfw = datafile.DataFileWriter(writer, datum_writer, schema_object)
    for line in sys.stdin.readlines():
        (left, right) = string.split(line.strip(), ',')
```

```
    dfw.append({'left':left, 'right':right});  
dfw.close()
```

Прежде чем запускать программу, необходимо установить Avro для Python:

```
% easy_install avro
```

Для запуска программы мы указываем имя файла, в который будут записываться данные (*pairs.avro*), и направляем входные пары в стандартный ввод, отмечая конец данных комбинацией клавиш Ctrl+D:

```
% python avro/src/main/py/write_pairs.py pairs.avro  
a,1  
c,2  
b,3  
b,2  
^D
```

C API

Следующая программа — написанная на языке C — выводит содержимое *pairs.avro* (листинг 4.11¹).

Листинг 4.11. Программа на языке C для чтения пар из файла данных

```
#include <avro.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char *argv[]) {  
    if (argc != 2) {  
        fprintf(stderr, "Usage: dump_pairs <data_file>\n");  
        exit(EXIT_FAILURE);  
    }  
  
    const char *avrofile = argv[1];  
    avro_schema_error_t error;  
    avro_file_reader_t filereader;  
    avro_datum_t pair;  
    avro_datum_t left;  
    avro_datum_t right;  
    int rval;  
    char *p;
```

продолжение ↗

¹ В общем случае JAR-файл инструментария Avro содержит команду `tojson`, которая выводит содержимое файла данных Avro в формате JSON.

```
avro_file_reader(avrofile, &filereader);
```

Листинг 4.11 (продолжение)

```
while (1) {
    rval = avro_file_reader_read(filereader, NULL, &pair);
    if (rval) break;
    if (avro_record_get(pair, "left", &left) == 0) {
        avro_string_get(left, &p);
        fprintf(stdout, "%s,", p);
    }
    if (avro_record_get(pair, "right", &right) == 0) {
        avro_string_get(right, &p);
        fprintf(stdout, "%s\n", p);
    }
}
avro_file_reader_close(filereader);
return 0;
}
```

Фактически программа решает три задачи:

1. Открытие структуры чтения данных типа `avro_file_reader_t` вызовом функции Avro `avro_file_reader`¹.
2. Чтение данных Avro из структуры функцией `avro_file_reader_read` в цикле `while` до тех пор, пока не кончатся пары (проверка по возвращаемому значению `rval`).
3. Закрытие структуры чтения вызовом `avro_file_reader_close`.

Функция `avro_file_reader_read` получает во втором аргументе схему на тот случай, когда схема чтения отличается от схемы, использованной для записи (см. следующий раздел), но мы просто передаем `NULL`, тем самым приказывая Avro использовать схему файла данных. Третий аргумент содержит указатель на объект `avro_datum_t`, заполняемый содержимым следующей записи, прочитанной из файла. Структура пар распаковывается по полям вызовом `avro_record_get`, после чего вызов `avro_string_get` извлекает значения полей в строковом виде, которые выводятся на консоль.

При запуске программы с файлом, полученным в результате выполнения программы на Python, мы получаем исходные входные данные:

```
% ./dump_pairs pairs.avro
a,1
c,2
```

¹ Функции и типы Avro имеют префикс `avro_` и определяются в заголовочном файле `avro.h`.

```
b,3
b,2
```

Итак, мы успешно организовали передачу составных данных между двумя разными реализациями Avro.

Преобразование схемы

Схема, используемая для чтения данных (*схема чтения*), может отличаться от схемы, использованной для их записи (*схема записи*). Данное различие чрезвычайно важно, потому что оно открывает возможность эволюции схем. Для демонстрации рассмотрим новую схему для строковых пар с добавленным полем `description`:

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings with an added field.",
  "fields": [
    {"name": "left", "type": "string"},
    {"name": "right", "type": "string"},
    {"name": "description", "type": "string", "default": ""}
  ]
}
```

Эта схема может использоваться для чтения данных, сериализованных ранее, потому что для поля `description` указано значение по умолчанию (пустая строка)¹, которое будет использоваться Avro при отсутствии определения поля в читаемых записях. Если опустить атрибут `default`, то при попытке чтения старых данных произойдет ошибка.



Чтобы вместо пустой строки по умолчанию использовалось значение `null`, включите в определение поля `description` объединение с типом Avro `null`:

```
{"name": "description", "type": ["null", "string"], "default": null}
```

Когда схема чтения отличается от схемы записи, мы используем конструктор `GenericDatumReader`, получающий два объекта схемы — записи и чтения (в указанном порядке):

```
DatumReader<GenericRecord> reader =
```

продолжение ↗

¹ Значения полей по умолчанию кодируются в JSON. Описание этой кодировки для каждого типа данных приведено в спецификации Avro.

```

new GenericDatumReader<GenericRecord>(schema, newSchema);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(), null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
assertThat(result.get("description").toString(), is(""));

```

Для файлов данных, у которых схема записи хранится в метаданных, достаточно явно задать только схему чтения; для этого достаточно передать `null` вместо схемы записи:

```

DatumReader<GenericRecord> reader =
    new GenericDatumReader<GenericRecord>(null, newSchema);

```

Другое распространенное применение другой схемы чтения — исключение полей из записи. Это бывает полезно, если запись содержит большое количество полей и вы хотите прочитать только часть из них. Например, следующая схема может использоваться для получения только правого поля `StringPair`:

```

{
  "type": "record",
  "name": "StringPair",
  "doc": "The right field of a pair of strings.",
  "fields": [
    {"name": "right", "type": "string"}
  ]
}

```

Правила преобразования схем напрямую влияют на возможность эволюции схем между версиями. Они четко определены в спецификации Avro для всех типов Авто. Сводка правил эволюции с точки зрения сторон, выполняющих чтение и запись (или серверов и клиентов), представлена в табл. 4.12.

Таблица 4.12. Правила преобразования схем

Новая схема	Схема записи	Схема чтения	Действие
Добавление поля	Старая	Новая	При чтении используется значение по умолчанию нового поля, отсутствующее в записанных данных
	Новая	Старая	При чтении информация о новом поле, добавленном при записи, отсутствует, поэтому поле игнорируется

Новая схема	Схема записи	Схема чтения	Действие
Удаление поля	Старая	Новая	При чтении удаленное поле игнорируется
Удаление поля	Новая	Старая	Удаленное поле не сохраняется при записи. Если в старой схеме для него определено значение по умолчанию, оно используется при чтении; в противном случае происходит ошибка. В этом случае схему чтения лучше обновить — либо одновременно, либо до обновления схемы записи

Другой полезный метод эволюции схем Avro основан на использовании *псевдонимов* (aliases). С ними в схеме чтения данных Avro могут использоваться имена, отличающиеся от имен, использовавшихся для исходной записи данных. Например, следующая схема чтения может использоваться для чтения данных `StringPair` с новыми именами полей `first` и `second` — вместо имен `left` и `right`, с которыми данные были записаны.

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings with aliased field names.",
  "fields": [
    {"name": "first", "type": "string", "aliases": ["left"]},
    {"name": "second", "type": "string", "aliases": ["right"]}
  ]
}
```

Следует заметить, что псевдонимы используются для преобразования (в момент чтения) схемы записи в схему чтения, но исходные имена недоступны читающей стороне. В нашем примере сторона, читающая данные, не сможет использовать имена полей `left` и `right`, потому что они уже были преобразованы в `first` и `second`.

Порядок сортировки

Avro определяет порядок сортировки объектов. Для большинства типов Avro этот порядок выглядит естественно — например, числовые типы упорядочиваются по возрастанию числового значения. С другими типами дело обстоит сложнее. Например, перечисления сравниваются по порядку определения симвлических имен, а не по значениям строк.

Для всех типов, кроме записей, устанавливаются предопределенные правила сортировки, которые описаны в спецификации Avro и не могут переопределяться пользователем. Порядком сортировки записей можно управлять, задавая полям атрибут `order`. Атрибут принимает одно из трех значений: `ascending` (используется по умолчанию), `descending` (обратный порядок) и `ignore` (поле игнорируется при сравнении).

Например, следующая схема (*SortedStringPair.avsc*) сортирует записи `StringPair` по убыванию правого поля. Левое поле игнорируется для определения порядка сортировки, но остается в данных:

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings, sorted by right field descending.",
  "fields": [
    {"name": "left", "type": "string", "order": "ignore"},
    {"name": "right", "type": "string", "order": "descending"}
  ]
}
```

Поля записи сравниваются попарно в порядке схемы чтения. Таким образом, задавая подходящую схему чтения, можно установить произвольное упорядочение записей данных. Следующая схема (*SwitchedStringPair.avsc*) определяет порядок сортировки, при котором записи сначала упорядочиваются по правому полю, а затем по левому:

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings, sorted by right then left.",
  "fields": [
    {"name": "right", "type": "string"},
    {"name": "left", "type": "string"}
  ]
}
```

В Avro реализованы эффективные двоичные сравнения. Иначе говоря, Avro не нужно выполнять десериализацию двоичных данных в объекты для сравнения — вместо этого Avro работает прямо с байтовыми потоками¹. Например, в исходной схеме `StringPair` (без атрибутов `order`) Avro реализует двоичное сравнение

¹ У этого свойства есть одно полезное следствие: хеш-код данных Avro может вычисляться как по объекту, так и по двоичному представлению (во втором случае используется статический метод `hashCode()` с объектом `BinaryData`). В обоих случаях будут получены одинаковые результаты.

следующим образом. Первое поле `left` представляет собой строку в кодировке Utf8, для которой Avro может сравнить байты в лексикографическом порядке. Если они различаются, то порядок определен и Avro может на этом остановиться. Если же двухбайтовые последовательности совпадают, Avro сравнивает два вторых поля (`right`) — снова в лексикографическом порядке на уровне байтов, потому что поле также представляет собой строку в формате Utf8.

Обратите внимание: функция сравнения основана на такой же логике, как и двоичный объект сравнения, написанный для `Writable` в разделе «Реализация Raw-Comparator» на с. 155. При этом Avro предоставляет реализацию сравнения, так что нам не придется заниматься написанием и сопровождением кода. Кроме того, порядок сортировки легко изменяется простым редактированием схемы чтения. Для схем `SortedStringPair.avsc` и `SwitchedStringPair.avsc` функция сравнения, используемая Avro, по сути, не отличается от только что описанной. Различаются только поля, порядок их рассмотрения и порядок сортировки (по возрастанию или по убыванию).

Позднее в этой главе мы используем логику сортировки Avro в сочетании с MapReduce для организации параллельной сортировки файлов данных Avro.

Avro и MapReduce

Avro содержит ряд классов, упрощающих выполнение программ MapReduce с данными Avro. Например, классы `AvroMapper` и `AvroReducer` из пакета `org.apache.avro.mapred` представляют собой специализации (традиционных) классов Hadoop `Mapper` и `Reducer`. Они устраниют различия между ключами и значениями для ввода и вывода, так как файлы данных Avro представляют собой простые последовательности значений. Однако промежуточные данные все равно делятся на пары «ключ-значение» для тасовки (`shuffle`).

Переработаем программу MapReduce для определения максимальной температуры по каждому году в наборе погодных данных — на этот раз с использованием Avro MapReduce API. Для представления записей метеорологических данных будет использоваться следующая схема:

```
{  
  "type": "record",  
  "name": "WeatherRecord",  
  "doc": "A weather reading.",  
  "fields": [  
    {"name": "year", "type": "int"},  
    {"name": "temperature", "type": "int"},  
  ]}
```

продолжение ↗

```

        {"name": "stationId", "type": "string"}
    ]
}

```

Программа из листинга 4.12 читает текстовые данные (в формате, который уже встречался нам в предшествующих главах) и записывает выходные файлы данных Avro.

Листинг 4.12. Программа поиска максимальной температуры, создающая выходные данные Avro

```

public class AvroGenericMaxTemperature extends Configured implements Tool {

    private static final Schema SCHEMA = new Schema.Parser().parse(
        "{" +
            " \\"type\\": \\"record\\\", " +
            " \\"name\\": \\"WeatherRecord\\\", " +
            " \\"doc\\": \\"A weather reading.\", " +
            " \\"fields\\": [ " +
                "   {\\"name\\": \\"year\\", \\"type\\": \\"int\\\"}, " +
                "   {\\"name\\": \\"temperature\\", \\"type\\": \\"int\\\"}, " +
                "   {\\"name\\": \\"stationId\\", \\"type\\": \\"string\\\"} " +
            " ] " +
        "}"
    );
}

public static class MaxTemperatureMapper
    extends AvroMapper<Utf8, Pair<Integer, GenericRecord>> {
    private NcdcRecordParser parser = new NcdcRecordParser();
    private GenericRecord record = new GenericData.Record(SCHEMA);
    @Override
    public void map(Utf8 line,
                    AvroCollector<Pair<Integer, GenericRecord>> collector,
                    Reporter reporter) throws IOException {
        parser.parse(line.toString());
        if (parser.isValidTemperature()) {
            record.put("year", parser.getYearInt());
            record.put("temperature", parser.getAirTemperature());
            record.put("stationId", parser.getStationId());
            collector.collect(
                new Pair<Integer, GenericRecord>(parser.getYearInt(), record));
        }
    }
}

public static class MaxTemperatureReducer

```

```
extends AvroReducer<Integer, GenericRecord, GenericRecord> {  
  
    @Override  
    public void reduce(Integer key, Iterable<GenericRecord> values,  
                      AvroCollector<GenericRecord> collector, Reporter reporter)  
        throws IOException {  
        GenericRecord max = null;  
        for (GenericRecord value : values) {  
            if (max == null ||  
                (Integer) value.get("temperature") > (Integer) max.  
get("temperature")) {  
                max = newWeatherRecord(value);  
            }  
        }  
        collector.collect(max);  
    }  
    private GenericRecord newWeatherRecord(GenericRecord value) {  
        GenericRecord record = new GenericData.Record(SCHEMA);  
        record.put("year", value.get("year"));  
        record.put("temperature", value.get("temperature"));  
        record.put("stationId", value.get("stationId"));  
        return record;  
    }  
}  
  
    @Override  
    public int run(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.err.printf("Usage: %s [generic options] <input> <output>\n",  
                             getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.err);  
            return -1;  
        }  
  
        JobConf conf = new JobConf(getConf(), getClass());  
        conf.setJobName("Max temperature");  
  
        FileInputFormat.addInputPath(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        AvroJob.setInputSchema(conf, Schema.create(Schema.Type.STRING));  
        AvroJob.setMapOutputSchema(conf,  
                                  Pair.getPairSchema(Schema.create(Schema.Type.INT), SCHEMA));  
        AvroJob.setOutputSchema(conf, SCHEMA);  
    }
```

Листинг 4.12 (продолжение)

```

        conf.setInputFormat(AvroUtf8InputFormat.class);

        AvroJob.setMapperClass(conf, MaxTemperatureMapper.class);
        AvroJob.setReducerClass(conf, MaxTemperatureReducer.class);

        JobClient.runJob(conf);
        return 0;
    }

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

```

Программа использует обобщенную привязку Avro. Это обстоятельство избавляет нас от генерирования кода представления записей за счет безопасности типов (для ссылок на имена полей используются строковые значения — например, "temperature")¹. Схема записей метеорологических данных для удобства встроена в код (и читается в константу **SCHEMA**), хотя на практике схема, вероятно, будет читаться из локального файла в коде управляющей программы и передаваться функциям отображения и свертки через конфигурацию задания Hadoop. (О том, как это делается, рассказано в разделе «Распространение побочных данных» на с. 374.)

Обратите внимание на пару отличий от обычного Hadoop MapReduce API. Во-первых, для упаковки выходного ключа и значения отображения в **MaxTemperatureMapper** используется класс **org.apache.avro.mapred.Pair**. (Класс **org.apache.avro.mapred.AvroMapper** не имеет фиксированного выходного ключа и значения для того, чтобы задания, состоящие только из отображений, могли выводить в файлы данных Avro только значения.) В нашей программе MapReduce ключом является год (целое число), а значением — запись метеорологических данных, представленная классом **Avro GenericRecord**.

Однако Avro MapReduce сохраняет пары «ключ-значение» входных данных для свертки, потому что эти пары являются результатом тасовки, и распаковывает **Pair** перед обращением к **org.apache.avro.mapred.AvroReducer**. **MaxTemperatureReducer**, перебирает записи для каждого ключа (года) и находит запись с максимальной температурой. Так как итератор заново использует экземпляр по соображениям эффективности (с обновлением полей), необходимо создать копию записи с максимальной температурой, найденной на данный момент.

¹ Использование конкретной привязки с генерируемыми классами продемонстрировано в классе **AvroSpecificMaxTemperature** из кода примеров.

Второе существенное отличие от обычного использования MapReduce — применение `AvroJob` для настройки задания. Вспомогательный класс `AvroJob` предназначен для определения схем Avro для ввода, выходных данных отображений и окончательных выходных данных. В нашей программе входная схема представляет собой строку Avro, потому что мы читаем данные из текстового файла и устанавливаем соответствующий формат `AvroUtf8InputFormat`. У выходной схемы отображения ключ имеет схему Avro `int`, а значение — схему записи погодных данных. Итоговая выходная схема описывает запись метеорологических данных, а выходной формат по умолчанию `AvroOutputFormat` сохраняет информацию в файлах данных Avro.

Следующая командная строка запускает программу для небольшого набора тестовых данных:

```
% hadoop jar avro-examples.jar AvroGenericMaxTemperature \
  input/ncdc/sample.txt output
```

Чтобы просмотреть выходные данные после завершения программы, можно воспользоваться JAR-файлом инструментария Avro для построчного преобразования файла данных Avro в формат JSON:

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson output/part-00000.avro
>{"year":1949,"temperature":111,"stationId":"012650-99999"}
>{"year":1950,"temperature":22,"stationId":"011990-99999"}
```

В этом примере мы используем `AvroMapper` и `AvroReducer`, но API поддерживает любые комбинации стандартных функций отображения и свертки и функций, специфических для Avro. Данная возможность может пригодиться для выполнения преобразований между форматами Avro и другими форматами — например, `SequenceFile`. За подробностями обращайтесь к документации пакета Avro MapReduce.

Сортировка с использованием Avro MapReduce

В этом разделе мы напишем программу для сортировки файлов данных Avro, используя для этого средства сортировки Avro в сочетании с MapReduce (листинг 4.13).

Листинг 4.13. Программа MapReduce для сортировки файла данных Avro

```
public class AvroSort extends Configured implements Tool {
  static class SortMapper<K> extends AvroMapper<K, Pair<K, K>> {
    public void map(K datum, AvroCollector<Pair<K, K>> collector,
```

продолжение ↗

Листинг 4.13 (продолжение)

```
    Reporter reporter) throws IOException {
        collector.collect(new Pair<K, K>(datum, null, datum, null));
    }
}

static class SortReducer<K> extends AvroReducer<K, K, K> {
    public void reduce(K key, Iterable<K> values,
                       AvroCollector<K> collector,
                       Reporter reporter) throws IOException {
        for (K value : values) {
            collector.collect(value);
        }
    }
}

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

    if (args.length != 3) {
        System.err.printf(
            "Usage: %s [generic options] <input> <output> <schema-file>\n",
            getClass().getSimpleName());
        ToolRunner.printGenericCommandUsage(System.err);
        return -1;
    }

    String input = args[0];
    String output = args[1];
    String schemaFile = args[2];

    JobConf conf = new JobConf(getConf(), getClass());
    conf.setJobName("Avro sort");

    FileInputFormat.addInputPath(conf, new Path(input));
    FileOutputFormat.setOutputPath(conf, new Path(output));

    Schema schema = new Schema.Parser().parse(new File(schemaFile));
    AvroJob.setInputSchema(conf, schema);
    Schema intermediateSchema = Pair.getPairSchema(schema, schema);
    AvroJob.setMapOutputSchema(conf, intermediateSchema);
    AvroJob.setOutputSchema(conf, schema);

    AvroJob.setMapperClass(conf, SortMapper.class);
```

```
    AvroJob.setReducerClass(conf, SortReducer.class);

    JobClient.runJob(conf);
    return 0;
}

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

Эта программа (использующая обобщенную привязку Avro, а следовательно, не требующая генерирования кода) может сортировать записи Avro любого типа, представленные в JavaScript обобщенным параметром типа K. Мы выбираем значение, совпадающее с ключом, чтобы при группировке значений по ключу в случае совпадения ключей (в соответствии с функцией сортировки) выдавались все значения без потери записей¹. Функция отображения просто выдает объект `org.apache.avro.mapred.Pair` с соответствующим ключом и значением.

Сортировка выполняется на стадии тасовки MapReduce. Функция сортировки определяется схемой Avro, переданной программе. Воспользуемся программой для сортировки созданного ранее файла `pairs.avro`, используя схему `SortedStringPair.avsc` для сортировки поля `right` по убыванию. Начнем с просмотра входных данных:

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson input/avro/pairs.avro
>{"left":"a","right":1}
>{"left":"c","right":2}
>{"left":"b","right":3}
>{"left":"b","right":2}
```

Затем проведем сортировку:

```
% hadoop jar avro-examples.jar AvroSort input/avro/pairs.avro output \
ch04-avro/src/main/resources/SortedStringPair.avsc
```

В завершение просмотрим вывод и убедимся в том, что данные были отсортированы правильно.

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson output/part-00000.avro
{"left":"b","right":3}
```

продолжение ↗

¹ Идея дублирования информации из ключа в значении снова встречается в разделе «Вторичная сортировка», с. 361.

```
{"left":"c","right":"2"}  
{"left":"b","right":"2"}  
{"left":"a","right":"1"}
```

Avro MapReduce в других языках

В других языках, кроме Java, выбор возможностей для работы с данными Avro не столь широк.

Класс `AvroAsTextInputFormat` был создан для того, чтобы программы Hadoop Streaming могли читать файлы данных Avro. Каждая единица данных в файле преобразуется в строку, являющуюся ее JSON-представлением, или просто в байты для типа Avro `bytes`. Также можно назначить заданию Streaming выходной формат `AvroTextOutputFormat`, чтобы создать файлы данных Avro со схемой `bytes`, в которых каждая единица данных представляет собой пару «ключ-значение», разделенную символами табуляции и записанную из вывода Streaming. Оба класса находятся в пакете `org.apache.avro.mapred`.

Кроме того, Avro предоставляет связующую инфраструктуру (пакет `org.apache.avro.mapred.tether`) — Avro-аналог Hadoop Pipes с более мощным интерфейсом, чем у Streaming. На момент написания книги привязок для других языков не существовало, но в будущих выпусках появится реализация для Python.

Также при обработке данных Avro стоит рассмотреть возможность использования Pig и Hive, поскольку обе платформы позволяют читать и записывать файлы данных Avro при указании соответствующих форматов хранения данных.

Файловые структуры данных

В некоторых приложениях для хранения данных необходимо использовать специализированные структуры данных. При обработке данных на базе MapReduce вариант с размещением каждого блока двоичных данных в отдельном файле не масштабируется, поэтому в Hadoop для подобных ситуаций был создан ряд контейнеров более высокого уровня.

SequenceFile

Представьте себе журнальный файл, в котором каждая запись представлена отдельной строкой текста. Если вы захотите сохранить в журнале двоичные данные, простой текстовый формат для этого не подойдет. Класс Hadoop `SequenceFile`

выручит в подобной ситуации: он предоставляет структуру данных для двоичных пар «ключ-значение». Чтобы использовать его в качестве формата журнального файла, следует выбрать ключ (например, временную метку в формате `LongWritable`), а значением будет объект `Writable`, представляющий сохраняемые данные.

`SequenceFile` также хорошо подходит в качестве контейнера для файлов небольшого размера. HDFS и MapReduce оптимизированы для больших файлов, поэтому упаковка файлов в `SequenceFile` существенно повышает эффективность хранения и обработки малых файлов. (В разделе «Обработка всего файла как записи» на с. 318 приведена программа для упаковки файлов в `SequenceFile`).¹

Запись `SequenceFile`

Для создания объекта `SequenceFile` используйте один из его статических методов `createWriter()`, возвращающих экземпляр `SequenceFile.Writer`. Существует несколько перегруженных версий, но всем им передается поток, в который будет осуществляться запись (либо `FSDataOutputStream`, либо пара из `FileSystem` и `Path`), объект `Configuration`, а также типы ключа и значения. В дополнительных аргументах могут передаваться тип сжатия и кодек, функция обратного вызова для оповещения о ходе записи и экземпляр `Metadata`, который должен быть сохранен в заголовке `SequenceFile`.

Ключи и значения, хранящиеся в `SequenceFile`, не обязательно должны быть реализациями `Writable`. Допускается использование любых типов, сериализуемых и десериализуемых с использованием `Serialization`.

Располагая объектом `SequenceFile.Writer`, вы можете записывать пары «ключ-значение» с использованием метода `append()`. После завершения записи вызывается метод `close()` (`SequenceFile.Writer` реализует интерфейс `java.io.Closeable`).

В листинге 4.14 приведена короткая программа для записи пар «ключ-значение» в `SequenceFile` с использованием описанного API.

Листинг 4.14. Запись в `SequenceFile`

```
public class SequenceFileWriteDemo {  
  
    private static final String[] DATA = {  
        "One, two, buckle my shoe",  
        "Three, four, shut the door",  
    };
```

продолжение ↗

¹ В статье «A Million Little Files» из блога Стюарта Съера (Stuart Sierra) приведен код преобразования tar-файла в `SequenceFile`.

Листинг 4.14 (продолжение)

```

    "Five, six, pick up sticks",
    "Seven, eight, lay them straight",
    "Nine, ten, a big fat hen"
};

public static void main(String[] args) throws IOException {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    Path path = new Path(uri);

    IntWritable key = new IntWritable();
    Text value = new Text();
    SequenceFile.Writer writer = null;
    try {
        writer = SequenceFile.createWriter(fs, conf, path,
            key.getClass(), value.getClass());

        for (int i = 0; i < 100; i++) {
            key.set(100 - i);
            value.set(DATA[i % DATA.length]);
            System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
            writer.append(key, value);
        }
    } finally {
        IOUtils.closeStream(writer);
    }
}
}

```

Ключами последовательного файла являются целые числа от 100 до 1, представленные объектами `IntWritable`, а значениями — объекты `Text`. Перед присоединением каждой записи к `SequenceFile.Writer` мы вызываем метод `getLength()` для определения текущей позиции в файле. (Эта информация будет использоваться в следующем разделе, когда мы займемся непоследовательным чтением файла.) Позиция выводится на консоль вместе с парами «ключ-значение». Результат выполнения программы выглядит так:

```
% hadoop SequenceFileWriteDemo numbers.seq
[128] 100      One, two, buckle my shoe
[173] 99       Three, four, shut the door
[220] 98       Five, six, pick up sticks
[264] 97       Seven, eight, lay them straight
```

```
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
...
[1976] 60 One, two, buckle my shoe
[2021] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen
```

Чтение из SequenceFile

Чтобы прочитать последовательный файл от начала до конца, достаточно создать экземпляр `SequenceFile.Reader` и перебрать записи, многократно вызывая один из его методов `next()`. Выбор метода зависит от того, какую инфраструктуру сериализации вы используете. Если используются типы `Writable`, используйте метод `next()`, который получает аргументы ключа и значения и читает в них следующий ключ и значение из потока:

```
public boolean next(Writable key, Writable val)
```

Возвращаемое значение равно `true`, если пара «ключ-значение» была успешно прочитана, или `false` при достижении конца файла.

Для других инфраструктур сериализации (например, Apache Thrift) используйте следующие два метода:

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

В этом случае необходимо проследить за тем, чтобы сериализация, которую вы хотите использовать, была задана в свойстве `io.serializations`; см. «Программные среды сериализации», с. 158.

Если метод `next()` возвращает объект, отличный от `null`, значит, пара «ключ-значение» была прочитана из потока, а значение может быть получено методом

`getCurrentValue()`. Если же `next()` возвращает `null`, значит, достигнут конец файла.

Программа в листинге 4.15 демонстрирует чтение последовательного файла с `Writable`-ключами и значениями. Обратите внимание на получение информации о типах от `SequenceFile.Reader` вызовами `getKeyClass()` и `getValueClass()`, с последующим использованием `ReflectionUtils` для создания экземпляров ключа и значения. Полученная программа может использоваться с любым последовательным файлом, имеющим `Writable`-ключи и значения.

Листинг 4.15. Чтение из SequenceFile

```
public class SequenceFileReadDemo {  
  
    public static void main(String[] args) throws IOException {  
        String uri = args[0];  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(URI.create(uri), conf);  
        Path path = new Path(uri);  
  
        SequenceFile.Reader reader = null;  
        try {  
            reader = new SequenceFile.Reader(fs, path, conf);  
            Writable key = (Writable)  
                ReflectionUtils.newInstance(reader.getKeyClass(), conf);  
            Writable value = (Writable)  
                ReflectionUtils.newInstance(reader.getValueClass(), conf);  
            long position = reader.getPosition();  
            while (reader.next(key, value)) {  
                String syncSeen = reader.syncSeen() ? "*" : " ";  
                System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);  
                position = reader.getPosition(); // Начало следующей записи  
            }  
        } finally {  
            IOUtils.closeStream(reader);  
        }  
    }  
}
```

Еще одна особенность программы — вывод позиций точек синхронизации в последовательном файле. Точной синхронизации называется точка потока, которая может использоваться для восстановления синхронизации по границе записи при ее «потере» — например, после перехода к произвольной позиции в потоке. Точки синхронизации фиксируются объектом `SequenceFile.Writer`, который вставляет

специальную метку точки синхронизации после вывода некоторого количества записей в процессе вывода последовательного файла. Метки достаточно малы, чтобы побочные затраты на их хранение были незначительными — менее 1%. Точки синхронизации всегда выравниваются по границам записей.

При выполнении программы в листинге 4.15 точки синхронизации в последовательном файле обозначаются звездочками. Первая находится в позиции 2021 (а вторая — в позиции 4075, но в результатах она не показана):

```
% hadoop SequenceFileReadDemo numbers.seq
[128] 100 One, two, buckle my shoe
[173] 99 Three, four, shut the door
[220] 98 Five, six, pick up sticks
[264] 97 Seven, eight, lay them straight
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
[590] 90 One, two, buckle my shoe
...
[1976] 60 One, two, buckle my shoe
[2021*] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen
```

Существует два способа позиционирования к заданной позиции в последовательном файле. Метод `seek()` перемещает объект чтения в заданную позицию файла. Например, позиционирование на границу записи работает именно так, как и ожидается:

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat((IntWritable) key.get(), is(95));
```

Но если текущая позиция в файле находится не на границе записи, при вызове `next()` произойдет ошибка:

```
reader.seek(360);
reader.next(key, value); // Ошибка с выдачей IOException
```

Второй способ поиска границ записей основан на использовании точек синхронизации. Метод `sync(long position)` класса `SequenceFile.Reader` перемещает объект чтения к следующей точке синхронизации после `position`. (Если в файле не осталось точек синхронизации, объект чтения переводится в конец файла.) Таким образом, `sync()` может использоваться с любой позицией потока — например, не находящейся на границе записи. Объект чтения перемещается к следующей точке синхронизации для продолжения чтения:

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```



Класс `SequenceFile.Writer` содержит метод `sync()` для вставки точки синхронизации в текущую позицию потока. Не путайте его с одноименным, но в остальном с ним никак не связанным методом `sync()`, определенным в интерфейсе `Syncable` для синхронизации буферов с устройством.

Полезность точек синхронизации в полной мере проявляется при использовании последовательных файлов в качестве входных данных MapReduce, так как они позволяют разбивать файл и передавать его части для независимой обработки разными задачами отображения (см. «`SequenceFileInputFormat`», с. 327).

Вывод содержимого `SequenceFile` в интерфейсе командной строки

Команда `hadoop fs` поддерживает параметр `-text` для вывода содержимого последовательных файлов в текстовой форме. В этом режиме команда пытается распознать сигнатуру (magic number) файла и преобразовать его в текст. Распознаются файлы, сжатые в формате gzip, и последовательные файлы; в остальных случаях предполагается, что входные данные хранятся в формате простого текста.

Для последовательных файлов эта команда действительно полезна только в том случае, если ключи и значения имеют осмысленное строковое представление (определенное методом `toString()`). Кроме того, если вы используете собственные классы ключей и значений, необходимо проследить за тем, чтобы они находились в каталоге, входящем в путь классов Hadoop. Для последовательного файла, созданного в предыдущем разделе, выводится следующий результат:

```
% hadoop fs -text numbers.seq | head
100      One, two, buckle my shoe
```

```
99      Three, four, shut the door
98      Five, six, pick up sticks
97      Seven, eight, lay them straight
96      Nine, ten, a big fat hen
95      One, two, buckle my shoe
94      Three, four, shut the door
93      Five, six, pick up sticks
92      Seven, eight, lay them straight
91      Nine, ten, a big fat hen
```

Сортировка и слияние SequenceFile

Самый мощный механизм сортировки (и слияния) одного или нескольких последовательных файлов основан на использовании MapReduce. Технология MapReduce параллельна по своей природе; задавая количество используемых сверток, вы определяете количество выходных разделов. Например, с одной сверткой будет получен один выходной файл. Следующий пример сортировки взят из поставки Hadoop — мы указываем, что в качестве входных и выходных данных используются последовательные файлы, и задаем типы ключей и значений:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*examples.jar sort -r 1 \
-inFormat org.apache.hadoop.mapred.SequenceFileInputFormat \
-outFormat org.apache.hadoop.mapred.SequenceFileOutputFormat \
-outKey org.apache.hadoop.io.IntWritable \
-outValue org.apache.hadoop.io.Text \
numbers.seq sorted
% hadoop fs -text sorted/part-00000 | head
1      Nine, ten, a big fat hen
2      Seven, eight, lay them straight
3      Five, six, pick up sticks
4      Three, four, shut the door
5      One, two, buckle my shoe
6      Nine, ten, a big fat hen
7      Seven, eight, lay them straight
8      Five, six, pick up sticks
9      Three, four, shut the door
10     One, two, buckle my shoe
```

Сортировка более подробно рассматривается в разделе «Сортировка», с. 350.

Вместо использования MapReduce для сортировки/слияния также можно воспользоваться классом `SequenceFile.Sorter`, содержащим набор методов `sort()` и `merge()`. Эти функции появились до MapReduce и работают на более низком уровне, чем MapReduce (например, для обеспечения параллелизма данные

приходится разделять вручную), так что в общем случае MapReduce является предпочтительным механизмом организации сортировки и слияния последовательных файлов.

Формат SequenceFile

Последовательный файл состоит из заголовка, за которым следует одна или несколько записей (рис. 4.2). Первые три байта последовательного файла содержат сигнатуру SEQ, за которой следует один байт с номером версии. Заголовок содержит и другие поля, включая имена классов ключей и значений, параметры сжатия, определяемые пользователем метаданные и маркер синхронизации¹. Как было сказано ранее, маркер синхронизации используется для синхронизации объекта чтения по границе записи из произвольной позиции файла. Каждый файл содержит случайно сгенерированный маркер синхронизации, значение которого хранится в заголовке. Маркеры синхронизации вставляются между записями последовательного файла. Система маркеров проектировалась так, чтобы дополнительные затраты на их хранение составляли менее 1%, поэтому маркеры не всегда вставляются между каждой парой записей.

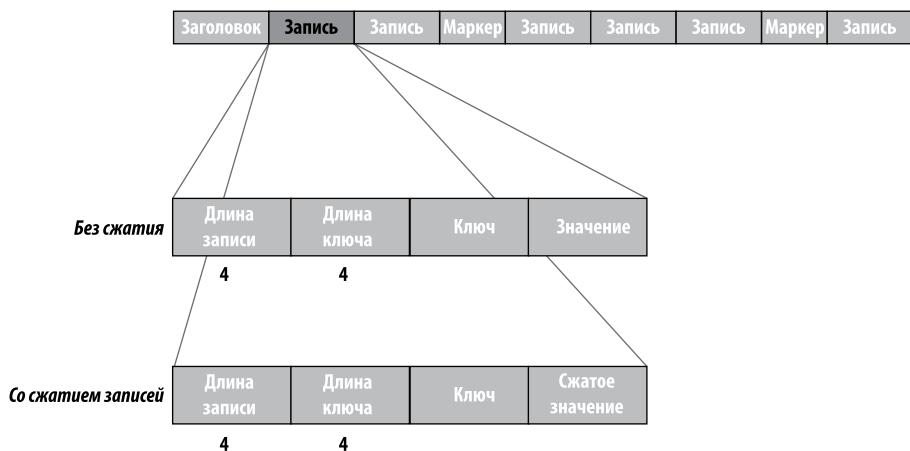


Рис. 4.2. Внутренняя структура последовательного файла без сжатия и со сжатием записей

Внутренний формат записей зависит от того, включено ли сжатие, и если включено — то какое именно (сжатие записей или сжатие блоков).

¹ Полное описание формата этих полей приведено в документации SequenceFile и в исходном коде.

Если сжатие не включено (по умолчанию), каждая запись состоит из длины записи (в байтах), длины ключа, самого ключа и значения. Поля длины записываются в виде четырехбайтовых целых в соответствии с контрактом метода `writeInt()` класса `java.io.DataOutput`. Ключи и значения сериализуются с использованием реализации `Serialization`, определенной для класса, записываемого в последовательный файл.

Формат сжатия записей почти идентичен отсутствию сжатия, не считая того, что байты значения сжимаются с использованием кодека, определенного в заголовке. Обратите внимание: ключи при этом не сжимаются.

В режиме сжатия блоков одновременно сжимаются сразу несколько записей; таким образом, этот формат сжатия более компактен и обычно является предпочтительным, потому что в нем может использоваться сходство содержимого записей (рис. 4.3). Записи добавляются в блок вплоть до достижения минимального размера в байтах, определяемого свойством `io.seqfile.compress.blocksize`; по умолчанию он равен 1 миллиону байт. Перед началом каждого блока записывается маркер синхронизации. Формат блока состоит из поля, обозначающего количество записей в блоке, за которым следуют четыре сжатых поля: длины ключей, ключи, длины значений и значения.

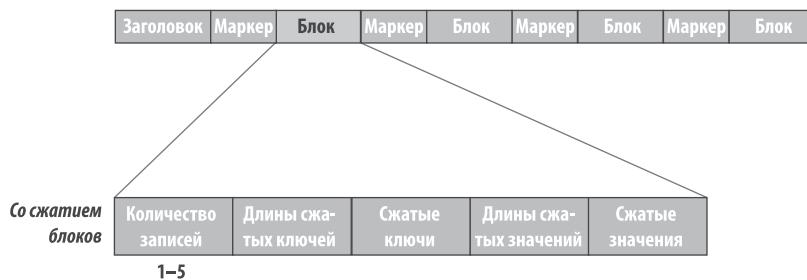


Рис. 4.3. Внутренняя структура последовательного файла со сжатием блоков

MapFile

Класс `MapFile` представляет отсортированный объект `SequenceFile` с индексом для поиска по ключу. `MapFile` может использоваться как разновидность `java.util.Map` (хотя и не реализует этот интерфейс), способная расти за пределы контейнера `Map`, хранящегося в памяти.

Запись MapFile

Запись в `MapFile` сходна с записью в `SequenceFile`: вы точно так же создаете экземпляр `MapFile.Writer`, а затем вызываете метод `append()` для упорядоченного присоединения записей данных. (Попытка добавления записей с нарушением порядка приводит к `IOException`.) Ключи должны быть экземплярами `WritableComparable`, а значения — экземплярами `Writable`. Сравните с классом `SequenceFile`, который позволяет использовать для своих записей любую инфраструктуру сериализации.

Программа в листинге 4.16 создает объект `MapFile` и записывает в него данные. По своей структуре она очень похожа на программу создания `SequenceFile` из листинга 4.14.

Листинг 4.16. Запись MapFile

```
public class MapFileWriteDemo {  
  
    private static final String[] DATA = {  
        "One, two, buckle my shoe",  
        "Three, four, shut the door",  
        "Five, six, pick up sticks",  
        "Seven, eight, lay them straight",  
        "Nine, ten, a big fat hen"  
    };  
  
    public static void main(String[] args) throws IOException {  
        String uri = args[0];  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(URI.create(uri), conf);  
  
        IntWritable key = new IntWritable();  
        Text value = new Text();  
        MapFile.Writer writer = null;  
        try {  
            writer = new MapFile.Writer(conf, fs, uri,  
                key.getClass(), value.getClass());  
  
            for (int i = 0; i < 1024; i++) {  
                key.set(i + 1);  
                value.set(DATA[i % DATA.length]);  
                writer.append(key, value);  
            }  
        }  
    }  
}
```

```
    } finally {
        IOUtils.closeStream(writer);
    }
}
```

Используем эту программу для построения MapFile:

```
% hadoop MapFileWriteDemo numbers.map
```

Присмотревшись к MapFile, мы видим, что в действительности это каталог, содержащий два файла с именами *data* и *index*:

```
% ls -l numbers.map
total 104
-rw-r--r-- 1 tom tom 47898 Jul 29 22:06 data
-rw-r--r-- 1 tom tom 251 Jul 29 22:06 index
```

Оба файла имеют формат SequenceFile. Файл *data* содержит все записи данных, расположенные по порядку:

```
% hadoop fs -text numbers.map/data | head
1      One, two, buckle my shoe
2      Three, four, shut the door
3      Five, six, pick up sticks
4      Seven, eight, lay them straight
5      Nine, ten, a big fat hen
6      One, two, buckle my shoe
7      Three, four, shut the door
8      Five, six, pick up sticks
9      Seven, eight, lay them straight
10     Nine, ten, a big fat hen
```

В файле *index* хранится часть ключей и информация о соответствии ключей и их смещений в файле данных:

```
% hadoop fs -text numbers.map/index
1      128
129    6079
257    12054
385    18030
513    24002
641    29976
769    35947
897    41922
```

Как видно из выходных данных, по умолчанию в индекс включается только каждый 128-й ключ, хотя этот параметр можно изменить — либо задав свойство `io.map.index.interval`, либо вызовом метода `setIndexInterval()` для экземпляра `MapFile.Writer`.

Например, индексный интервал можно увеличить для сокращения объема памяти, необходимой `MapFile` для хранения индекса. И наоборот, уменьшение индексного интервала улучшит время произвольной выборки (так как в среднем будет пропускаться меньше записей) за счет дополнительных затрат памяти.

Так как индекс содержит только часть ключей, класс `MapFile` не может предоставить методы для перебора (или хотя бы подсчета) всех содержащихся в нем ключей. Единственный способ выполнения этих операций — чтение всего файла.

Чтение из `MapFile`

Последовательный перебор записей `MapFile` напоминает аналогичную процедуру для `SequenceFile`: вы создаете экземпляр `MapFile.Reader`, затем вызываете метод `next()` до тех пор, пока он не вернет `false` (признак достижения конца файла):

```
public boolean next(WritableComparable key, Writable val) throws IOException.
```

Произвольный доступ к записям осуществляется вызовом метода `get()`:

```
public Writable get(WritableComparable key, Writable val) throws IOException.
```

По возвращаемому значению можно определить, была ли запись найдена в `MapFile`; если оно равно `null`, значение для указанного ключа не существует. Если ключ был найден, его значение читается в `val`, а также возвращается вызовом метода.

Полезно рассмотреть реализацию этого метода. Следующий фрагмент кода читает запись данных для объекта `MapFile`, созданного в предыдущем разделе:

```
Text value = new Text();
reader.get(new IntWritable(496), value);
assertThat(value.toString(), is("One, two, buckle my shoe"));
```

В этой операции `MapFile.Reader` читает файл `index` в память (данные кэшируются, чтобы при последующих обращениях произвольного доступа использовался тот же индекс в памяти). Затем `Reader` выполняет двоичный поиск по индексу в памяти для поиска ключа, меньшего либо равного искомому ключу 496. В нашем примере будет найден ключ 385 со значением 18030, которое определяет смещение в файле данных. Затем `MapFile.Reader` переходит к этому смещению в файле данных и читает записи до тех пор, пока ключ не будет больше либо равен искомому (496). В последнем случае совпадение найдено, и значение читается из файла данных. В целом поиск требует одной операции позиционирования и перебора до

128 записей. Для операций произвольного доступа такая эффективность достаточно высока.

Метод `getClosest()` похож на `get()`, но при неудачном поиске вместо `null` он возвращает совпадение, «ближайшее» к заданному ключу. Точнее, если `MapFile` содержит заданный ключ, то возвращается соответствующая запись, а если нет — то возвращается ключ `MapFile`, находящийся непосредственно перед (или после — в зависимости от логического аргумента) заданным ключом.

Очень большой индекс `MapFile` может занимать много памяти. Вместо того, чтобы выполнять переиндексирование для изменения индексного интервала, можно на время чтения `MapFile` загрузить в память только часть ключей индекса — для этого следует задать свойство `io.map.index.skip`. Обычно это свойство равно 0 (ключи индекса не пропускаются); значение 1 означает, что для каждого ключа в индексе пропускается один ключ (таким образом, в индекс попадает каждый второй ключ), значение 2 — что пропускаются 2 ключа (и в индекс попадает третья ключей), и так далее. Большие значения экономят память за счет времени поиска, так как в среднем приходится сканировать больше записей на диске.

Разновидности `MapFile`

Hadoop предлагает несколько вариаций на тему общего интерфейса `MapFile` «ключ-значение»:

- `SetFile` — специализация `MapFile` для хранения набора ключей `Writable`. Ключи должны добавляться в порядке сортировки.
- `ArrayFile` — разновидность `MapFile`, в которой ключ является целым числом, представляющим индекс элемента в массиве, а значение реализует `Writable`.
- `BloomMapFile` — разновидность `MapFile` с быстрой версией метода `get()`, особенно для разреженных файлов. Реализация использует динамический фильтр Блума для проверки присутствия заданного ключа. Проверка выполняется очень быстро, потому что данные находятся в памяти, однако существует ненулевая вероятность ложноположительных срабатываний.

Настройка фильтра осуществляется при помощи двух параметров: `io.mapfile.bloom.size` определяет (приблизительное) число записей данных в структуре (по умолчанию 1 048 576), а `io.mapfile.bloom.error.rate` — желательную максимальную вероятность ошибки (по умолчанию 0, 005, то есть 0,5%).

Преобразование `SequenceFile` в `MapFile`

`MapFile` может рассматриваться как индексированный и отсортированный контейнер `SequenceFile`. Соответственно, на практике часто встречается задача

преобразования `SequenceFile` в `MapFile`. Сортировка `SequenceFile` уже рассматривалась в разделе «Сортировка и слияние `SequenceFile`» на с. 193, так что сейчас мы рассмотрим построение индекса для `SequenceFile`. В листинге 4.17 используется статический вспомогательный метод `fix()` класса `MapFile`, который заново строит индекс для `MapFile`.

Листинг 4.17. Построение индекса для `MapFile`

```
public class MapFileFixer {
    public static void main(String[] args) throws Exception {
        String mapUri = args[0];

        Configuration conf = new Configuration();

        FileSystem fs = FileSystem.get(URI.create(mapUri), conf);
        Path map = new Path(mapUri);
        Path mData = new Path(map, MapFile.DATA_FILE_NAME);

        // Получение типов ключей и значений из последовательного файла данных
        SequenceFile.Reader reader = new SequenceFile.Reader(fs, mData, conf);
        Class keyClass = reader.getKeyClass();
        Class valueClass = reader.getValueClass();
        reader.close();
        // Создание индексного файла
        long entries = MapFile.fix(fs, map, keyClass, valueClass, false, conf);
        System.out.printf("Created MapFile %s with %d entries\n", map, entries);
    }
}
```

Метод `fix()` обычно используется для восстановления поврежденных индексов — но он строит индекс заново, а это именно то, что нам нужно. Действуйте по следующей схеме:

1. Отсортируйте последовательный файл `numbers.seq` в новый каталог `number.map`, содержимое которого образует `MapFile` (если последовательный файл уже отсортирован, пропустите этот шаг; вместо этого скопируйте его в файл `number.map/data` и переходите к шагу 3):

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort -r 1 \
-inFormat org.apache.hadoop.mapred.SequenceFileInputFormat \
-outFormat org.apache.hadoop.mapred.SequenceFileOutputFormat \
-outKey org.apache.hadoop.io.IntWritable \
-outValue org.apache.hadoop.io.Text \
numbers.seq numbers.map
```

2. Переименуйте вывод MapReduce в файл *data*:

```
% hadoop fs -mv numbers.map/part-00000 numbers.map/data
```

3. Создайте индексный файл:

```
% hadoop MapFileFixer numbers.map
Created MapFile numbers.map with 100 entries
```

MapFile *numbers.map* существует и готов к использованию.

5

Разработка приложений MapReduce

В главе 2 была представлена модель MapReduce. В этой главе рассматриваются практические аспекты разработки приложений MapReduce в среде Hadoop.

Программы MapReduce пишутся по определенной схеме. Сначала вы пишете функции отображения и свертки — в идеале вместе с модульными тестами, которые проверяют, работают ли функции так, как ожидается. Затем пишется управляющая программа для запуска задания, которая может запускаться из интегрированной среды разработки (IDE) с небольшим подмножеством данных для проверки работоспособности. Если запуск завершится неудачей, вы используете отладчик своей IDE для поиска источника ошибки. Располагая полной информацией, вы дорабатываете модульные тесты и функции отображения/свертки, добиваясь правильной обработки этих входных данных.

Когда программа заработает с малым набором данных так, как ожидалось, можно «выпускать» ее в кластер. Вероятно, при запуске для полного набора данных обнаружатся новые проблемы, которые исправляются доработкой тестов и функций отображения/свертки. Отладка программ в кластере — непростая задача, поэтому мы рассмотрим некоторые стандартные приемы, упрощающие ее.

Когда программа заработает, стоит заняться оптимизацией — сначала проведя некоторые стандартные проверки для ускорения работы программ MapReduce, затем посредством профилирования задач. Профилирование распределенных программ

тоже создает массу проблем, но Hadoop предоставляет точки входа, упрощающие этот процесс.

Прежде чем браться за написание программ MapReduce, необходимо подготовить и настроить среду разработки. А для этого стоит разобраться в том, как в Hadoop происходит настройка конфигурации.

API конфигурации

Для настройки компонентов Hadoop использует собственный интерфейс конфигурации. Экземпляр класса `Configuration` (из пакета `org.apache.hadoop.conf`) представляет коллекцию конфигурационных свойств и их значений. Имя каждого свойства задается типом `String`, а значение может относиться к одному из нескольких типов, включая примитивы Java (такие, как `boolean`, `int`, `long` и `float`), а также ряд других полезных типов: `String`, `Class`, `java.io.File` и коллекции `String`.

Конфигурационные свойства читаются из ресурсов — XML-файлов с простой структурой для определения пар «имя-значение» (листинг 5.1).

Листинг 5.1. Простой конфигурационный файл configuration-1.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>color</name>
    <value>yellow</value>
    <description>Color</description>
  </property>

  <property>
    <name>size</name>
    <value>10</value>
    <description>Size</description>
  </property>

  <property>
    <name>weight</name>
    <value>heavy</value>
    <final>true</final>
    <description>Weight</description>
  </property>

  <property>
```

продолжение ↗

Листинг 5.1 (продолжение)

```
<name>size-weight</name>
<value>${size},${weight}</value>
<description>Size and weight</description>
</property>
</configuration>
```

Если конфигурация хранится в файле с именем *configuration-1.xml*, то для обращения к ее свойствам может использоваться фрагмент кода следующего вида:

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
assertThat(conf.get("color"), is("yellow"));
assertThat(conf.getInt("size", 0), is(10));
assertThat(conf.get("breadth", "wide"), is("wide"));
```

Обратите внимание на пару моментов: информация о типах не хранится в XML-файле; вместо этого тип свойства интерпретируется во время чтения. Кроме того, методы *get()* позволяют задать значение по умолчанию, которое используется при отсутствии свойства в XML-файле (как в случае со свойством *breadth* из приведенного примера).

Объединение ресурсов

Если для определения конфигурации используется несколько ресурсов, все становится намного интереснее. Множественные ресурсы используются в Hadoop для отделения свойств системы по умолчанию, определяемых в файле *core-default.xml*, от специализированных переопределений свойств в файле *core-site.xml*. Файл в листинге 5.2 определяет свойства *size* и *weight*.

Листинг 5.2. Второй конфигурационный файл *configuration-2.xml*

```
<?xml version="1.0"?>
<configuration>
<property>
    <name>size</name>
    <value>12</value>
</property>

<property>
    <name>weight</name>
    <value>light</value>
</property>
</configuration>
```

Ресурсы добавляются в Configuration по порядку:

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
conf.addResource("configuration-2.xml");
```

Свойства, добавляемые позднее, заменяют более ранние определения. Таким образом, свойство `size` получает значение из второго файла `configuration-2.xml`:

```
assertThat(conf.getInt("size", 0), is(12));
```

Однако свойства с пометкой `final` не могут заменяться в последующих определениях. Так, свойство `weight` в первом конфигурационном файле имеет пометку `final`, поэтому попытка его переопределения во втором файле ни к чему не приводит и свойство сохраняет значение из первого файла:

```
assertThat(conf.get("weight"), is("heavy"));
```

Попытка переопределения `final`-свойств обычно свидетельствует об ошибке конфигурации, а в журнале сохраняется предупреждение, упрощающее диагностику. Администраторы снабжают свойства пометкой `final`, чтобы предотвратить переопределение свойств пользователями в конфигурационных файлах на стороне клиента или в параметрах отправки заданий.

Расширение переменных

В определениях свойств конфигурации могут использоваться другие свойства (а также системные свойства). Например, свойство `size-weight` в первом файле определяется в виде `${size}, ${weight}`, после чего внутренние свойства расширяются значениями, найденными в конфигурации:

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

Системные свойства обладают более высоким приоритетом, чем свойства, определенные в ресурсных файлах:

```
System.setProperty("size", "14");
assertThat(conf.get("size-weight"), is("14,heavy"));
```

Эта возможность полезна для переопределения свойств в командной строке при помощи аргументов JVM `-Dproperty=значение`.

Обратите внимание: свойства конфигурации могут определяться с использованием системных свойств, но, если только системные свойства не переопределяются с использованием свойств конфигурации, они остаются недоступными в API конфигурации, поэтому:

```
System.setProperty("length", "2");
assertThat(conf.get("length"), is((String) null));
```

Настройка среды разработки

Прежде всего создайте проект, чтобы вы могли строить программы MapReduce и запускать их в локальном (автономном) режиме из командной строки или среды разработки. РОМ-файл Maven из листинга 5.3 содержит зависимости, необходимые для построения и тестирования программ MapReduce.

Листинг 5.3. РОМ-файл Maven для построения и тестирования приложения MapReduce

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.hadoopbook</groupId>
  <artifactId>hadoop-book-mr-dev</artifactId>
  <version>3.0</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
  <dependencies>
    <!-- Главный артефакт Hadoop -->
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-core</artifactId>
      <version>1.0.0</version>
    </dependency>
    <!-- Артефакты модульного тестирования -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-all</artifactId>
      <version>1.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.mrunit</groupId>
```

```
<artifactId>mruni</artifactId>
<version>0.8.0-incubating</version>
<scope>test</scope>
</dependency>
<!-- Тестовые артефакты Hadoop для запуска мини-кластеров -->
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-test</artifactId>
    <version>1.0.0</version>
    <scope>test</scope>
</dependency>
<!-- Отсутствующая зависимость для запуска мини-кластеров -->
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.8</version>
    <scope>test</scope>
</dependency>
</dependencies>
<build>
    <finalName>hadoop-examples</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.3.2</version>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <outputDirectory>${basedir}</outputDirectory>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

В этом РОМ-файле интерес представляет секция зависимостей. (Также можно использовать другое средство сборки — например, Gradle или Ant с Ivy; важно,

чтобы при этом использовался тот набор зависимостей, который определен здесь.) Для построения заданий MapReduce необходима только зависимость `hadoop-core`, содержащая все классы Hadoop. Для выполнения модульных тестов мы используем `junit`, а также пару вспомогательных библиотек: `hamcrest-all` предоставляет полезные средства для написания тестовых проверок условий, а `mrunit` используется для написания тестов MapReduce. Библиотека `hadoop-test` содержит «мини»-кластеры, удобные для тестирования кластеров Hadoop, работающих в одной виртуальной машине Java (также мы подключаем `jersey-core`, потому что эта зависимость отсутствует в РОМ-файле Hadoop).



В выпусках после 1.x упаковка JAR изменилась, поэтому, к сожалению, простая замена номера версии в зависимости `hadoop-core` не решит проблемы. В примерах кода на сайте книги содержатся актуальные объявления зависимости для разных версий Hadoop.

Многие среды разработки умеют напрямую читать РОМ-файлы Maven, так что вы можете просто указать каталог, содержащий файл `pom.xml`, и приступить к написанию кода¹. Также можно сгенерировать конфигурационные файлы для вашей IDE при помощи Maven. Например, следующая команда создает файлы для импортирования проекта в Eclipse:

```
% mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

Управление конфигурацией

Во время разработки приложений Hadoop разработчик часто переключается между запуском приложения в локальном режиме и в кластере. Более того, вы можете работать с несколькими кластерами или же создать локальный псевдораспределенный кластер, который будет использоваться для тестирования («псевдораспределенным» называется кластер, все демоны которого выполняются на локальном компьютере).

Например, для этого можно создать конфигурационные файлы Hadoop с настройками подключения для каждого кластера, в котором запускается программа, и указывать нужный файл при запуске приложений или инструментов Hadoop. Такие файлы желательно хранить вне установочного дерева каталогов Hadoop, так как

¹ Karmasphere также предоставляет плагины Eclipse и NetBeans для разработки и запуска заданий MapReduce и просмотра кластеров Hadoop.

это позволяет легко переключаться между версиями Hadoop без дублирования или потери настроек.

Будем считать, что существует каталог с именем *conf*, в котором находятся три конфигурационных файла: *hadoop-local.xml*, *hadoop-localhost.xml* и *hadoop-cluster.xml* (файлы входят в код примеров книги). Учтите, что в именах файлов нет ничего особенного; это не более чем удобная схема упаковки параметров конфигурации.

Файл *hadoop-local.xml* содержит конфигурацию Hadoop для файловой системы и трекера заданий по умолчанию:

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>file:///</value>
    </property>

    <property>
        <name>mapred.job.tracker</name>
        <value>local</value>
    </property>

</configuration>
```

Настройки *hadoop-localhost.xml* определяют узел имен и трекер заданий, выполняемые на локальном хосте:

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost/</value>
    </property>

    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:8021</value>
    </property>

</configuration>
```

Наконец, в *hadoop-cluster.xml* хранится подробная информация об адресах узла имен и трекера заданий кластера. На практике в имя файла обычно включается имя кластера (вместо «cluster» в приведенном примере):

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://namenode/</value>
  </property>

  <property>
    <name>mapred.job.tracker</name>
    <value>jobtracker:8021</value>
  </property>

</configuration>
```

При необходимости в эти файлы можно добавить другие свойства конфигурации. Например, если вы хотите задать свое имя пользователя Hadoop для конкретного кластера, включите его в соответствующий файл.

ИДЕНТИФИКАЦИЯ ПОЛЬЗОВАТЕЛЯ

Идентификационные данные пользователя, используемые Hadoop для определения разрешений в HDFS, определяются по результатам выполнения команды whoami в клиентской системе. Аналогичным образом имена групп вычисляются по результатам выполнения команды groups.

Но если ваши идентификационные данные Hadoop отличаются от имени учетной записи пользователя на клиентской машине, вы можете явно задать имена пользователя и групп Hadoop в свойстве hadoop.job.ugi. Имена пользователя и групп задаются в виде списка строк, разделенного запятыми (например, список preston,directors,inventors задает имя пользователя preston и имена групп directors и inventors).

Чтобы задать идентификационные данные пользователя, с которыми работает веб-интерфейс HDFS, задайте свойство dfs.web.ugi в том же синтаксисе. По умолчанию используется значение webuser,webgroup, не дающее привилегий суперпользователя, так что системные файлы недоступны через веб-интерфейс.

Учтите, что по умолчанию аутентификация в системе не применяется. О том, как использовать аутентификацию Kerberos в Hadoop, рассказано в разделе «Безопасность» на с. 419.

При такой настройке вы можете легко выбрать любую конфигурацию при помощи ключа командной строки `-conf`. Например, следующая команда выводит содержимое каталога на сервере HDFS, работающем в псевдораспределенном режиме на локальном хосте:

```
% hadoop fs -conf conf/hadoop-localhost.xml -ls .
Found 2 items
drwxr-xr-x  - tom supergroup          0 2009-04-08 10:32 /user/tom/input
drwxr-xr-x  - tom supergroup          0 2009-04-08 13:09 /user/tom/output
```

Если опустить параметр `-conf`, будет выбрана конфигурация Hadoop в подкаталоге `conf` каталога `$HADOOP_INSTALL`. В зависимости от настройки это может быть автономная конфигурация или псевдораспределенный кластер.

Инструменты, поставляемые в Hadoop, поддерживают параметр `-conf`, но интерфейс `Tool` позволяет легко включить его поддержку в ваши собственные программы (например, программы запуска заданий MapReduce).

GenericOptionsParser, Tool и ToolRunner

В поставку Hadoop входят вспомогательные классы, упрощающие запуск заданий из командной строки. Класс `GenericOptionsParser` интерпретирует часто используемые параметры командной строки Hadoop и устанавливает их для объекта `Configuration` вашего приложения. Обычно `GenericOptionsParser` не используется напрямую — удобнее реализовать интерфейс `Tool` и запустить приложение при помощи класса `ToolRunner`, использующего `GenericOptionsParser` в своей внутренней реализации:

```
public interface Tool extends Configurable {
    int run(String [] args) throws Exception;
}
```

В листинге 5.4 приведена очень простая реализация `Tool`, которая выводит ключи и значения всех свойств из объекта `Configuration`.

Листинг 5.4. Простая реализация `Tool` для вывода свойств объекта `Configuration`

```
public class ConfigurationPrinter extends Configured implements Tool {

    static {
        Configuration.addDefaultResource("hdfs-default.xml");
        Configuration.addDefaultResource("hdfs-site.xml");
        Configuration.addDefaultResource("mapred-default.xml");
```

продолжение ↗

Листинг 5.4 (продолжение)

```

        Configuration.addDefaultResource("mapred-site.xml");
    }

@Override
public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    for (Entry<String, String> entry: conf) {
        System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
    }
    return 0;
}

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

```

Мы объявляем `ConfigurationPrinter` субклассом `Configured`, реализующим интерфейс `Configurable`. Все реализации `Tool` должны реализовать интерфейс `Configurable` (так как `Tool` расширяет его), и субклассирование `Configured` часто оказывается самым простым решением этой задачи. Метод `run()` получает объект `Configuration` с использованием метода `getConf()` интерфейса `Configurable`, а затем перебирает его свойства, направляя их в стандартный вывод.

Блок `static` гарантирует, что в дополнение к базовым конфигурациям (о которых `Configuration` уже знает) также будут выбраны конфигурации HDFS и MapReduce. Метод `main()` класса `ConfigurationPrinter` не вызывает свой метод `run()` напрямую. Вместо этого он вызывает статический метод `run()` класса `ToolRunner`, который создает объект `Configuration` для `Tool` перед тем, как вызывать его метод `run()`. `ToolRunner` также использует `GenericOptionsParser` для выбора стандартных параметров, указанных в командной строке, и их установки для экземпляра `Configuration`. Чтобы увидеть, как устанавливаются свойства из файла `conf/hadoop-localhost.xml`, выполните следующую команду:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop ConfigurationPrinter -conf conf/hadoop-localhost.xml \
| grep mapred.job.tracker=
mapred.job.tracker=localhost:8021
```

КАКИЕ СВОЙСТВА МОЖНО УСТАНАВЛИВАТЬ?

ConfigurationPrinter – полезный инструмент для проверки текущих значений свойств в вашей среде.

Чтобы просмотреть значения по умолчанию для всех открытых свойств Hadoop, поищите в каталоге docs своей установки Hadoop HTML-файлы с именами core-default.html, hdfs-default.html и mapred-default.html. В описании каждого свойства объясняется, для чего оно нужно и какие значения ему могут присваиваться.

Документацию по настройкам по умолчанию можно найти в Интернете. УLR-адреса имеют вид <http://hadoop.apache.org/common/docs/r<версия>/<компонент>-default.html>; например, настройки HDFS по умолчанию для версии 1.0.0 доступны по адресу <http://hadoop.apache.org/common/docs/r1.0.0/hdfs-default.html>.

Учтите, что изменение некоторых свойств в клиентской конфигурации ни к чему не приводит. Например, если вы при отправке задания зададите свойство mapred.tasktracker.map.tasks.maximum, ожидая, что это приведет к изменению количества слотов задач для трекеров задач, обслуживающих ваше задание, вы будете разочарованы, потому что это свойство работает только при установке в файле mapred-site.xml трекера задач. Как правило, компонент, в котором может задаваться свойство, можно определить по его имени; раз mapred.tasktracker.map.tasks.maximum начинается с mapred.tasktracker, это подсказывает, что задавать его можно только в демоне tasktracker. Впрочем, у данного правила существуют свои исключения, так что в некоторых случаях приходится действовать методом проб и ошибок и даже читать исходный код.

GenericOptionsParser также позволяет задавать отдельные свойства, например:

```
% hadoop ConfigurationPrinter -D color=yellow | grep color
color=yellow
```

Параметр *-D* задает свойству конфигурации с ключом *color* значение *yellow*. Параметры, заданные при помощи *-D*, обладают более высоким приоритетом, чем свойства из конфигурационных файлов. Этот механизм очень полезен, потому что вы можете поместить значения по умолчанию в конфигурационных файлах, а потом переопределять их в *-D* по мере необходимости. Типичный пример – назначение

количества задач свертки для задания MapReduce в конструкции `-D mapred.reduce.tasks=n`. Заданное значение заменяет количество, назначенное для кластера или заданное в конфигурационных файлах на стороне клиента.

Другие параметры, поддерживаемые `GenericOptionsParser` и `ToolRunner`, перечислены в табл. 5.1. Подробнее о конфигурационном API Hadoop рассказано в разделе «API конфигурации», с. 203.



Не пытайтесь задание свойств Hadoop с использованием параметра `-D` свойство=значение для `GenericOptionsParser` (и `ToolRunner`) с заданием системных свойств виртуальной машины Java (JVM) с использованием параметра `-D` свойство=значение команды `java`. Синтаксис системных свойств виртуальной машины Java не допускает наличия пробелов между `D` и именем свойства, тогда как `GenericOptionsParser` пробелы разрешает.

Системные свойства JVM могут быть получены от класса `java.lang.System`, тогда как свойства Hadoop доступны только из объекта `Configuration`. Таким образом, следующая команда не выведет ничего, даже если системное свойство `color` было задано (через `HADOOP_OPTS`), потому что класс `System` не используется классом `ConfigurationPrinter`:

```
% HADOOP_OPTS='-Dcolor=yellow' \
hadoop ConfigurationPrinter | grep color
```

Если вы хотите иметь возможность задавать конфигурацию через системные свойства, продублируйте нужные системные свойства в конфигурационном файле. За дополнительной информацией обращайтесь к разделу «Расширение переменных» на с. 205.

Таблица 5.1. Параметры `GenericOptionsParser` и `ToolRunner`

Параметр	Описание
<code>-D</code> свойство=значение	Задает свойству конфигурации Hadoop указанное значение. При этом переопределяются все свойства по умолчанию и свойства сайтов в конфигурации, а также свойства, заданные при помощи параметра <code>-conf</code>
<code>-conf</code> имя_файла ...	Включает заданные файлы в список ресурсов в конфигурации. Это удобный способ задания свойств сайтов и одновременно задания нескольких свойств
<code>-fs</code> uri	Связывает файловую систему по умолчанию с заданным URI; сокращенная форма записи для <code>-D fs.default.name=uri</code>
<code>-jt</code> хост:порт	Связывает трекер заданий с указанным хостом и портом; сокращенная форма записи для <code>-D mapred.job.tracker=хост:порт</code>

Параметр	Описание
-files файл1,файл2,...	Копирует заданные файлы из локальной файловой системы (или любой файловой системы, если задана схема) в общую файловую систему, используемую трекером заданий (обычно HDFS), и открывает доступ к ним программам MapReduce в рабочем каталоге задачи. (Использование механизма распределенного кэширования для копирования файлов на машины трекеров задач описано в разделе «Распределенный кэш», с. 375)
-archives архив1,архив2,...	Копирует заданные архивы из локальной файловой системы (или любой файловой системы, если задана схема) в общую файловую систему, используемую трекером заданий (обычно HDFS), распаковывает и открывает доступ к ним программам MapReduce в рабочем каталоге задачи
-libjars jar1,jar2,...	Копирует заданные JAR-файлы из локальной файловой системы (или любой файловой системы, если задана схема) в общую файловую систему, используемую трекером заданий (обычно HDFS), и добавляет их в путь к классам задачи MapReduce. Это удобный способ передачи файлов JAR, от которых зависит задание

Написание модульных тестов с MRUnit

Функции отображения и свертки MapReduce легко тестируются в изолированном режиме, что объясняется самим их функциональным стилем. MRUnit (<http://incubator.apache.org/mrunit/>) – библиотека тестирования, которая позволяет легко передать нужные входные данные функции отображения или свертки и проверить, возвращают ли они ожидаемый результат. MRUnit используется в сочетании со стандартной средой выполнения тестов (такой, как JUnit), так что запуск тестов для заданий MapReduce может стать нормальной частью среды разработки. Например, все тесты, описанные здесь, могут запускаться из IDE по инструкциям, приведенным ранее в разделе «Настройка среды разработки», с. 206.

Функция отображения

Тест для функции отображения представлен в листинге 5.5.

Листинг 5.5. Модульный тест для MaxTemperatureMapper

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.MapDriver;
```

продолжение ↗

Листинг 5.5 (продолжение)

```

import org.junit.*;

public class MaxTemperatureMapperTest {

    @Test
    public void processesValidRecord() throws IOException, InterruptedException {
        Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382"
+
                // Год ^^^^
                "99999V0203201N00261220001CN9999999N9-00111+99999999999");
                // Температура ^^^^^
        new MapDriver<LongWritable, Text, Text, IntWritable>()
            .withMapper(new MaxTemperatureMapper())
            .withInputValue(value)
            .withOutput(new Text("1950"), new IntWritable(-11))
            .runTest();
    }
}

```

Идея теста очень проста: мы передаем запись с метеорологическими данными в качестве входных данных функции отображения, а потом проверяем, что на выходе функция выдает год и температуру.

При тестировании функции отображения мы используем класс `MapDriver` из `MRUnit`, в настройках которого указывается тестируемая функция (`MaxTemperatureMapper`), входное значение, ожидаемый выходной ключ (объект `Text`, представляющий 1950 год) и ожидаемое выходное значение (объект `IntWritable`, представляющий температуру -1.1°C), после чего вызывается метод `runTest()` для выполнения теста. Если функция отображения не выдаст ожидаемые выходные значения, `MRUnit` сообщает о том, что тест не прошел. Обратите внимание: входной ключ на задается, потому что наша функция отображения его игнорирует.

Продолжая действовать в стиле «разработки через тестирование», мы создаем реализацию `Mapper`, которая проходит тест (листинг 5.6). Так как в этой главе мы будем вносить изменения в классы, каждый из них помещается в отдельный пакет, помеченный номером версии. Например, `v1.MaxTemperatureMapper` является версией 1 класса `MaxTemperatureMapper`. Конечно, на практике эволюция классов должна происходить без смены пакета.

Листинг 5.6. Первая версия Mapper, проходящая тест `MaxTemperatureMapperTest`

```

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    @Override

```

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

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature = Integer.parseInt(line.substring(87, 92));
    context.write(new Text(year), new IntWritable(airTemperature));
}
```

Это очень простая реализация, которая извлекает поля года и температуры из строки и записывает их в `Context`. Добавим тест для отсутствующих значений, которые в низкоуровневых данных представлены температурой +9999:

```
@Test
public void ignoresMissingTemperatureRecord() throws IOException,
    InterruptedException {
    Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
        // Год ^^^^
        "99999V0203201N00261220001CN9999999N9+99991+9999999999");
        // Температура ^^^^^^
    new MapDriver<LongWritable, Text, Text, IntWritable>()
        .withMapper(new MaxTemperatureMapper())
        .withInputValue(value)
        .runTest();
}
```

`MapDriver` может использоваться для проверок нуля, одной или нескольких выходных записей в соответствии с количеством вызовов `withOutput()`. Так как в нашем приложении записи с отсутствующими температурами должны отфильтровываться, этот тест подтверждает, что для этого конкретного входного значения выходные данные не генерируются.

В текущей реализации происходит ошибка с исключением `NumberFormatException`, так как `parseInt()` не умеет разбирать целые числа с начальным знаком +. Исправим реализацию для поддержки отсутствующих значений (версия 2):

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

    String line = value.toString();
    String year = line.substring(15, 19);
    String temp = line.substring(87, 92);
    if (!missing(temp)) {
        int airTemperature = Integer.parseInt(temp);
    }
}
```

продолжение ➔

```

        context.write(new Text(year), new IntWritable(airTemperature));
    }
}

private boolean missing(String temp) {
    return temp.equals("+9999");
}

```

Убедившись в том, что тест функции отображения проходит успешно, перейдем к функции свертки.

Функция свертки

Функция свертки должна находить максимальное значение для заданного ключа. Ниже приведен простой тест, использующий класс `ReduceDriver`:

```

@Test
public void returnsMaximumIntegerInValues() throws IOException,
    InterruptedException {
    new ReduceDriver<Text, IntWritable, Text, IntWritable>()
        .withReducer(new MaxTemperatureReducer())
        .withInputKey(new Text("1950"))
        .withInputValues(Arrays.asList(new IntWritable(10), new IntWritable(5)))
        .withOutput(new Text("1950"), new IntWritable(10))
        .runTest();
}

```

Мы строим список значений `IntWritable`, после чего убеждаемся в том, что `MaxTemperatureReducer` выбирает наибольшее из них. В листинге 5.7 приведена реализация `MaxTemperatureReducer`, которая проходит тест.

Листинг 5.7. Функция свертки для примера с максимальной температурой

```

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

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

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
    }
}

```

```
    }
    context.write(key, new IntWritable(maxValue));
}
}
```

Локальное выполнение с тестовыми данными

Итак, у нас есть функции отображения и свертки, работающие для контролируемых входных данных. Следующим шагом должно стать написание управляющей программы задания и ее запуск с тестовыми данными на машине разработки.

Локальный запуск задания

Интерфейс `Tool`, представленный ранее в этой главе, позволяет легко написать управляющую программу `MaxTemperatureDriver` для запуска задания MapReduce, определяющего максимальную температуру за год (листинг 5.8).

Листинг 5.8. Приложение для определения максимальной температуры

```
public class MaxTemperatureDriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s [generic options] <input> <output>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.err);
            return -1;
        }

        Job job = new Job(getConf(), "Max temperature");
        job.setJarByClass(getClass());

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

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setCombinerClass(MaxTemperatureReducer.class);
        job.setReducerClass(MaxTemperatureReducer.class);

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

продолжение ↗

Листинг 5.8 (продолжение)

```
job.setOutputValueClass(IntWritable.class);

return job.waitForCompletion(true) ? 0 : 1;
}

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

`MaxTemperatureDriver` реализует интерфейс `Tool`, поэтому мы сможем задать параметры, поддерживаемые `GenericOptionsParser`. Метод `run()` конструирует по данным конфигурации объект `Job`, который используется для запуска задания. Среди возможных параметров конфигурации задания мы задаем пути к входным и выходным файлам, классы отображения, свертки и комбинирования, а также выходные типы (входные типы определяются входным форматом; по умолчанию используется формат `TextInputFormat` с ключами `LongWritable` и значениями `Text`). Также полезно указать имя задания (`Max temperature`), чтобы вам было проще найти его в списке заданий во время выполнения и после завершения. По умолчанию в качестве имени задания используется имя файла JAR, которое обычно не отличается содержательностью.

Теперь мы можем запустить свое приложение для локальных файлов. В поставку Hadoop включается усеченная версия исполнительного ядра MapReduce для запуска заданий MapReduce на одной виртуальной машине Java. Локальный исполнитель заданий предназначен для тестирования. Его чрезвычайно удобно использовать в IDE, так как в отладчике он позволяет в пошаговом режиме выполнять код функций отображения и свертки.



Локальный исполнитель заданий предназначен только для простого тестирования программ MapReduce, поэтому он неизбежно отличается от полноценной реализации MapReduce. Главное отличие заключается в том, что он не позволяет выполнять более одной свертки (также поддерживаются случай нуля сверток). Обычно это не создает проблем, так как многие приложения работают с одной сверткой, хотя в кластере стоит выбрать большее их количество для использования преимуществ параллелизма. Учтите, что даже если задать количество сверток большим 1, локальный исполнитель проигнорирует настройку и использует одну свертку.

Возможно, в будущих версиях Hadoop это ограничение будет снято.

Локальный исполнитель заданий включается в настройках конфигурации. Обычно свойство `mapred.job.tracker` содержит пару `host:port`, определяющую адрес трекера заданий, но со специальным значением `local` (которое используется по умолчанию) задание будет выполняться во внутрипроцессном режиме без обращений к внешнему трекеру заданий.

В MapReduce 2 (см. «YARN (MapReduce 2)», с. 265) используется эквивалентное свойство `mapreduce.framework.name`, которому должно быть задано значение `local` (также действует по умолчанию).

Чтобы запустить управляющую программу в командной строке, введите

```
% mvn compile  
% export HADOOP_CLASSPATH=target/classes/  
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \  
  input/ncdc/micro output
```

Также можно воспользоваться параметрами `-fs` и `-jt`, предоставляемыми `GenericOptionsParser`:

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local input/ncdc/micro output
```

Команда выполняет `MaxTemperatureDriver` с использованием входных данных из локального каталога `input/ncdc/micro` и направляет выходные данные в локальный каталог `output`. Хотя значение параметра `-fs` указывает на использование локальной файловой системы (`file:///`), локальный исполнитель заданий будет нормально работать в любой файловой системе, включая HDFS (и это может быть удобно, если некоторые файлы находятся в HDFS).

При запуске программы происходит ошибка с выдачей исключения:

```
java.lang.NumberFormatException: For input string: "+0000"
```

Исправление ошибки в отображении

Исключение показывает, что метод отображения все еще не способен обрабатывать положительные температуры. (Если бы трассировка стека не дала достаточной информации для диагностики, тест можно было бы выполнить в локальном отладчике.) Ранее мы успешно организовали обработку отсутствующих данных температуры +9999, но не общего случая любой положительной температуры. С расширением логики отображения будет разумно выделить специальный класс, инкапсулирующий логику разбора (версия 3 приведена в листинге 5.9).

Листинг 5.9. Класс для разбора метеорологических данных в формате NCDC

```
public class NcdcRecordParser {

    private static final int MISSING_TEMPERATURE = 9999;

    private String year;
    private int airTemperature;
    private String quality;

    public void parse(String record) {
        year = record.substring(15, 19);
        String airTemperatureString;
        // Удаляем начальный плюс, несовместимый с parseInt
        if (record.charAt(87) == '+') {
            airTemperatureString = record.substring(88, 92);
        } else {
            airTemperatureString = record.substring(87, 92);
        }
        airTemperature = Integer.parseInt(airTemperatureString);
        quality = record.substring(92, 93);
    }

    public void parse(Text record) {
        parse(record.toString());
    }

    public boolean isValidTemperature() {
        return airTemperature != MISSING_TEMPERATURE && quality.matches("[01459]");
    }

    public String getYear() {
        return year;
    }
    public int getAirTemperature() {
        return airTemperature;
    }
}
```

Новая версия отображения намного проще (листинг 5.10). Она просто вызывает метод `parse()` класса `NcdcRecordParser`, который выделяет нужные поля из входной строки, проверяет найденную температуру методом `isValidTemperature()`, и если она действительна — получает год и температуру при помощи `get`-методов парсера. Также обратите внимание на проверку поля качества данных и отсутствующих температур в `isValidTemperature()` для исключения некорректных значений температуры.

Выделение разбора в отдельный класс имеет и другое преимущество: класс упрощает написание похожих отображений для сходных заданий без дублирования кода. Кроме того, у разработчика появляется возможность писать модульные тесты прямо для парсера, что делает тестирование более направленным.

Листинг 5.10. Класс отображения, использующий вспомогательный класс для разбора записей

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

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

        parser.parse(value);
        if (parser.isValidTemperature()) {
            context.write(new Text(parser.getYear()),
                new IntWritable(parser.getAirTemperature()));
        }
    }
}
```

С такими изменениями тест проходит успешно.

Тестирование управляемой программы

Кроме гибких возможностей настройки конфигурации, реализация приложением интерфейса `Tool` также упрощает его тестирование, потому что у разработчика появляется возможность внедрения произвольных экземпляров `Configuration`. Это позволит нам написать тест, использующий локальный исполнитель задач для запуска задания с известными входными данными, и убедиться в том, что результат совпадает с ожидаемым.

Есть два подхода к решению этой задачи. Во-первых, можно воспользоваться локальным исполнителем задач и запустить задание для тестового файла в локальной файловой системе. Код в листинге 5.11 дает представление о том, как это делается. Тест явно задает свойства `fs.default.name` и `mapred.job.tracker` для использования локальной файловой системы и локального исполнителя заданий. Затем он запускает `MaxTemperatureDriver` через интерфейс `Tool` для небольшого объема известных данных. В конце теста вызывается метод `checkOutput()`, который строка за строкой сравнивает фактический вывод с ожидаемым.

Листинг 5.11. Тест MaxTemperatureDriver, использующий локальный внутрипроцессный исполнитель заданий

```
@Test
public void test() throws Exception {
    Configuration conf = new Configuration();
    conf.set("fs.default.name", "file:///");  
conf.set("mapred.job.tracker", "local");

    Path input = new Path("input/ncdc/micro");
    Path output = new Path("output");
    FileSystem fs = FileSystem.getLocal(conf);
    fs.delete(output, true); // Удаление старых выходных данных

    MaxTemperatureDriver driver = new MaxTemperatureDriver();
    driver.setConf(conf);

    int exitCode = driver.run(new String[] {
        input.toString(), output.toString() });
    assertThat(exitCode, is(0));

    checkOutput(conf, output);
}
```

Второй способ тестирования управляющей программы основан на запуске ее с использованием «мини»-кластера. Hadoop содержит тестовые классы `MiniDFSCluster`, `MiniMRCluster` и `MiniYARNCluster`, позволяющие на программном уровне создавать внутрипроцессные кластеры. В отличие от локального исполнителя заданий, эти классы позволяют проводить тестирование для полных механизмов MapReduce и HDFS. Однако учтите, что трекеры задач в мини-кластерах запускают отдельные виртуальные машины Java для запуска задач, что может усложнить процесс отладки.

Мини-кластеры широко используются в собственных средствах автоматизации тестирования Hadoop, но они также могут использоваться для тестирования пользовательского кода. Абстрактный класс Hadoop `ClusterMapReduceTestCase` предоставляет базу для написания таких тестов; он обеспечивает все подробности запуска и остановки внутрипроцессных кластеров MapReduce и HDFS в методах `setUp()` и `tearDown()` и генерирует объект конфигурации, настроенный для работы с ними. Субклассам остается только заполнить данные в HDFS (возможно, копированием из локального файла), запустить задание MapReduce и проверить, соответствуют ли выходные данные ожидаемым. Пример кода содержится в классе `MaxTemperatureDriverMiniTest` в примерах, прилагаемых к книге.

Подобные тесты служат для выполнения регрессионного тестирования и формируют полезный набор граничных случаев входных данных и их ожидаемых

результатов. С появлением новых тестовых примеров вы можете просто включить их во входной файл и внести соответствующие изменения в файл ожидаемого вывода.

Запуск в кластере

Итак, наша программа заработала для небольшого тестового набора данных, и все готово к тому, чтобы опробовать ее с полным набором данных в кластере Hadoop. Как настроить полноценный распределенный кластер, рассказано в главе 9, хотя материал этого раздела также можно опробовать в псевдораспределенном кластере.

Упаковка задания

Локальный исполнитель заданий использует одну виртуальную машину Java для запуска задания. Если все классы, необходимые заданию, расположены в его пути к классам, программа будет работать.

В распределенной среде дело обстоит сложнее. Прежде всего, классы задания должны быть упакованы в JAR-файл задания для отправки кластеру. Hadoop находит JAR-файл задания автоматически, проводя поиск в пути к классам управляющей программы JAR-файла, содержащего класс, заданный в методе `setJarByClass()` (для `JobConf` или `Job`). Если же вы захотите явно задать JAR-файл по его пути, используйте метод `setJar()`.

JAR-файл задания удобно строится при помощи таких инструментов, как Ant или Maven. Например, следующая команда Maven создает в каталоге проекта JAR-файл с именем *hadoop-examples.jar*, содержащий все откомпилированные классы:

```
% mvn package -DskipTests.
```

Если один JAR-файл соответствует одному заданию, вы можете указать главный выполняемый класс в манифесте JAR-файла. Если главный класс не указан в манифесте, он должен быть задан в командной строке (как мы вскоре увидим при запуске задания).

Все зависимые JAR-файлы могут быть упакованы в подкаталог *lib* JAR-файла задания, хотя существуют и другие способы включения зависимостей, о которых мы поговорим позднее. Файлы ресурсов могут быть упакованы в подкаталог *classes*. (По аналогии с архивами веб-приложений Java — WAR-файлами, не считая того, что в данном случае JAR-файлы находятся в подкаталоге *WEB-INF/lib*, а классы — в подкаталоге *WEB-INF/classes* WAR-файла.)

Клиентский путь к классам

Клиентский путь к классам, задаваемый командой `hadoop jar <jar>`, состоит из:

- JAR-файла задания;
- всех JAR-файлов в каталоге *lib* JAR-файла задания и каталога *classes* (если он присутствует);
- пути к классам, определяемого переменной `HADOOP_CLASSPATH` (если она задана).

Кстати говоря, это объясняет, почему при запуске с использованием локального исполнителя заданий без JAR-файла задания (`hadoop ИМЯ_КЛАССА`) необходимо включать в `HADOOP_CLASSPATH` данные зависимых классов и библиотек.

Путь к классам задач

В кластере (в том числе и в псевдораспределенном) задачи отображения и свертки выполняются на разных виртуальных машинах, а их пути к классам не контролируются переменной `HADOOP_CLASSPATH`.

`HADOOP_CLASSPATH` относится к клиентским настройкам и задает путь к классам только для виртуальной машины Java управляющей программы, отправившей задание.

Вместе с тем пользовательский путь к классам задачи состоит из:

- JAR-файла задания;
- всех JAR-файлов в каталоге *lib* JAR-файла задания и каталога *classes* (если он присутствует);
- файлов, включенных в распределенный кэш с использованием параметра `-libjars` (табл. 5.1), метода `addFileToClassPath()` объекта `DistributedCache` (старый API) или `Job` (новый API).

Упаковка зависимостей

Существуют разные способы включения библиотечных зависимостей заданий:

- Распакуйте библиотеки и заново упакуйте их в JAR-файл задания.
- Упакуйте библиотеки в каталог *lib* JAR-файла задания.
- Храните библиотеки отдельно от JAR-файла задания. Включите их в клиентский путь к классам с использованием `HADOOP_CLASSPATH` и в путь к классам задач — с использованием `-libjars`.

Последний вариант, использующий распределенный кэш, является самым простым с точки зрения построения, потому что зависимости не нужно упаковывать заново в JAR-файл задания. Кроме того, распределенный кэш может сократить количество пересылок JAR-файлов по кластеру, так как файлы могут кэшироваться на узлах между задачами. (Распределенное кэширование более подробно рассматривается на с. 375).

Приоритеты путей к классам задач

Пользовательские JAR-файлы добавляются в конец как клиентского пути к классам, так и пути к классам задач, что в некоторых случаях может приводить к конфликтам зависимостей с встроенными библиотеками Hadoop, если Hadoop использует другую версию библиотеки, несовместимую с используемой в вашем коде. Иногда бывает нужно управлять порядком путей к классам задач, чтобы ваши классы выбирались раньше других. Чтобы заставить Hadoop поместить пользовательский путь к классам на первое место в порядке поиска, задайте переменной окружения `HADOOP_USER_CLASSPATH_FIRST` значение `true`. Для пути к классам задач задайте значение `true` свойству `mapreduce.task.classpath.first`.

Учтите, что установка этих параметров может изменить процесс загрузки классов для зависимостей инфраструктуры Hadoop (но только в вашем задании), что теоретически может привести к ошибкам отправки заданий или задач. Будьте осторожны при использовании этих параметров.

Запуск задания

Чтобы запустить задание, необходимо запустить управляющую программу с указанием кластера, в котором оно должно выполняться, при помощи параметра `-conf` (с таким же успехом можно использовать параметры `-fs` и `-jt`):

```
% unset HADOOP_CLASSPATH  
% hadoop jar hadoop-examples.jar v3.MaxTemperatureDriver \  
-conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```



Мы сбрасываем переменную окружения `HADOOP_CLASSPATH`, потому что задание не имеет сторонних зависимостей. Если оставить переменной значение `target/classes/` (из более ранних примеров этой главы), Hadoop не сможет найти JAR-файл задания, если класс `MaxTemperatureDriver` будет загружен из `target/classes`, а не из JAR, и задание не запустится.

Метод `waitForCompletion()` класса `Job` запускает задание и отслеживает ход его выполнения, выводя строку с описанием изменений состояния отображения и свертки в случае их изменения. Ниже приведен протокол выполнения (часть удалена для наглядности):

```
09/04/11 08:15:52 INFO mapred.FileInputFormat: Total input paths to process : 101
09/04/11 08:15:53 INFO mapred.JobClient: Running job: job_200904110811_0002
09/04/11 08:15:54 INFO mapred.JobClient: map 0% reduce 0%
09/04/11 08:16:06 INFO mapred.JobClient: map 28% reduce 0%
09/04/11 08:16:07 INFO mapred.JobClient: map 30% reduce 0%
...
09/04/11 08:21:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/11 08:21:38 INFO mapred.JobClient: Job complete: job_200904110811_0002
09/04/11 08:21:38 INFO mapred.JobClient: Counters: 19
09/04/11 08:21:38 INFO mapred.JobClient: Job Counters
09/04/11 08:21:38 INFO mapred.JobClient: Launched reduce tasks=32
09/04/11 08:21:38 INFO mapred.JobClient: Rack-local map tasks=82
09/04/11 08:21:38 INFO mapred.JobClient: Launched map tasks=127
09/04/11 08:21:38 INFO mapred.JobClient: Data-local map tasks=45
09/04/11 08:21:38 INFO mapred.JobClient: FileSystemCounters
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_READ=12667214
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_READ=33485841275
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_WRITTEN=989397
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=904
09/04/11 08:21:38 INFO mapred.JobClient: Map-Reduce Framework
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input groups=100
09/04/11 08:21:38 INFO mapred.JobClient: Combine output records=4489
09/04/11 08:21:38 INFO mapred.JobClient: Map input records=1209901509
09/04/11 08:21:38 INFO mapred.JobClient: Reduce shuffle bytes=19140
09/04/11 08:21:38 INFO mapred.JobClient: Reduce output records=100
09/04/11 08:21:38 INFO mapred.JobClient: Spilled Records=9481
09/04/11 08:21:38 INFO mapred.JobClient: Map output bytes=10282306995
09/04/11 08:21:38 INFO mapred.JobClient: Map input bytes=274600205558
09/04/11 08:21:38 INFO mapred.JobClient: Combine input records=1142482941
09/04/11 08:21:38 INFO mapred.JobClient: Map output records=1142478555
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input records=103
```

В протоколе можно найти много полезной информации. Перед запуском задания выводится его идентификатор; он используется для ссылок на задание (например, в файлах журналов) или при запросах информации командой `hadoop job`. При завершении задания выводится статистика, по которой можно убедиться в том, что задание отработало так, как ожидалось. Например, мы видим, что в этом задании было проанализировано около 275 Гбайт входных данных (`Map input bytes`) и прочитано около 34 Гбайт сжатых данных из HDFS (`HDFS_BYTES_READ`). Входные данные были разбиты на 101 файл, сжатый в формате gzip, так что проблем с разбиением не было.

ИДЕНТИФИКАТОРЫ ЗАДАНИЯ, ЗАДАЧИ И ПОПЫТКИ

Формат идентификатора задания состоит из времени начала работы трекера заданий (не самого задания) и счетчика, однозначно определяющего задание для данного экземпляра трекера заданий. Таким образом, задание с идентификатором:

`job_200904110811_0002`

является вторым (0002; нумерация заданий начинается с 1) заданием, запущенным в 08:11 11 апреля 2009 года. Счетчик дополняется начальными нулями, чтобы упростить сортировку идентификаторов — например, при выводе содержимого каталогов. Однако при достижении 10 000 счетчик не сбрасывается, а идентификаторы заданий становятся длиннее (и перестают хорошо сортироваться).

Задачи принадлежат заданиям, а их идентификаторы образуются заменой префикса job идентификатора задания префиксом task и добавлением суффикса, идентифицирующего задачу в задании. Например, задача

`task_200904110811_0002_m_000003`

является четвертой (000003, нумерация задач начинается с 0) задачей отображения (m) задания с идентификатором `job_200904110811_0002`. Идентификаторы задач создаются для заданий в момент инициализации, поэтому они не обязательно определяют порядок выполнения задач.

Задачи могут выполняться более одного раза из-за сбоев (см. «Сбой задачи», с. 272) или спекулятивного выполнения (см. «Спекулятивное выполнение», с. 288), поэтому для идентификации разных попыток выполнения задач трекер заданий присваивает им уникальные идентификаторы. Например, попытка

`attempt_200904110811_0002_m_000003_0`

является первой (0; нумерация попыток начинается с 0) попыткой выполнения задачи `task_200904110811_0002_m_000003`. Попытки выполнения задач создаются в процессе выполнения задания по мере надобности, а их порядок определяет последовательность их создания для работы трекеров задач.

Веб-интерфейс MapReduce

В комплект поставки Hadoop включен веб-интерфейс для просмотра информации о заданиях. В нем удобно следить за ходом выполнения запущенных заданий,

а также находить статистику и журналы после завершения заданий. Веб-интерфейс доступен по адресу <http://jobtracker-host:50030/>.

Страница трекера заданий

Внешний вид домашней страницы показан на рис. 5.1. В первом разделе приводится расширенная информация об установке Hadoop: номер версии, дата/время компилирования, текущее состояние трекера заданий (работает в нашем примере), дата/время запуска.

ip-10-250-110-47 Hadoop Map/Reduce Administration

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name):
Example: 'user smith 5300' will filter by 'smith' only in the user field and '5300' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	N/A

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	N/A

Failed Jobs

none

Local Logs

Log directory: Job Tracker History
Hadoop, 2009.

Рис. 5.1. Страница трекера заданий

Далее идет сводка по кластеру с данными о емкости и текущей загрузке кластера. В этом разделе выводится количество отображений и сверток, выполняемых в кластере в настоящее время, общее количество отправленных заданий, количество

доступных узлов трекеров задач и емкость кластера (в слотах отображений и сверток, доступных в кластере, — «Map Task Capacity» и «Reduce Task Capacity») и среднее количество свободных слотов на узел. Также приводится количество трекеров задач, находящихся в «черном списке» трекера заданий («черные списки» рассматриваются в разделе «Сбой трекера задач», с. 273).

Под сводкой находится раздел с информацией о выполняемом планировщике заданий (в нашем примере используется планировщик по умолчанию). Здесь можно просмотреть информацию об очередях заданий.

Еще ниже находятся разделы выполняемых, завершенных (успешно) и сбойных заданий. Каждый раздел представляет собой таблицу, в которой для каждого задания выводится идентификатор, имя владельца (заданное в конструкторе `Job` или методе `setJobName()`; в обоих случаях во внутренней реализации задается свойство `mapred.job.name`) и информация о ходе выполнения.

Наконец, в конце страницы находятся ссылки на журналы трекера заданий и историю трекера заданий с информацией обо всех заданиях, запускавшихся трекером. В главном представлении выводится информация о 100 заданиях (значение определяется свойством `mapred.jobtracker.completeuserjobs.maximum`), а лишние задания отправляются на страницу истории. Следует учесть, что Hadoop хранит долгосрочную историю с информацией от предыдущих запусков трекера заданий.

ИСТОРИЯ ЗАДАНИЙ

«Историей заданий» называется информация о событиях и конфигурации завершенных заданий. История сохраняется независимо от того, успешно ли завершилось задание.

Файлы истории заданий хранятся в локальной файловой системе трекера заданий в подкаталоге `history` каталога `logs`. Для ее хранения можно выбрать произвольную файловую систему Hadoop, задав значение свойства `hadoop.job.history.location`. Файлы истории трекера заданий хранятся 30 дней, после чего удаляются системой.

Вторая копия сохраняется для пользователя в подкаталоге `_logs/history` выходного каталога задания. Ее местонахождение можно переопределить при помощи свойства `hadoop.job.history.user.location`. Если задать специальное значение `none`, пользовательская информация о заданиях сохраняться не будет (останется только централизованная история заданий). Пользовательские файлы истории заданий никогда не удаляются системой.

В журнал истории включаются события заданий, задач и попыток. Информация хранится в виде текстового файла. История конкретного задания может просматриваться через веб-интерфейс или в командной строке с использованием `hadoop.job-history` (с указанием выходного каталога задания).

Страница задания

Если щелкнуть на идентификаторе задания, открывается страница задания, изображенная на рис. 5.2. В верхней части страницы выводится сводная информация о задании с основными сведениями (владелец задания, название, время выполнения). Ссылка в строке `Job file` открывает консолидированный конфигурационный файл задания со всеми свойствами и значениями, действовавшими на момент запуска. Если вы не уверены в том, какое значение было задано тому или иному свойству, достаточно просмотреть этот файл.

На периодически обновляемой странице задания можно следить за ходом его выполнения. Под сводкой находится таблица с информацией о ходе отображений и сверток. В столбце «Num Tasks» выводится общее количество задач отображения и свертки для этого задания (по строке для каждой задачи). В других столбцах выводится состояние этих задач: «Pending» (ожидает выполнения), «Running» (выполняется), «Complete» (успешно выполнена) или «Killed» (при выполнении задачи произошел сбой). В последнем столбце выводится общее количество сбойных попыток выполнения для всех задач отображения или свертки по заданию (попытки могут помечаться как сбойные, если они являются дубликатами в схеме спекулятивного выполнения, при потере работоспособности трекера задач, на котором они выполняются, или при уничтожении их пользователем). За дополнительной информацией обращайтесь к разделу «Сбой задачи», с. 272.

Далее на странице задания находятся диаграммы завершения для каждой задачи, на которых ход выполнения представлен в графическом виде. Диаграмма завершения сверток разделена на три фазы задачи свертки: копирование (перенос выходных данных отображений на трекер задач свертки), сортировка (слияние входных данных свертки) и свертка (выполнение функции свертки для получения окончательного вывода). Фазы более подробно описаны в разделе «Тасовка и сортировка», с. 279.

В середине страницы расположена таблица счетчиков заданий (job counters). Счетчики динамически обновляются в ходе выполнения задания и предоставляют другую полезную возможность для отслеживания хода и общего состояния задания. Дополнительная информация о счетчиках приведена в разделе «Встроенные счетчики», с. 338.

Получение результатов

Результаты выполненного задания можно получить несколькими разными способами. Каждая из задач свертки генерирует один выходной файл, соответственно, в каталоге `max-temp` создаются 30 частичных файлов с именами от `part-r-00000` до `part-r-00029`.

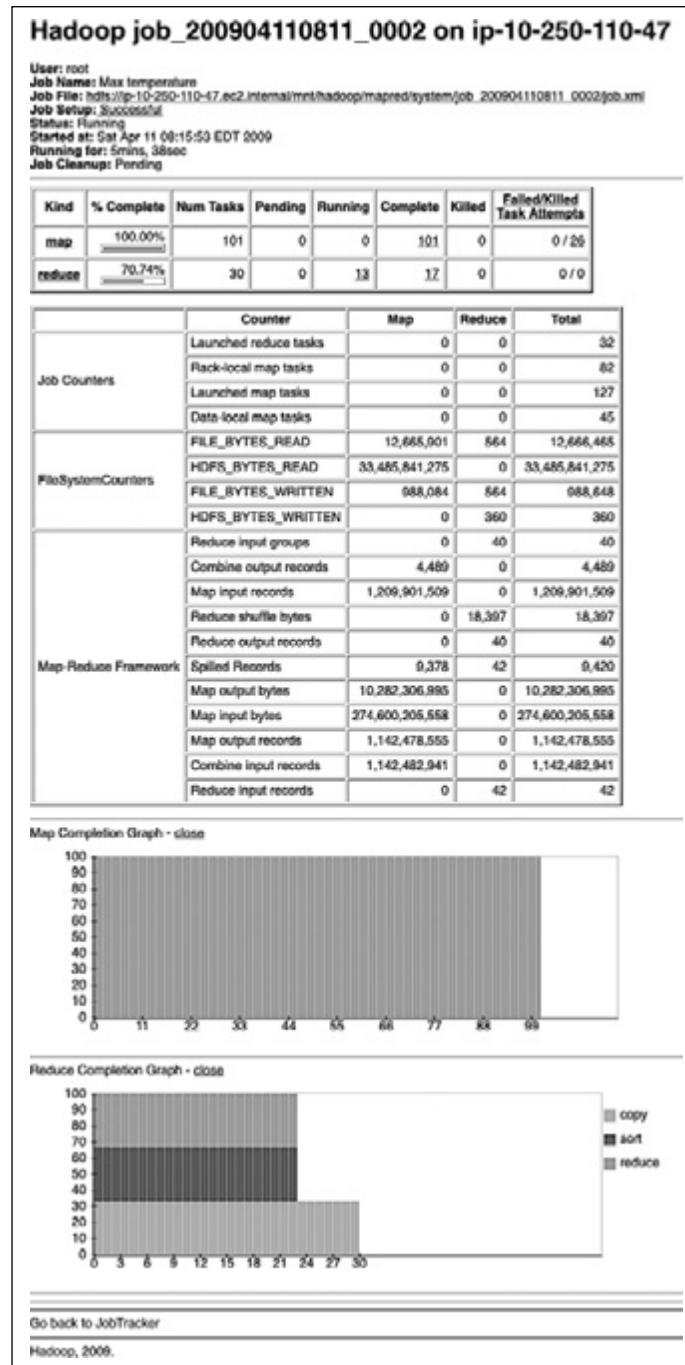


Рис. 5.2. Страница задания



Как подсказывает само название, частичные файлы удобно рассматривать как составные части «файла» `max-temp`.

Если выходные данные занимают много места (в нашем случае это не так), важно разделить их на несколько частей, чтобы несколько задач свертки могли работать параллельно. Обычно файлы, хранящиеся в разделенной форме, использовать достаточно легко — например, в качестве входных данных для другого задания MapReduce. В таких случаях структура разделов используется для выполнения соединений на стороне отображения (см. «Соединения на стороне отображения», с. 369) или поиска в MapFile (см. «Поиск в MapFile», с. 353).

Наше задание генерирует очень небольшой объем выходных данных, которые будет удобно скопировать из HDFS на машину разработки. В этом нам поможет параметр `-getmerge` команды `hadoop fs` — он берет все файлы в заданном каталоге и сливает их в один файл в локальной файловой системе:

```
% hadoop fs -getmerge max-temp max-temp-local  
% sort max-temp-local | tail  
1991      607  
1992      605  
1993      567  
1994      568  
1995      567  
1996      561  
1997      565  
1998      568  
1999      568  
2000      558
```

Мы сортируем результаты, так как выходные разделы свертки не упорядочены (из-за хеш-функции разделения). Данные, полученные от MapReduce, очень часто подвергаются завершающей обработке перед тем, как передаваться аналитическим инструментам — таким, как R, электронные таблицы и даже реляционные базы данных.

Другой способ получения выходных данных небольшого объема — использование параметра `-cat` для вывода выходных файлов на консоль:

```
% hadoop fs -cat max-temp/*
```

При внимательном рассмотрении мы видим, что некоторые результаты выглядят неправдоподобно. Например, максимальная температура за 1951 год (не показана в примере) равна 590°C! Как узнать, из-за чего это происходит? Из-за некорректных входных данных или из-за ошибки в программе?

Отладка задания

Конечно, прошедший проверку временем способ отладки программ с использованием отладочной печати возможен и в Hadoop. Однако при этом задача усложняется: если программы работают на десятках, сотнях и тысячах узлов, как найти и проанализировать вывод отладочных команд, разбросанных по этим узлам? В нашей конкретной ситуации, в которой мы ищем нетипичный случай, можно воспользоваться отладочной командой для вывода сообщения в стандартный поток ошибок, а также обновлением сообщения о состоянии задачи. Как вы сейчас увидите, в веб-интерфейсе это делается просто.

Мы также создадим пользовательский счетчик для подсчета общего количества записей с недостоверными температурами во всем наборе данных. Он даст нам полезную информацию для решения этой проблемы. Если проблема будет встречаться часто, возможно, нам следует получше разобраться в ситуации и в том, как извлекать температуру в подобных случаях (вместо простого удаления записи). В процессе отладки задания вы всегда должны спрашивать себя, нельзя ли использовать счетчик для получения необходимой информации. Даже если без журнала или сообщения о состоянии не обойтись, счетчик может пригодиться для оценки масштаба проблемы. (За дополнительной информацией о счетчиках обращайтесь к разделу «Счетчики», с. 337).

Если объем журнальных данных, генерируемых в процессе отладки, окажется слишком большим, у вас есть два варианта. Во-первых, информацию можно записать в выходные данные отображения (вместо стандартного потока ошибок) для последующего анализа и обобщения задачей свертки. Обычно такой подход требует внесения структурных изменений в программу, так что начать лучше с других приемов. Во-вторых, можно написать программу (для MapReduce, конечно), которая будет анализировать журналы, созданные вашим заданием.

Мы включим средства отладки в функцию отображения (версия 4), так как мы хотим узнать, как выглядят исходные данные, приводящие к появлению аномального вывода:

```
public class MaxTemperatureMapper  
    extends Mapper<LongWritable, Text, Text, IntWritable> {  
  
    enum Temperature {  
        OVER_100  
    }  
  
    private NcdcRecordParser parser = new NcdcRecordParser();  
    @Override  
    public void map(LongWritable key, Text value, Context context)  
        продолжение ↗
```

```
throws IOException, InterruptedException {  
  
    parser.parse(value);  
    if (parser.isValidTemperature()) {  
        int airTemperature = parser.getAirTemperature();  
        if (airTemperature > 1000) {  
            System.err.println("Temperature over 100 degrees for input: " + value);  
            context.setStatus("Detected possibly corrupt record: see logs.");  
            context.getCounter(Temperature.OVER_100).increment(1);  
        }  
        context.write(new Text(parser.getYear()), new IntWritable(airTemperature));  
    }  
}  
}
```

Если температура превышает 100°С (числовое значение 1000 — температуры задаются в десятых долях градусов), мы выводим подозрительную строку в стандартный поток ошибок, а также обновляем сообщение о состоянии задачи отображения методом `setStatus()` для объекта `Context`. Кроме того, мы увеличиваем счетчик, который в Java представляется полем типа `enum`. В этой программе для подсчета записей с температурой выше 100°С определяется одно поле `OVER_100`.

Внесите эти изменения, перекомпилируйте код, заново создайте JAR-файл и запустите задание. Пока оно работает, откройте страницу задач.

Страница задач

Страница задач содержит ссылки для получения более подробной информации по задачам. Например, щелчок на ссылке «tar» открывает страницу с информацией обо всех задачах отображения. Также можно ограничиться просмотром только завершенных задач. На рис. 5.3 изображена часть страницы для запуска задания с отладочными командами. Каждая строка таблицы соответствует одной задаче. В ней представлена такая информация, как начальное и завершающее время выполнения задачи, любые ошибки, полученные от трекера задач, и ссылка для просмотра счетчиков отдельной задачи.

Столбец «Status» может пригодиться при отладке — в нем содержится последнее сообщение о состоянии задачи. Перед запуском задачи в качестве состояния выводится строка «initializing», а когда задача начнет читать записи, в столбце выводится информация о текущем сплите в виде имени файла со смещением в байтах и длиной. На рисунке показано состояние, установленное нами для отладки задачи `task_200904110811_0003_m_000044`; перейдем на страницу журналов, чтобы найти соответствующее отладочное сообщение. (Обратите внимание на увеличенное количество счетчиков для этой задачи — значение нашего пользовательского счетчика в ней отлично от нуля.)

Страница расширенной информации о задаче

На странице задач можно щелкнуть на любой задаче, чтобы вывести дополнительную информацию о ней. На странице расширенной информации о задаче (рис. 5.4) перечислены все попытки выполнения задачи. В нашем случае попытка была всего одна, и она завершилась успешно. В таблице содержатся другие полезные данные — например, узел, на котором была сделана попытка запуска, а также ссылки на счетчики и журналы задачи.

Столбец «Actions» содержит ссылки для уничтожения попытки запуска. По умолчанию данная возможность отключена, а веб-интерфейс доступен только для чтения. Чтобы активизировать ссылки действий, задайте свойству `webinterface.private.actions` значение `true`.

Hadoop map task list for job 200904110811_0003 on ip-10-250-110-47						
Completed Tasks						
Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_200904110811_0003_m_000043	100.00%	hdfs://ip-10-250-110-47.ec2.internal /user/root/input/hcdc/all /1949.gz:0+220338475	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 18sec)		10
task_200904110811_0003_m_000044	100.00%	Detected possibly corrupt record: see logs.	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		11
task_200904110811_0003_m_000045	100.00%	hdfs://ip-10-250-110-47.ec2.internal /user/root/input/hcdc/all /1970.gz:0+208374610	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		10

Рис. 5.3. Страница задач

Job job_200904110811_0003									
All Task Attempts									
Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
taskattempt_200904110811_0003_m_000044_0	default-rack/ip-10-250-110-47.ec2.internal	SUCCEEDED	100.00%	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 19sec)	Last 4KB read 100% All		11	
Input Split Locations									
<code>default-rack/10.250.202.127</code>									
<code>default-rack/10.250.123.223</code>									
<code>default-rack/10.250.115.79</code>									
Go back to the job Go back to Job Tracker									
Hadoop, 2009.									

Рис. 5.4. Страница расширенной информации о задаче

Задавая свойству `webinterface.private.actions` значение `true`, вы также разрешаете любому пользователю с доступом к веб-интерфейсу HDFS удалять файлы. Свойство `dfs.web.ugi` определяет пользователя, с правами которого работает веб-интерфейс HDFS; с его помощью можно управлять тем, какие файлы могут просматриваться и удаляться пользователями.

Для задач отображения также имеется секция с информацией о том, на каких узлах располагается входной сплит.

Переходя по одной из ссылок журналов успешной попытки запуска задачи (можно просмотреть последние 4 или 8 Кбайт каждого файла или же весь файл), мы находим подозрительную запись:

```
Temperature over 100 degrees for input:  
033599999433181957042302005+37950+139117SA0  
+0004RJSNV020113590031500703569999994  
3320195701010005+35317+139650SA0+000899999V0200235900265007624  
9N004000599+0067...
```

Похоже, она отличается по формату от других записей. В ней присутствуют пробелы, которые не описаны в спецификации.

После завершения задания мы смотрим на значение счетчика, определенного нами для подсчета количества записей с температурой более 100°C в наборе данных. Значение счетчика можно вывести как через веб-интерфейс, так и в командной строке:

```
% hadoop job -counter job_200904110811_0003  
'v4.MaxTemperatureMapper$Temperature' \ OVER_100  
3
```

С параметром `-counter` передается идентификатор задания, имя группы счетчиков (в нашем случае это полное имя класса) и имя счетчика. Во всем наборе данных, состоящем из миллиарда записей, обнаружены всего три записи с некорректным форматом. Игнорирование некорректных записей — стандартное решение многих проблем в больших данных, хотя в нашем случае необходима осторожность, так как мы определяем максимальную температуру вместо вычисления какой-нибудь обобщающей характеристики. Впрочем, игнорирование трех записей вряд ли изменит общий результат.

Обработка некорректных данных

Входные данные, создающие проблему, пригодятся — мы можем использовать их в тесте, чтобы проверить правильность работы отображения. В следующем тесте MRUnit мы проверяем, обновляется ли счетчик для некорректно сформированных входных данных:

```
@Test
public void parsesMalformedTemperature() throws IOException,
    InterruptedException {
    Text value = new Text("0335999999433181957042302005+37950+139117SA0 +0004" +
        // Год ^^^^
        "RJSN V02011359003150070356999999433201957010100005+353");
        // Температура ^^^^^
    Counters counters = new Counters();
    new MapDriver<LongWritable, Text, Text, IntWritable>()
        .withMapper(new MaxTemperatureMapper())
        .withInputValue(value)
        .withCounters(counters)
        .runTest();
    Counter c = counters.findCounter(MaxTemperatureMapper.Temperature.MALFORMED);
    assertThat(c.getValue(), is(1L));
}
```

Запись, которая создает проблемы, отличается по формату от других строк. В листинге 5.12 приведена измененная версия программы (версия 5), игнорирующая строки, в которых поле температуры не имеет начального знака (плюса или минуса). Мы также ввели счетчик для измерения количества записей, проигнорированных по этой причине.

Листинг 5.12. Функция отображения для поиска максимальной температуры

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    enum Temperature {
        MALFORMED
    }

    private NcdcRecordParser parser = new NcdcRecordParser();

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

        parser.parse(value);
        if (parser.isValidTemperature()) {
            int airTemperature = parser.getAirTemperature();
            context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
        } else if (parser.isMalformedTemperature()) {
            System.err.println("Ignoring possibly corrupt input: " + value);
        }
    }
}
```

продолжение ↗

Листинг 5.12 (продолжение)

```
        context.getCounter(Temperature.MALFORMED).increment(1);
    }
}
}
```

Журналы Hadoop

Hadoop создает журналы в разных местах и для разных аудиторий. Краткие описания журналов приведены в табл. 5.2.

Таблица 5.2. Типы журналов Hadoop

Журналы	Основная аудитория	Описание	Дополнительная информация
Журналы системных демонов	Администраторы	Каждый демон Hadoop создает файл журнала (с использованием log4j) и другой файл, объединяющий стандартный вывод и поток ошибок. Данные записываются в каталог, определенный переменной окружения HADOOP_LOG_DIR	«Системные журналы», с. 399 «Журналы», с. 530
Журналы аудита HDFS	Администраторы	Журнал всех запросов HDFS, отключенный по умолчанию. Данные записываются в журнал узла имен, хотя место хранения может настраиваться	«Журналы аудита», с. 443
Журналы истории заданий MapReduce	Пользователи	Журнал событий, происходящих в ходе выполнения задания (таких, как завершение задач). Данные сохраняются централизованно на трекере задач и в подкаталоге _logs/history выходном каталога задания	«История заданий», с. 231
Журналы задач Map-Reduce	Пользователи	Каждый дочерний процесс трекера задач создает журнал с использованием log4j (called syslog), файл для данных, отправляемых в стандартный вывод, и файл для стандартных ошибок. Данные записываются в подкаталог user-logs каталога, определенного переменной окружения HADOOP_LOG_DIR	Этот раздел

Как было показано в предыдущем разделе, с журналами задач MapReduce можно работать через веб-интерфейс — пожалуй, это самый удобный способ их просмотра.

Также можно найти файлы журналов в локальной файловой системе трекера задач, выполнившего попытку запуска задачи; данные хранятся в каталоге, имя которого совпадает с именем задачи. Если повторное использование JVM задач включено (см. «Повторное использование JVM задач», с. 292), в каждом файле накапливаются журнальные данные на протяжении всей работы виртуальной машины Java, поэтому в них накапливаются данные о многих попытках выполнения задач. Веб-интерфейс скрывает это обстоятельство; в нем отображается только та часть, которая относится к просматриваемой попытке.

Записать данные в файлы журналов несложно. Все, что пишется в стандартный вывод или стандартный поток ошибок, направляется в соответствующий файл журнала. (Конечно, при использовании Streaming стандартный вывод используется для выходных данных отображения и свертки, поэтому записанные в него данные в журнал не попадут).

В Java запись в системный журнал задачи осуществляется через Apache Commons Logging API. Пример представлен в листинге 5.13.

Листинг 5.13. Функция отображения, выполняющая запись в стандартный вывод и использующая Apache Commons Logging API

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.mapreduce.Mapper;
public class LoggingIdentityMapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
    extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

    private static final Log LOG = LogFactory.getLog(LoggingIdentityMapper.class);

    @Override
    public void map(KEYIN key, VALUEIN value, Context context)
        throws IOException, InterruptedException {
        // Запись в файл стандартного вывода
        System.out.println("Map key: " + key);

        // Запись в файл системного журнала
        LOG.info("Map key: " + key);
        if (LOG.isDebugEnabled()) {
            LOG.debug("Map value: " + value);
        }
        context.write((KEYOUT) key, (VALUEOUT) value);
    }
}
```

По умолчанию используется журнальный уровень `INFO`, так что сообщения уровня `DEBUG` не отображаются в системном журнале задач. Однако иногда требуется просмотреть и эти сообщения; для этого следует задать соответствующее значение свойству `mapred.map.child.log.level` или `mapred.reduce.child.log.level` (начиная с версии 0.22). Так, в нашем примере команда для включения в журнал сообщений отображения выглядит следующим образом:

```
% hadoop jar hadoop-examples.jar LoggingDriver -conf conf/hadoop-cluster.xml \
-D mapred.map.child.log.level=DEBUG input/ncdc/sample.txt logging-out
```

По умолчанию журналы удаляются минимум через 24 часа (промежуток времени задается свойством `mapred.userlog.retain.hours` property). Также можно ограничить минимальный размер файла журнала при помощи свойства `mapred.userlog.limit.kb`, которое по умолчанию равно 0 (размер файлов не ограничивается).



В некоторых ситуациях приходится отлаживать проблемы, которые, как вы подозреваете, происходят в виртуальной машине Java, выполняющей команду Hadoop, а не в кластере. Чтобы направить журнальные сообщения уровня `DEBUG` на консоль, используйте вызов следующего вида:

```
% HADOOP_ROOT_LOGGER=DEBUG,console hadoop fs -text /foo/bar
```

Удаленная отладка

В некоторых ситуациях имеющейся информации недостаточно для диагностики ошибки, и вам требуется запустить отладчик для этой задачи. При выполнении заданий в кластере запуск отладчика усложняется тем, что вы не знаете, какой узел будет обрабатывать ту или иную часть входных данных, и не можете заранее настроить свой отладчик. Впрочем, есть и другие варианты.

Локальное воспроизведение сбоя

Часто в задаче последовательно происходит сбой для некоторых исходных данных. Вы можете попытаться воспроизвести сбой локально — загрузите файл, который вызывает сбой, и запустите задание локально (возможно, с использованием такого отладчика, как VisualVM для Java).

Использование средств отладки JVM

Сбои часто возникают из-за нехватки памяти в виртуальной машине Java, обслуживающей задачу. Включите в значение свойства `mapred.child.java.opts` параметр `-XX:-HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/путь/к/дампу`. Он создает

дамп состояния кучи, который позднее может быть проанализирован при помощи таких инструментов, как *jhat* или Eclipse Memory Analyzer. Учтите, что настройки JVM должны добавляться к текущим настройкам памяти, определенным в `mapred.child.java.opts`. Эта тема более подробно рассмотрена в разделе «Память», с. 396.

Профилирование задач

Профилирование Java дает расширенную информацию о работе JVM, и Hadoop предоставляет механизм профилирования подмножества задач в задании. См. «Профилирование», с. 244.

IsolationRunner

В старые версии Hadoop входил специальный исполнитель задач IsolationRunner, который мог заново запускать сбойные задачи в кластере. К сожалению, в новых версиях он отсутствует, но вы можете найти информацию о его замене по адресу <https://issues.apache.org/jira/browse/MAPREDUCE-2637>.

Иногда бывает полезно сохранить промежуточные файлы сбойной попытки выполнения задачи для последующего анализа, особенно если дамп или профильные файлы создаются в рабочем каталоге задачи. Чтобы сохранить файлы сбойной задачи, задайте свойству `keep.failed.task.files` значение `true`.

Промежуточные файлы также можно сохранить и для успешно выполненных задач для последующего анализа. Для этого задайте свойству `keep.task.files.pattern` регулярное выражение, совпадающее с идентификаторами нужных задач.

Чтобы проанализировать промежуточные файлы, зайдите на узел, на котором произошел сбой, и найдите каталог соответствующей попытки. Он будет находиться под одним из локальных каталогов MapReduce, определяемых свойством `mapred.local.dir` (за дополнительной информацией обращайтесь к разделу «Важные свойства демонов Hadoop», с. 401). Если свойство содержит список каталогов, разделенный запятыми (для распределения нагрузки по физическим дискам компьютера), возможно, вам придется просмотреть все каталоги, прежде чем вы найдете каталог нужной попытки. Каталог попытки выполнения задачи находится по адресу:

`mapred.Local.dir/taskTracker/jobcache/идентификатор_задания/идентификатор_попытки`

Оптимизация задания

Когда задание заработает, разработчики обычно спрашивают: «Можно ли заставить его работать быстрее?»

В Hadoop есть несколько факторов, которые стоит проверить в первую очередь, если у вас возникают проблемы с производительностью. Прежде чем браться за профилирование или оптимизацию на уровне задач, пройдитесь по контрольному списку из табл. 5.3.

Таблица 5.3. Контрольный список настройки заданий

Область	Описание	Дополнительная информация
Количество отображений	Сколько времени выполняются задачи отображения? Если в среднем их выполнение занимает несколько секунд, подумайте, нельзя ли сократить их количество и заставить их работать дольше — например, минуту или около того. Возможность такого преобразования зависит от используемого формата входных данных	«Малые файлы и CombineFileInputFormat», с. 315
Количество сверток	Для достижения максимальной производительности количество сверток должно быть чуть меньше количества слотов свертки в кластере. В этом случае свертки завершаются одной «волной» с полным использованием кластера в фазе свертки.	«Выбор количества задач свертки», с. 305
Комбинирующие функции	Проверьте, нельзя ли использовать в вашем приложении комбинирующую функцию для сокращения объема данных, проходящих через тасовку	«Комбинирующие функции», с. 68
Промежуточное сжатие	Включение сжатия выходных данных отображений почти всегда ускоряет выполнение заданий	«Сжатие выходных данных отображений», с. 139
Пользовательская сериализация	Если вы используете собственные объекты Writable или пользовательские сравнения, не забудьте реализовать RawComparator	«Реализация RawComparator», с. 155
Настройка процесса тасовки	Механизм тасовки MapReduce предоставляет около дюжины параметров для настройки управления памятью. С ними вы сможете «выжать» из тасовки всю возможную производительность	«Настройка конфигурации», с. 283

Профилирование

Профилирование заданий, выполняемых в распределенных системах (таких, как MapReduce), тоже создает определенные трудности. Hadoop позволяет включить

профилирование для части задач в задании и при завершении каждой задачи отправляет информацию на вашу машину для последующего анализа стандартными средствами профилирования.

Конечно, можно — и, пожалуй, проще — профилировать задание в локальном исполнителе заданий. И если вы сможете предоставить достаточно входных данных для полноценной отработки задач отображения и свертки, вы можете получить ценную информацию для улучшения производительности отображений и сверток. Впрочем, здесь скрывается пара проблем. Среда локального исполнителя заданий сильно отличается от среды кластера, и схемы передачи данных тоже выглядят иначе. Оптимизация вычислительной производительности кода может стать бесполезной, если ваше задание MapReduce (как и большинство заданий) завязано на ввод/вывод. Чтобы настройка была эффективной, следует сравнивать новое время выполнения со старым при выполнении в реальном кластере. К сожалению, это проще сказать, чем сделать — время выполнения зависит от конкуренции за ресурсы с другими заданиями и решениями по распределению задач, принимаемыми планировщиком. Чтобы получить представление об исполнении задания в этих условиях, проведите серию запусков (с изменениями и без) и проверьте статистическую значимость любых улучшений.

К сожалению, некоторые проблемы (скажем, перерасходование памяти) могут быть воспроизведены только в кластере. В таких ситуациях возможность профилирования «на месте» незаменима.

Профилировщик HPROF

Для управления профилированием используются различные конфигурационные свойства, для работы с которыми также можно использовать вспомогательные методы `JobConf`. Следующая модификация `MaxTemperatureDriver` (версия 6) включает в себя удаленное профилирование HPROF. Программа профилирования HPROF, входящая в поставку JDK, обладает простейшими возможностями, но может предоставить полезную информацию об использовании процессора и памяти программой¹:

```
Configuration conf = getConf();
conf.setBoolean("mapred.task.profile", true);
conf.set("mapred.task.profile.params", "-agentlib:hprof=cpu=samples," +
    продолжение ↗
```

¹ HPROF использует для профилирования вставку байт-кода, поэтому для ее использования вам не придется перекомпилировать свое приложение со специальными параметрами. За дополнительной информацией о HPROF обращайтесь к статье Келли О'Хея (Kelly O'Hair) «HPROF: A Heap/CPU Profiling Tool in J2SE 5.0» по адресу <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>.

```

"heap=sites,depth=6,force=n,thread=y,verbose=n,file=%s");
conf.set("mapred.task.profile.maps", "0-2");
conf.set("mapred.task.profile.reduces", ""); // Без сверток
Job job = new Job(conf, "Max temperature");

```

Первая строка включает профилирование, которое по умолчанию отключено. (В новом API вместо `mapred.task.profile` можно использовать константу `JobContext.TASK_PROFILE`.)

Затем задаются параметры профиля — дополнительные аргументы командной строки, передаваемые JVM задачи. (При включенном профилировании для каждой задачи выделяется новая виртуальная машина Java, даже если повторное использование JVM включено; см. «Повторное использование JVM задач», с. 292.) Параметры по умолчанию задают профилировщик HPROF; мы задаем дополнительный параметр `depth=6`, чтобы увеличить глубину трассировки стека по сравнению с настройками HPROF по умолчанию. (Использование `JobContext.TASK_PROFILE_PARAMS` эквивалентно заданию свойства `mapred.task.profile.params`.)

Наконец, мы указываем, какие задачи следует профилировать. Обычно профильная информация должна передаваться минимальным числом задач, поэтому мы при помощи свойств `mapred.task.profile.maps` и `mapred.task.profile.reduces` задаем диапазон идентификаторов задач (отображения или свертки), для которых запрашивается профильная информация. Мы задаем свойству `maps` значение `0-2` (на самом деле оно используется по умолчанию); это означает, что профилируются задачи отображения с идентификаторами 0, 1 и 2. Поддерживаются наборы диапазонов и открытые диапазоны. Например, запись `0-1,4,6-` определяет все задачи, кроме задач с идентификаторами 2, 3 и 5. Профилируемыми задачами также можно управлять при помощи констант `JobContext.NUM_MAP_PROFILES` для задач отображения и `JobContext.NUM_REDUCE_PROFILES` для задач свертки.

При запуске задания с измененной управляющей программой выходные данные профилирования сохраняются в каталоге, из которого было запущено задание. Так как количество профилируемых задач невелико, задание можно запустить для подмножества набора данных.

Ниже приведен фрагмент одного из профильных файлов отображения с информацией об использовании процессора:

```

CPU SAMPLES BEGIN (total = 1002) Sat Apr 11 11:17:52 2009
rank  self  accum  count trace method
 1  3.49%  3.49%    35 307969 java.lang.Object.<init>
 2  3.39%  6.89%    34 307954 java.lang.Object.<init>
 3  3.19% 10.08%    32 307945 java.util.regex.Matcher.<init>
 4  3.19% 13.27%    32 307963 java.lang.Object.<init>
 5  3.19% 16.47%    32 307973 java.lang.Object.<init>

```

Переходя к трассировке с номером 307973, мы получаем данные трассировки стека из того же файла:

```
TRACE 307973: (thread=200001)
  java.lang.Object.<init>(Object.java:20)
  org.apache.hadoop.io.IntWritable.<init>(IntWritable.java:29)
  v5.MaxTemperatureMapper.map(MaxTemperatureMapper.java:30)
  v5.MaxTemperatureMapper.map(MaxTemperatureMapper.java:14)
  org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:50)
  org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:356)
```

Итак, задача отображения проводит 3% времени за конструированием объектов `IntWritable`. Это наблюдение наводит на мысль о повторном использовании выводимых экземпляров `Writable` — не приведет ли оно к повышению производительности для нашего конкретного кластера и набора данных? В одном из экспериментов объекты `Writable` сохранялись в переменных экземпляров. Запустив измененную программу в кластере из 11 узлов, я не увидел статистически значимых различий в общем времени выполнения. Конечно, ваши результаты могут выглядеть иначе.

Другие профилировщики

Механизм получения профильных выходных данных зависит от HPROF, так что при использовании другого профилировщика вам придется вручную получить выходные данные профилировщика с трекеров задач для анализа.

Если профилировщик не установлен на всех машинах трекеров задач, рассмотрите возможность использования распределенного кэша (см. «Распределенный кэш», с. 375), чтобы двоичные файлы профилировщика были доступны на всех необходимых машинах.

Модель MapReduce

До настоящего момента мы рассматривали механику написания программ, использующих MapReduce. Однако необходимо рассмотреть еще один важный аспект — преобразование задачи обработки данных в модель MapReduce.

Обработка данных, с которой мы имели дело в примерах, сводилась к относительно простой задаче: определению максимальной зарегистрированной температуры за заданные годы. Усложнение обработки данных обычно проявляется в увеличении количества заданий MapReduce, а не в усложнении функций отображения и свертки. Иначе говоря, обычно приходится решать проблемы, связанные

с добавлением новых заданий, а не с увеличением сложности. Для решения более сложных задач стоит воспользоваться языком более высокого уровня, чем MapReduce, — например, Pig, Hive, Cascading, Cascalog или Crunch. Явное преимущество таких решений заключается в том, что они избавляют вас от необходимости выполнения преобразований в задания MapReduce, позволяя сосредоточиться на выполняемых аналитических операциях.

Наконец, много полезной информации о проектировании алгоритмов MapReduce можно найти в книге Джимми Лина (Jimmy Lin) и Криса Дайера (Chris Dyer) «Data-Intensive Text Processing with MapReduce» (Morgan & Claypool Publishers, 2010, <http://mapreduce.me/>).

Разложение задачи на задания MapReduce

Рассмотрим пример более сложной задачи, которую нужно преобразовать к модели MapReduce.

Допустим, нам потребовалось вычислить среднее значение максимальных зарегистрированных температур для каждого дня года и для каждой метеорологической станции. Скажем, чтобы узнать среднюю максимальную температуру, зарегистрированную станцией 029070-99999 за 1 января, мы вычисляем среднее значение максимальных дневных температур для этой станции за 1 января 1901 г.; 1 января 1902 г. и так далее до 1 января 2000 г.

Как выполнить такие вычисления в MapReduce? Они естественным образом распадаются на две стадии:

1. Вычисление максимальной дневной температуры для каждой пары «станция-дата».

Программа MapReduce представляет собой разновидность программы поиска максимальной температуры, за исключением того, что ключом в этом случае является составная пара «станция-дата», а не просто год.

2. Вычисление средней ежедневной температуры для каждого ключа «станция-день-месяц».

Функция отображения берет выходные записи предыдущего задания (станция-дата, максимальная температура) и преобразует их в записи (станция-день-месяц, максимальная температура), удаляя компонент года. Функция свертки вычисляет среднее значение максимальных температур для каждого ключа «станция-день-месяц».

Вывод первой стадии для интересующей нас станции выглядит так (скриптий *mean_max_daily_temp.sh* из примера использует реализацию для Hadoop Streaming):

```
029070-99999 19010101 0  
029070-99999 19020101 -94  
...
```

Первые два поля образуют ключ, а последнее поле содержит максимальную температуру из всех показаний для конкретной станции и даты. На второй стадии эти дневные максимумы усредняются по годам:

```
029070-99999 0101 -68
```

Эта строка означает, что средняя ежедневная температура на 1 января для станции 029070-99999 составляла -6.8°C .

Вычисления можно провести за одну стадию MapReduce, но это потребует большей работы со стороны программиста¹. Вместе с тем увеличение количества (и упрощения) стадий MapReduce улучшает композицию и упрощает сопровождение отображений и сверток.

Впрочем, с композицией функций отображения и свертки можно пойти еще дальше. Отображение обычно выполняет операции разбора входного формата, проекции (выбора нужных полей) и фильтрации (удаления записей, не представляющих интереса). В приводившихся ранее примерах все эти функции были реализованы в одном отображении. Однако возможно и другое решение — разбиение на частичные отображения и их сцепление с использованием библиотечного класса `ChainMapper`, входящего в поставку Hadoop. А если воспользоваться `ChainReducer`, вы можете выполнить цепочку отображений, затем свертку и еще одну цепочку отображений в одном задании MapReduce.

JobControl

Если в потоке операций MapReduce задействовано несколько заданий, возникает вопрос: как организовать последовательное выполнение этих заданий? Есть несколько решений, а выбор зависит прежде всего от того, используется ли линейная цепочка заданий или более сложный направленный ациклический граф.

Для линейной цепочки проще всего запускать задания одно за другим, дожидаясь успешного завершения задания перед запуском следующего:

```
JobClient.runJob(conf1);  
JobClient.runJob(conf2);
```

¹ Впрочем, это может стать интересным упражнением. Подсказка: обратитесь к разделу «Вторичная сортировка», с. 361.

Если задание завершается неудачей, метод `runJob()` выдает исключение `IOException`, так что последующие задания в конвейере выполняться не будут. Возможно, в зависимости от приложения следует организовать перехват исключения с очисткой всех промежуточных данных, созданных предыдущими заданиями.

Такой подход в целом напоминает новый MapReduce API, если не считать того, что вы должны проверить логическое возвращаемое значение метода `waitForCompletion()` объекта `Job`: `true` — задание завершилось успешно, `false` — произошел сбой.

Если последовательность заданий сложнее линейной цепочки, существуют специальные библиотеки, которые помогут организовать поток операций (хотя они также могут использоваться для линейных цепочек и даже одноразовых заданий). Простейшее решение — класс `JobControl` из пакета `org.apache.hadoop.mapreduce.jobcontrol` (также существует эквивалентный класс в пакете `org.apache.hadoop.mapred.jobcontrol`). Экземпляр `JobControl` представляет собой граф выполняемых заданий. Разработчик добавляет конфигурации заданий, а затем передает `JobControl` информацию о зависимостях между заданиями. `JobControl` запускается в программном потоке и выполняет задания в порядке зависимостей. Разработчик может запрашивать информацию о ходе работы, а после завершения заданий можно получать информацию о состоянии и возможных ошибках. Если задание завершается с ошибкой, `JobControl` не будет выполнять его зависимости.

Apache Oozie

Apache Oozie — система управления операциями зависимых заданий. Она состоит из двух основных частей: *ядра потока операций*, которое хранит и запускает потоки операций, состоящие из разных типов заданий Hadoop (MapReduce, Pig, Hive и т. д.), и *координатора*, запускающего задания на основании заранее определенных расписаний и доступности данных. Система Oozie спроектирована с учетом возможного масштабирования; она может управлять выполнением в кластерах Hadoop тысяч потоков операций, включающих в себя десятки заданий.

Oozie также упрощает повторное выполнение сбойных потоков операции, так как на выполнение успешных частей время уже не тратится. Эту возможность оценят все, кому доводилось управлять сложной пакетной системой, — они знают, как трудно бывает восстановиться после пропуска заданий из-за неработоспособности оборудования или сбоев. (Более того, приложения координатора, представляющие собой отдельный конвейер данных, можно упаковать в пакет и запускать как единое целое.)

В отличие от класса `JobControl`, работающего на клиентской машине, отправляющей задания, Oozie работает в виде службы в кластере, а клиенты отправляют

определения потоков операций для немедленного или отложенного выполнения. В контексте Oozie поток операций представляет собой направленный ациклический граф узлов *действий* (action nodes) и узлов *управления* (control-flow nodes).

Узел действия выполняет некоторую задачу из потока операций — например, перемещение файлов в HDFS, запуск задания MapReduce, Streaming, Pig или Hive, импортирование Sqoop, выполнение произвольного сценария или Java-программы. Узел потока управления организует выполнение потока операций между действиями, поддерживая такие конструкции, как условная логика (таким образом, существует возможность выбора нескольких ветвей выполнения в зависимости от результата предшествующего узла действия) и параллельное выполнение. При завершении потока операций Oozie может обратиться к клиенту с обратным вызовом HTTP, чтобы проинформировать его о состоянии потока. Также существует возможность получения обратных вызовов каждый раз, когда поток операций входит в узел действия или выходит из него.

Определение потока операций Oozie

Определения потоков операций пишутся на XML с использованием языка Hadoop Process Definition Language, спецификация которого размещена на веб-сайте Oozie (<http://incubator.apache.org/oozie/>). В листинге 5.14 приведено определение простого потока операций Oozie для выполнения одного задания MapReduce.

Листинг 5.14. Определение потока операций Oozie для выполнения задания MapReduce

```
<workflow-app xmlns="uri:oozie:workflow:0.1" name="max-temp-workflow">
  <start to="max-temp-mr"/>
  <action name="max-temp-mr">
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="`${nameNode}/user/${wf:user()}/output"/>
      </prepare>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>OldMaxTemperature$OldMaxTemperatureMapper</value>
        </property>
        <property>
          <name>mapred.combiner.class</name>
```

продолжение ↗

Листинг 5.14 (продолжение)

```

<value>OldMaxTemperature$OldMaxTemperatureReducer</value>
</property>
<property>
    <name>mapred.reducer.class</name>
    <value>OldMaxTemperature$OldMaxTemperatureReducer</value>
</property>
<property>
    <name>mapred.output.key.class</name>
    <value>org.apache.hadoop.io.Text</value>
</property>
<property>
    <name>mapred.output.value.class</name>
    <value>org.apache.hadoop.io.IntWritable</value>
</property>
<property>
    <name>mapred.input.dir</name>
    <value>/user/${wf:user()}/input/ncdc/micro</value>
</property>
<property>
    <name>mapred.output.dir</name>
    <value>/user/${wf:user()}/output</value>
</property>
</configuration>
</map-reduce>
<ok to="end"/>
<error to="fail"/>
</action>
<kill name="fail">
    <message>MapReduce failed,
        error message[ ${wf:errorMessage(wf:lastErrorNode())} ]
    </message>
</kill>
<end name="end"/>
</workflow-app>
```

Поток операций состоит из трех узлов потока управления и одного узла действия: узел управления **start**, узел действия **map-reduce**, узел управления **kill** и узел управления **end**. Узлы и разрешенные переходы между ними показаны на рис. 5.5.

Все потоки операций имеют один начальный узел **start** и один конечный узел **end**. При запуске задания потока операций происходит переход в узел **start** (действие **max-temp-mr** в нашем примере). Задание потока операций завершается успешно при переходе к узлу **end**. Если задание потока операций переходит к узлу **kill**,

считается, что произошел сбой, и система выводит соответствующее сообщение об ошибке, заданное элементом `message` в определении.

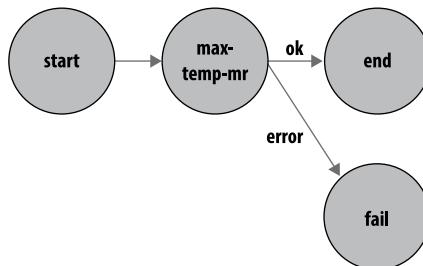


Рис. 5.5. Диаграмма переходов потока операций Oozie

Основная часть файла определения потока операций обуславливает действие `map-reduce`. Первые два элемента, `job-tracker` и `name-node`, определяют трекер заданий, которому будет отправляться задание, узел имен (а на самом деле URI файловой системы Hadoop) для входных и выходных данных. Оба элемента параметризованы, так что определение потока операций не привязано к конкретному кластеру (что упрощает тестирование). Как мы вскоре увидим, параметры задаются свойствами задания потока операций в момент отправки.

Необязательный элемент `prepare` выполняется перед заданием MapReduce и используется для удаления каталогов (а также их создания в случае необходимости, хотя в нашем примере это не используется). Убедившись в том, что перед запуском задания выходной каталог находится в стабильном состоянии, Oozie может безопасно заново выполнить действие в случае сбоя задания.

Выполняемое задание MapReduce определяется элементом `configuration` сложенными элементами, определяющими пары «имя-значение» конфигурации Hadoop. Раздел конфигурации MapReduce может рассматриваться как декларативная замена для классов управляющих программ, которые повсеместно используются в книге для запуска программ MapReduce (как, например, в листинге 2.6).

Два нестандартных свойства Hadoop `mapred.input.dir` и `mapred.output.dir` задают входные пути `FileInputFormat` и выходной путь `FileOutputFormat` соответственно.

В нескольких местах определения потока операций используется синтаксис JSP Expression Language (EL). Oozie предоставляет набор функций для взаимодействия с потоком операций. Например, конструкция `${wf:user()}`` возвращает имя пользователя, запустившего текущее задание; мы используем ее для настройки путей файловой системы. В спецификации Oozie перечислены все функции EL, поддерживаемые Oozie.

Упаковка и установка приложений Oozie

Приложение потока операций состоит из определения потока операций и сопутствующих ресурсов (JAR-файлы MapReduce, сценарии Pig и т. д.), необходимых для его выполнения. Приложения должны использовать простую структуру каталогов и развертываться в HDFS, где Oozie сможет работать с ними. Для нашего приложения все файлы были помещены в базовый каталог *max-temp-workflow*:

```
max-temp-workflow/
└── lib/
    └── hadoop-examples.jar
└── workflow.xml
```

Файл определения потока операций *workflow.xml* должен находиться на верхнем уровне этого каталога. JAR-файлы, содержащие классы MapReduce приложения, помещаются в каталог *lib*.

Приложения потока операций, соответствующие этой структуре, могут строиться любыми подходящими средствами сборки — такими, как Ant или Maven; пример приведен в архиве кода, прилагаемого к книге. Построенное приложение копируется в HDFS стандартными средствами Hadoop. Для нашего приложения соответствующая команда выглядит так:

```
% hadoop fs -put hadoop-examples/target/max-temp-workflow max-temp-workflow.
```

Запуск задания потока операций Oozie

Наконец, давайте посмотрим, как запустить задание потока операций для только что загруженного приложения. Для этого мы воспользуемся программой командной строки *oozie* — клиентской программой для взаимодействия с сервером Oozie. Для удобства мы экспортируем переменную окружения *OOZIE_URL*, чтобы передать команде *oozie* информацию об используемом сервере Oozie (в данном примере используется сервер, запущенный локально):

```
% export OOZIE_URL="http://localhost:11000/oozie".
```

Программа *oozie* поддерживает много подкоманд (чтобы получить полный список, введите *oozie help*). Для запуска задания потока операций вызывается подкоманда *job* с параметром *-run*:

```
% oozie job -config ch05/src/main/resources/max-temp-workflow.properties -run
job: 0000009-120119174508294-oozie-tom-W
```

Параметр *-config* задает локальный файл свойств Java с определениями параметров из XML-файла потока операций (в нашем случае *nameNode* и *jobTracker*),

а также свойства `oozie.wf.application.path`, которое сообщает Oozie местонахождение приложения в HDFS. Пример содержимого файла свойств:

```
nameNode=dfs://localhost:8020
jobTracker=localhost:8021
oozie.wf.application.path=${nameNode}/user/${user.name}/max-temp-workflow
```

Чтобы получить информацию о состоянии задания потока операций, мы используем параметр `-info` с идентификатором задания, выведенным ранее командой `git` (введите `oozie job`, чтобы получить список всех заданий):

```
% oozie job -info 0000009-120119174508294-oozie-tom-W.
```

В выходных данных показано состояние задания (RUNNING, KILLED или SUCCEEDED). Вся эта информация также доступна через веб-интерфейс Oozie по адресу `http://localhost:11000/oozie`.

После того, как задание будет успешно выполнено, результаты можно просмотреть обычным образом:

```
% hadoop fs -cat output/part-
1949    111
1950     22
```

Рассмотренный пример дает лишь самое поверхностное представление о написании потоков операций Oozie. В документации на веб-сайте Oozie рассмотрено создание более сложных потоков операций, а также написание и выполнение заданий координатора.

6

Как работает MapReduce

В этой главе подробно рассматриваются механизмы работы MapReduce в Hadoop. Этот материал заложит хорошую основу для написания более сложных программ MapReduce, которые будут рассматриваться в двух следующих главах.

Выполнение задания MapReduce

Для запуска задания MapReduce достаточно вызвать всего один метод: `submit()` для объекта `Job` (также можно вызвать метод `waitForCompletion()`, который отправляет задание на выполнение, если оно не было отправлено ранее, и ожидает его завершения)¹. Вызов метода скрывает многие события, происходящие «за кулисами». В этом разделе будут рассмотрены действия, предпринимаемые Hadoop при выполнении задания.

Как было показано в главе 5, процесс выполнения программ MapReduce в Hadoop зависит от пары параметров конфигурации.

В версиях Hadoop до 0.20 включительно механизм выполнения определялся свойством `mapred.job.tracker`. Если свойство конфигурации было равно `local`

¹ В старом MapReduce API используется вызов `JobClient.submitJob(conf)` или `JobClient.runJob(conf)`.

(значение по умолчанию), использовался локальный исполнитель заданий, выполняющий все задание в одной виртуальной машине Java. Локальный исполнитель предназначен для тестирования и запуска программ MapReduce с небольшими наборами данных.

Если же значение свойства `mapred.job.tracker` представляет собой пару из хоста и порта, разделенных двоеточием, то оно интерпретируется как адрес трекера заданий и исполнитель передает задание трекеру заданий по указанному адресу. Весь процесс достаточно подробно описан в следующем разделе.

В Hadoop 2.0 была представлена новая реализация MapReduce. Эта новая реализация (называемая MapReduce 2) построена на базе системы YARN — см. «YARN (MapReduce 2)», с. 265. Пока достаточно сказать, что инфраструктура, используемая для выполнения, задается свойством `mapreduce.framework.name` с возможными значениями `local` (локальный исполнитель заданий), `classic` («классическая» инфраструктура MapReduce, также называемая MapReduce 1, использующая трекеры заданий и задач) и `yarn` (новая инфраструктура).



Не путайте старую и новую версии MapReduce API с реализациями MapReduce — классической и YARN (MapReduce 1 и 2 соответственно). API относится к числу клиентских возможностей, обращенных к пользователю; они определяют способ написания программ MapReduce, тогда как реализации — всего лишь разные способы выполнения программ MapReduce. Поддерживаются все четыре комбинации; и старый, и новый API работают как с MapReduce 1, так и с MapReduce 2. Комбинации, поддерживаемые в разных выпусках Hadoop, перечислены в табл. 1.2.

Классическая реализация MapReduce (MapReduce 1)

Процесс выполнения задания в классической реализации MapReduce изображен на рис. 6.1. На верхнем уровне находятся четыре независимые сущности:

- Клиент, отправляющий задание MapReduce.
- Трекер заданий, координирующий выполнение задания, — приложение Java с главным классом `JobTracker`.
- Трекеры задач, выполняющие задачи, на которые было разбито задание, — приложения Java с главным классом `TaskTracker`.
- Распределенная файловая система (обычно HDFS — см. главу 3), используемая для передачи файлов задания между другими сущностями.

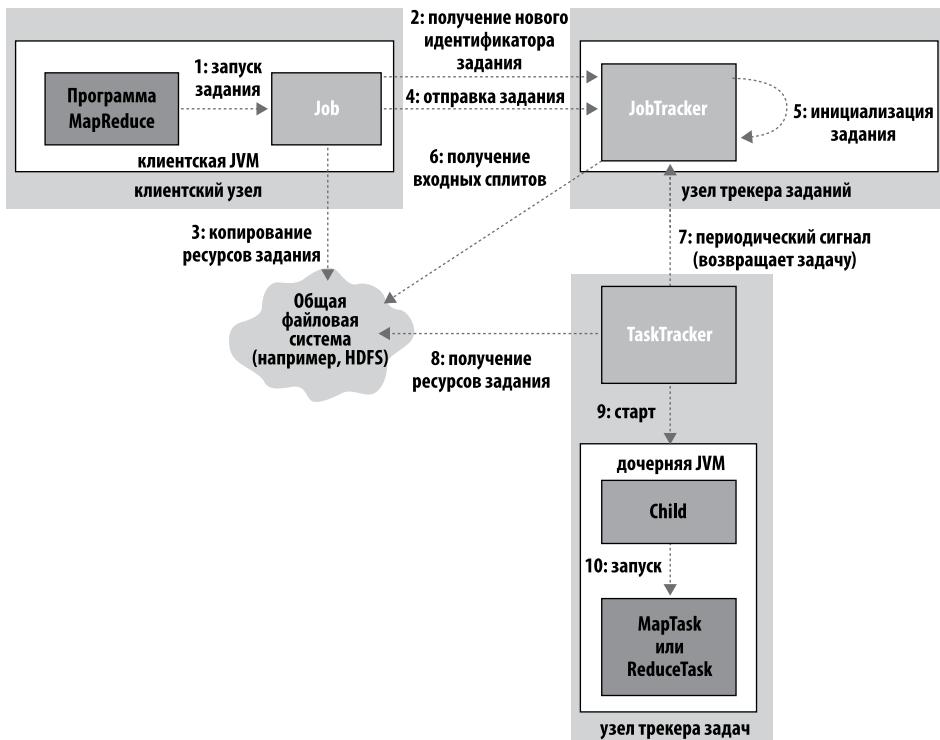


Рис. 6.1. Выполнение заданий MapReduce в Hadoop
с использованием классической инфраструктуры

Отправка заданий

Метод `submit()` класса `Job` создает внутренний экземпляр `JobSubmitter` и вызывает его метод `submitJobInternal()` (шаг 1 на рис. 6.1). После отправки задания `waitForCompletion()` один раз в секунду запрашивает информацию о прогрессе и выводит ее на консоль, если с момента последнего ответа произошли изменения. При успешном завершении задания выводятся счетчики; в противном случае выводится ошибка, ставшая причиной сбоя.

При отправке задания реализация `JobSubmitter`:

- запрашивает у трекера заданий новый идентификатор задания вызовом `mgetNewJobId()` для `JobTracker` (шаг 2);
- проверяет спецификацию вывода для задания. Например, если выходной каталог не указан или уже существует, то задание не отправляется, а программе `MapReduce` передается ошибка;

- вычисляет входные сплиты для задания. Если сплиты вычислить не удается (например, из-за того, что входные пути не существуют), задание не отправляется, а программе MapReduce передается ошибка;
- копирует ресурсы, необходимые для выполнения задания, включая JAR-файл задания, файл конфигурации и вычисленные входные сплиты, в файловую систему трекера заданий — в каталог, имя которого совпадает с идентификатором задания. JAR-файл задания копируется с высоким коэффициентом репликации (определенным свойством `mapred.submit.replication`, по умолчанию равным 10). Таким образом, в кластере появляется много копий, доступных для трекеров задач, запускающих задачи из задания (шаг 3);
- сообщает трекеру заданий, что задание готово к выполнению, вызывая метод `submitJob()` для экземпляра `JobTracker` (шаг 4).

Инициализация заданий

Получив вызов своего метода `submitJob()`, экземпляр `JobTracker` помещает его во внутреннюю очередь, из которой он будет извлечен и инициализирован планировщиком заданий. Инициализация включает в себя создание объекта, который представляет выполняемое задание, инкапсулирует его задачи и содержит служебную информацию для отслеживания состояния и прогресса его задач (шаг 5).

Чтобы создать список выполняемых задач, планировщик заданий сначала получает входные сплиты, вычисленные клиентом, из общей файловой системы (шаг 6). Затем он создает одну задачу отображения для каждого сплита. Количество создаваемых задач свертки определяется свойством `mapred.reduce.tasks` из объекта `Job`, которое задается методом `setNumReduceTasks()`, а пользователь просто создает нужное количество задач свертки. На этой стадии задачам присваиваются идентификаторы.

Кроме задач отображения и свертки, создаются еще две задачи: задача подготовки задания и задача деинициализации задания. Эти задачи выполняются трекерами задач и используются для выполнения кода, настраивающего задание до выполнения каких-либо задач отображения и осуществляющего «зачистку» после завершения всех задач свертки. Выполняемый код определяется экземпляром `OutputCommitter`, назначенным для задания; по умолчанию используется `FileOutputCommitter`. Для задачи подготовки задания он создает итоговый выходной каталог задания и временное рабочее пространство для выходных данных задач, а для задачи деинициализации удаляет временное рабочее пространство. Действия `OutputCommitter` более подробно описаны в разделе «`OutputCommitter`», с. 290.

Назначение задач

Трекеры задач работают в простом цикле, который периодически отправляет трекеру задач сигнал в виде вызова метода. Периодический сигнал сообщает трекеру заданий, что трекер задач продолжает работать; кроме того, он служит каналом для передачи сообщений. При передаче периодического сигнала трекер задач указывает, готов ли он к выполнению новой задачи, и если готов — трекер заданий выделяет ему новую задачу, о чем сообщает трекеру задач при помощи возвращаемого значения периодического сигнала (шаг 7).

Прежде чем выбирать задачу для трекера задач, трекер заданий должен выбрать задание, которому принадлежит присваиваемая задача. Существуют разные алгоритмы планирования (см. «Планирование заданий», с. 277); по умолчанию трекер ведет приоритетный список заданий. Выбрав задание, трекер заданий переходит к выбору задачи.

Трекеры задач используют фиксированное количество слотов для задач отображения и задач свертки, которые присваиваются независимо друг от друга. Например, трекер задач может быть настроен для одновременного выполнения двух задач отображения и двух задач свертки. (Точное количество зависит от количества ядер и доступной памяти на трекере задач; см. «Память», с. 396.) В контексте выделенного задания планировщик по умолчанию заполняет пустые слоты задач отображения до слотов задач свертки. Таким образом, если трекер задач имеет хотя бы один пустой слот задач отображения, трекер заданий выберет задачу отображения; в противном случае будет выбрана задача свертки.

Чтобы выбрать задачу свертки, трекер задач просто берет следующую задачу в своем списке задач свертки, ожидающих выполнения, так как локальность данных в этом случае несущественна. Однако для задач отображения учитывается местонахождение трекера задач в сети, и выбирается задача, входной сплит которой находится как можно ближе к трекеру задач. В оптимальном случае задача выполняется на том же узле, на котором находится сплит. Также задача может быть *сегментно-локальной*, то есть находиться в одном сегменте, но не на одном узле со сплитом. Некоторые задачи не обладают локальностью ни по данным, ни по сегментам и получают свои данные из сегмента, отличного от того, в котором они выполняются. Соотношение задач каждого типа можно определить по счетчикам задания (см. «Встроенные счетчики», с. 338).

Выполнение задач

После того, как трекеру задач была присвоена задача, он должен ее запустить. Сначала трекер локализует JAR-файл задачи, копируя его из общей файловой системы в файловую систему трекера задач. Кроме того, он копирует все необходимые файлы из распределенного кэша на локальный диск; см. «Распределенный кэш»,

с. 375 (шаг 8). Затем трекер задач создает локальный рабочий каталог для задачи, «распаковывает» содержимое JAR-файла в этот каталог и создает экземпляр `TaskRunner` для запуска задачи.

`TaskRunner` запускает новую виртуальную машину Java (JVM, шаг 9) для каждой задачи (шаг 10), чтобы возможные ошибки в пользовательских функциях отображения и свертки не влияли на работоспособность трекера задач (скажем, приводя к его аварийному завершению или «зависанию»). Впрочем, повторное использование JVM между задачами возможно; см. «Повторное использование JVM задач», с. 292.

Дочерний процесс взаимодействует с родителем через связующий интерфейс. Он информирует родителя о ходе выполнения каждой задачи каждые несколько секунд, пока задача не завершится. Каждая задача может выполнять подготовительные и завершающие операции, которые выполняются в той же виртуальной машине, что и сама задача, и определяются экземпляром `OutputCommitter` для задания (см. «`OutputCommitter`», с. 290). Завершающее действие используется для *закрепления* задачи; в случае файловых заданий это означает, что выходные данные записываются в итоговый приемник задачи. Протокол закрепления гарантирует, что в режиме спекулятивного выполнения (см. «Спекулятивное выполнение», с. 288) только одна из задач-дубликатов закрепляется, а другая отменяется.

Streaming и Pipes

И Streaming, и Pipes запускают специальные задачи отображения и свертки для выполнения исполняемых модулей, предоставленных пользователем, и взаимодействия с ними (рис. 6.2).

В случае Streaming задача Streaming взаимодействует с процессом (который может быть написан на любом языке) с использованием стандартного входного и выходного потоков. Напротив, задача Pipes прослушивает сокет и передает процессу C++ номер порта из своего окружения, чтобы при запуске процесс C++ мог создать постоянное подключение через сокет к родительской задаче Java Pipes.

В обоих случаях в ходе выполнения задачи процесс Java передает входные пары «ключ-значение» внешнему процессу, пропускающему их через определенную пользователем функцию отображения или свертки и возвращает выходные пары «ключ-значение» процессу Java. С точки зрения трекера задач все выглядит так, как если бы дочерний процесс трекера задач сам выполнял код отображения или свертки.

Обновления состояния

Задания MapReduce выполняются в пакетном режиме, а их выполнение занимает от нескольких минут до нескольких часов. Соответственно, пользователю важно получать информацию обратной связи о прогрессе. Задание и каждая из его задач

имеет состояние, в которое включается такая информация, как статус (выполнение, успешное завершение, сбой), ход выполнения отображений и сверток, значения счетчиков задания и сообщение или описание (которое может задаваться в пользовательском коде). Состояние изменяется в процессе выполнения задания, как же пользователь узнает об этих изменениях?

Во время выполнения задачи пользователь получает информацию о прогрессе. Для задач отображения в качестве метрики используется доля обработанных входных данных. Для задач свертки задача усложняется, но система и в этом случае может оценить часть обработанных входных данных свертки. Для этого общий прогресс делится на три части, соответствующие трем фазам тасовки (см. «Тасовка и сортировка», с. 279). Так, если задача выполнила свертку для половины своих входных данных, то прогресс задачи составит $5/6$, потому что она завершила фазы копирования и сортировки (по $1/3$ каждая) и наполовину завершила фазу свертки ($1/6$).

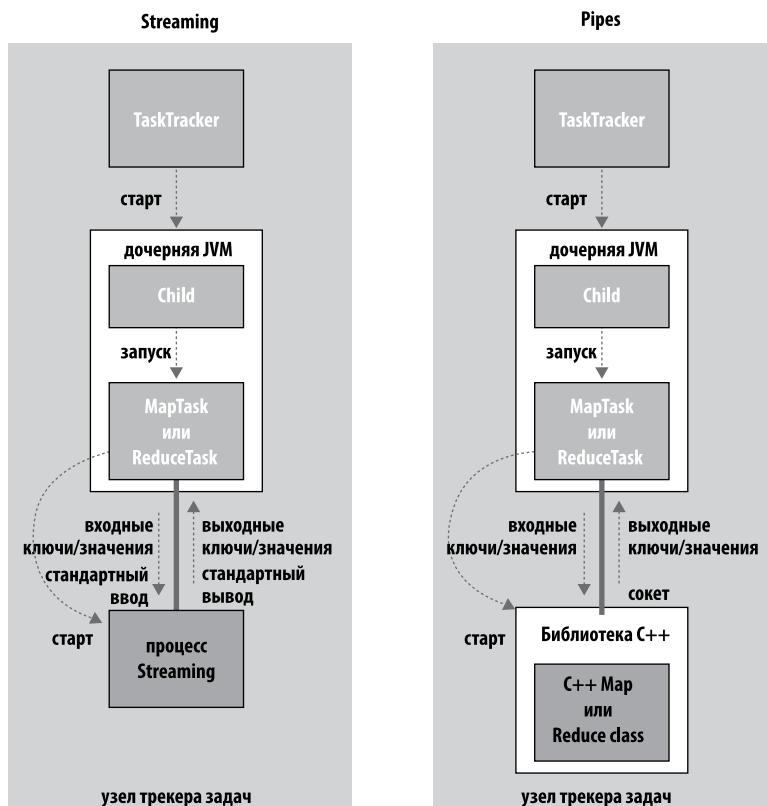


Рис. 6.2. Связь исполняемых модулей Streaming и Pipes с трекером задач и его дочерним процессом

КАК ОЦЕНИВАЕТСЯ ПРОГРЕСС В MAPREDUCE?

Прогресс заданий не всегда поддается объективной оценке, но, по крайней мере, Hadoop знает, что задача не стоит на месте. Например, задача, записывающая выходные данные, работает, даже если прогресс не выражается в процентах от общего количества выводимых записей (просто потому, что оно не известно заранее).

Информация о прогрессе важна, так как Hadoop не считает такую задачу, о которой она поступает, сбойной. Прогрессом считается выполнение любых операций из следующего списка:

- Чтение входных данных (в задаче отображения или свертки).
- Запись входных данных (в задаче отображения или свертки).
- Назначение описания состояния (метод setStatus() объекта Reporter).
- Увеличение счетчика (метод incrCounter() объекта Reporter).
- Вызов метода progress() объекта Reporter.

С задачей также связывается набор счетчиков, подсчитывающих различные события в процессе выполнения задачи (пример приведен в разделе «Тестовый запуск», с. 57) — как встроенных в инфраструктуру (например, для подсчета количества выходных записей отображений), так и определяемых пользователем.

Когда задача сообщает о прогрессе, она устанавливает флаг, указывающий, что информация об изменении состояния должна быть передана трекеру задач. Отдельный программный поток проверяет флаг каждые три секунды, и если он установлен — оповещает трекер задач о текущем состоянии задачи. При этом трекер задач отправляет периодический сигнал трекеру задач каждые пять секунд (минимум — фактический интервал зависит от размера кластера; в больших кластерах используются более длинные интервалы). При вызове передается состояние всех задач, выполняемых трекером задач. Данные счетчиков передаются реже, потому что они создают относительно высокую нагрузку на канал связи.

Трекер задач объединяет данные всех обновлений для формирования глобального представления о состоянии всех выполняемых заданий и входящих в них задач. Наконец, как упоминалось ранее, объект `Job` получает обновленную информацию состояния, ежесекундно опрашивая трекер заданий. Клиенты также могут использовать метод `getStatus()` объекта `Job` для получения экземпляра `JobStatus`, содержащего полную информацию о состоянии задания.

Схема использования этих методов представлена на рис. 6.3.

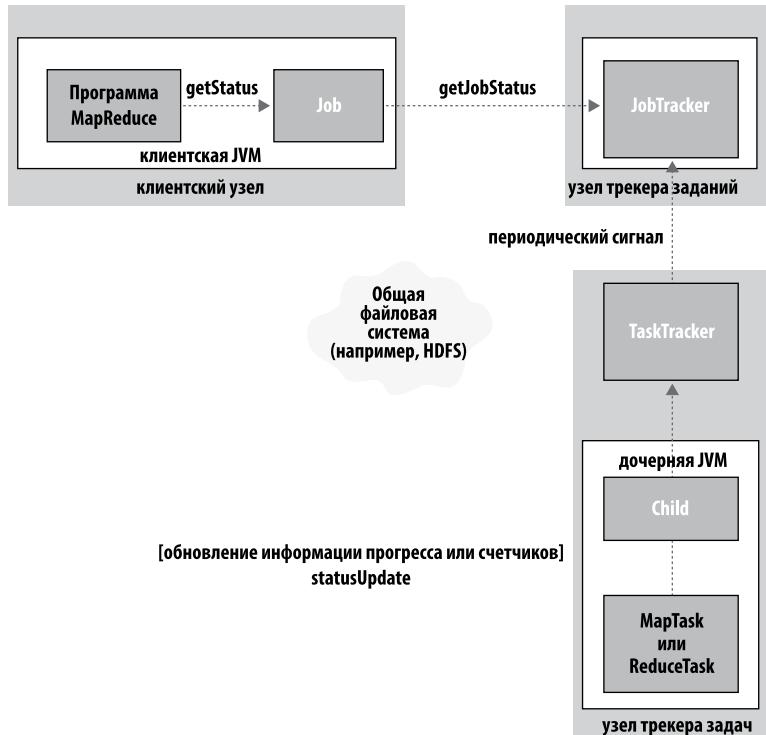


Рис. 6.3. Распространение обновленной информации состояния в системе MapReduce 1

Завершение заданий

Когда трекер заданий получает оповещение о завершении последней задачи задания (специальной задачи деинициализации), он устанавливает для задания признак состояния «успешное завершение». Затем при запросе информации состояния объект `Job` узнает о том, что задание успешно завершилось, выводит сообщение для пользователя и возвращается из метода `waitForCompletion()`. В этот момент на консоль выводится статистика задания и счетчики.

Трекер заданий также отправляет HTTP-оповещение о состоянии задания, если соответствующая возможность включена в настройке. Оповещение настраивается клиентами, желающими получать информацию, при помощи свойства `job.end.notification.url`.

Наконец, трекер задачий удаляет рабочее состояние задания и приказывает трекерам сделать то же самое (например, при этом удаляется весь промежуточный вывод).

YARN (MapReduce 2)

В очень больших кластерах, состоящих из 4000 узлов и более, описанная в предыдущем разделе система MapReduce начинает сталкиваться с ограничивающими факторами масштабируемости. В 2010 году группа из Yahoo! занялась проектированием следующего поколения MapReduce. Результатом этой работы стала система YARN, сокращение от «Yet Another Resource Negotiator» (или для любителей рекурсивных сокращений — «YARN Application Resource Negotiator»)¹.

YARN решает проблемы масштабируемости «классической» модели MapReduce за счет распределения обязанностей трекера заданий. В классической модели трекер заданий занимается как планированием заданий (распределением задач по трекерам задач), так и отслеживанием их прогресса (отслеживание задач, перезапуск сбойных или медленно выполняемых задач, учет служебной информации — например, хранение значений счетчиков).

YARN разделяет эти две роли, поручая их двум независимым демонам: *менеджер ресурсов* управляет использованием ресурсов в кластере, а *контроллер приложений* управляет жизненным циклом приложений, выполняемых в кластере. Контроллер приложений согласует использование ресурсов кластера с менеджером ресурсов; результаты согласования представляются в виде контейнеров, имеющих определенные ограничения по памяти, после чего процессы, зависящие от приложений, выполняются в этих контейнерах. За контейнерами наблюдают менеджеры узлов, работающие в узлах кластера; они следят за тем, чтобы приложение не использовало больше ресурсов, чем ему было выделено².

В отличие от трекера задач, у каждого экземпляра приложения — в нашем случае у задания MapReduce — имеется выделенный контроллер приложений, который работает в течение выполнения этого приложения. Описанная модель ближе к исходному описанию MapReduce в статье Google, в которой показано, как процесс-контроллер запускается для координации задач отображения и свертки, выполняемых группой рабочих процессов.

¹ О побудительных причинах и ходе разработки YARN рассказано в статье Аруна С. Марти (Arun C. Murthy) «The Next Generation of Apache Hadoop MapReduce».

² На момент написания книги память была единственным управляемым ресурсом, а менеджеры узлов уничтожали любой контейнер, превысивший выделенные лимиты памяти.

Как видно из описания, YARN имеет более общую природу, чем MapReduce. Собственно, MapReduce — всего лишь одна разновидность приложений YARN. Есть и другие — например, распределенный командный процессор, который может выполнять сценарий на группе узлов в кластере. В настоящее время активно разрабатываются другие приложения YARN (некоторые из них перечислены на странице <http://wiki.apache.org/hadoop/PoweredByYarn>). Элегантность архитектуры YARN заключается в том, что разные приложения YARN могут сосуществовать в одном кластере (скажем, приложение MapReduce может выполняться одновременно с приложением MPI), а это обстоятельство имеет огромные преимущества в отношении управляемости и эффективного использования кластера.

Более того, пользователи даже могут запускать разные версии MapReduce в одном кластере YARN, что дополнительно упрощает процесс обновления MapReduce. (Учтите, что некоторые компоненты MapReduce — например, сервер истории заданий и обработчик тасовки, а также сама система YARN — должны обновляться в масштабах кластера.)

В работе MapReduce на базе YARN задействовано больше сущностей, чем в классической модели¹:

- Клиент, отправляющий задание MapReduce.
- Менеджер ресурсов YARN, координирующий операции выделения вычислительных ресурсов в кластере.
- Менеджеры узлов YARN, запускающие вычислительные контейнеры на машинах кластера и наблюдающие за ними.
- Контроллер приложений MapReduce, координирующий выполнение задач в заданиях MapReduce. Контроллер приложений и задачи MapReduce выполняются в контейнерах, которые планируются менеджером ресурсов и управляются менеджерами узлов.
- Распределенная файловая система (обычно HDFS — см. главу 3), используемая для передачи файлов задания между другими сущностями.

Процесс выполнения задания показан на рис. 6.4 и описан в следующих разделах.

¹ В этом разделе не рассматривается демон сервера истории заданий (для хранения данных истории заданий) и дополнительная служба обработчика тасовки (для предоставления вывода отображений задачам свертки). В классической модели MapReduce они являются частью трекера заданий и трекера задач (соответственно), но в YARN это независимые компоненты.

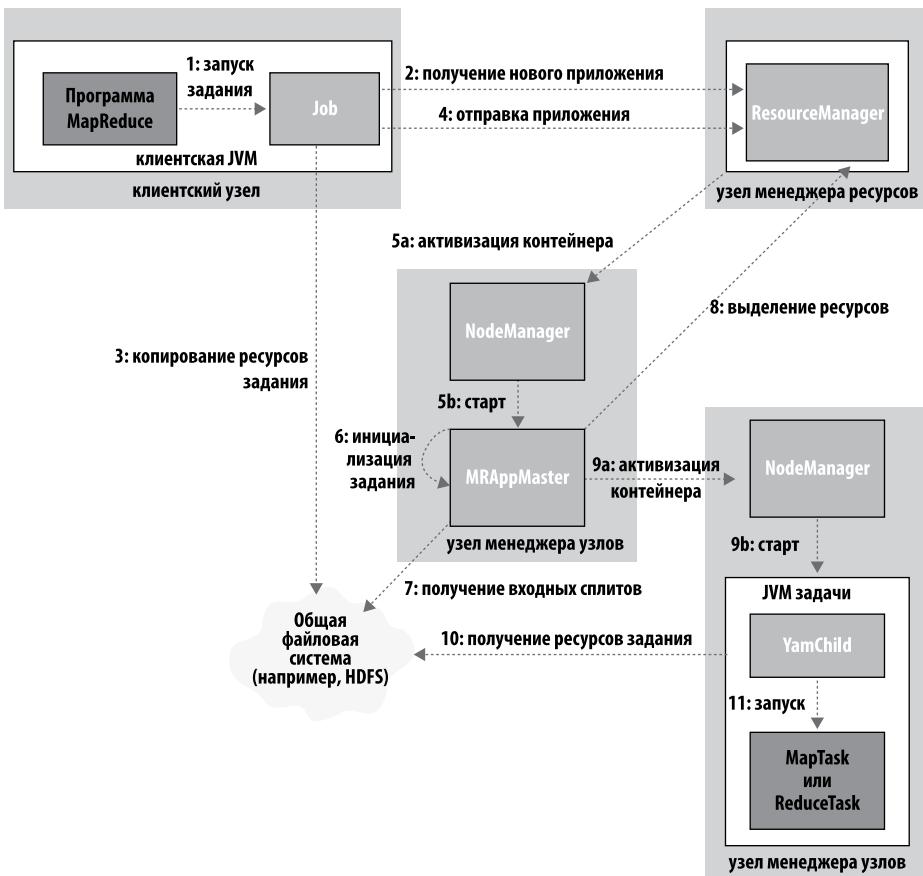


Рис. 6.4. Выполнение заданий MapReduce на базе YARN

Отправка заданий

Отправка заданий осуществляется в MapReduce 2 с использованием того же пользовательского API, что и в MapReduce 1 (шаг 1). MapReduce 2 содержит реализацию `ClientProtocol`, которая активизируется заданием свойству `mapreduce.framework.name` значения `yarn`. Процесс отправки имеет много общего с классической реализацией. Новый идентификатор задания выдается менеджером ресурсов (а не трекером заданий), хотя в терминологии YARN он называется идентификатором приложения (шаг 2). Клиент задания проверяет выходную спецификацию задания; вычисляет входные сплиты (хотя их можно генерировать в кластере, используя параметр `yarn.app.mapreduce.am.compute-splits-in-cluster`; эта возможность может быть полезна в заданиях с большим количеством сплитов); и копирует ресурсы задания (включая JAR-файл задания, конфигурацию и информацию

сплитов) в HDFS (шаг 3). Наконец, задание отправляется на выполнение вызовом метода `submitApplication()` менеджера ресурсов (шаг 4).

Инициализация заданий

Когда менеджер ресурсов получает вызов своего метода `submitApplication()`, он передает запрос планировщику. Планировщик выделяет контейнер, а менеджер ресурсов запускает процесс контроллера приложений под управлением менеджера узлов (шаги 5a и 5b).

Контроллер приложений для заданий MapReduce представляет собой Java-приложение с главным классом `MRAppMaster`. Он инициализирует задание, создавая ряд служебных объектов для отслеживания прогресса заданий; информацию о прогрессе и завершении он будет получать от задач (шаг 6). Затем он получает входные сплиты, вычисленные в клиенте из общей файловой системы (шаг 7). Для каждого сплита создается объект задачи отображения, а также объекты задач свертки, количество которых определяется свойством `mapreduce.job.reduces`.

Далее контроллер приложений решает, как выполнять задачи, образующие задание MapReduce. Если задание невелико, контроллер приложений может запустить задачи на той же виртуальной машине Java, на которой работает он сам. Это происходит, когда, по его оценке, затраты на выделение и запуск задач в новых контейнерах перевесят пользу от их параллельного выполнения (по сравнению с их последовательным выполнением на одном узле) — здесь ситуация отличается от MapReduce 1, где малые задания никогда не выполняются на одном трекере задач. Такие задачи называются *суперзадачами* (*ubertasks*).

Какое задание считается малым? По умолчанию малым считается задание, которое содержит менее 10 задач отображения, только одну задачу свертки и размер входных данных которого меньше размера одного блока HDFS. (Вы можете изменить эти параметры для задания при помощи свойств `mapreduce.job.ubertask.maxmaps`, `mapreduce.job.ubertask.maxreduces` и `mapreduce.job.ubertask.maxbytes`.) Также можно полностью запретить выполнение малых заданий на одном узле, задав свойству `mapreduce.job.ubertask.enable` значение `false`.)

Перед выполнением каких-либо задач вызывается метод подготовки задания (из экземпляра `OutputCommitter`), который создает выходной каталог задания. В отличие от модели MapReduce 1, в которой он вызывался специальной задачей, запускаемой трекером задач, в реализации YARN метод вызывается напрямую контроллером приложений.

Назначение задач

Если задание не подходит для выполнения на одном узле, контроллер приложений запрашивает у менеджера ресурсов контейнеры для всех задач отображения

и свертки в задании (шаг 8). Все запросы, совмещаемые с вызовами передачи периодических сигналов, включают в себя информацию о локальности данных каждой задачи отображения, и в частности о хостах и соответствующих сегментах, на которых находится входной сплит. Планировщик использует эту информацию для принятия решений (как и планировщик трекера заданий). Он пытается разместить задачи на узлах, локальных по отношению к данным, а если это невозможно, планировщик предпочитает сегментную локальность нелокальному размещению.

В запросах также указываются требования к памяти задач. По умолчанию задачам отображения и свертки выделяются 1024 Мбайт памяти, но эту величину можно настроить при помощи свойств `mapreduce.map.memory.mb` и `mapreduce.reduce.memory.mb`.

Способ распределения памяти отличается от модели MapReduce 1, в которой трекеры задач имеют фиксированное количество «слотов», заданное при настройке кластера, и каждая задача выполняется в одном слоте. Выделяемая слоту память ограничена, причем эта величина также фиксирована в пределах кластера, что ведет как к проблемам недозагруженности (другие ожидающие задачи не могут распоряжаться неиспользуемой памятью), так и к сбоям приложений, когда задача не может завершиться из-за нехватки памяти для нормального выполнения.

В YARN управление ресурсами осуществляется с большей гранулярностью, что позволяет избежать обеих проблем. В частности, приложения могут запросить блок памяти в диапазоне от минимального до максимального значения, причем размер блока должен быть кратен минимальному. По умолчанию операции выделения памяти зависят от планировщика; для планировщика Capacity Scheduler минимум по умолчанию составляет 1024 Мбайт (задается свойством `yarn.scheduler.capacity.minimum-allocation-mb`), а максимум по умолчанию — 10240 МБ (задается свойством `yarn.scheduler.capacity.maximum-allocation-mb`). Таким образом, задачи могут запрашивать блоки памяти от 1 до 10 Гбайт (включительно) с приращениями 1 Гбайт (при необходимости планировщик округляет до ближайшего кратного), задавая соответствующие значения свойств `mapreduce.map.memory.mb` и `mapreduce.reduce.memory.mb`.

Выполнение задач

После того, как планировщик менеджера ресурсов назначит задаче контейнер, контроллер приложений запускает контейнер на выполнение; для этого он связывается с менеджером узлов (шаги 9a и 9b). Задача выполняется Java-приложением с главным классом `YarnChild`. До запуска задачи оно локализует все необходимые ресурсы, включая конфигурацию задания и JAR-файл, а также файлы из распределенного кэша (шаг 10). Наконец, задача отображения или свертки запускается на выполнение (шаг 11).

`YarnChild` выполняется в выделенной виртуальной машине Java — по той же причине, по которой трекеры задач создают новые JVM для задач в MapReduce 1: для изоляции пользовательского кода от системных демонов. Однако в отличие от MapReduce 1, YARN не поддерживает повторное использование JVM, так что каждая задача работает в новой виртуальной машине.

Программы Streaming и Pipes работают по тому же принципу, что и MapReduce 1. `YarnChild` запускает процесс Streaming или Pipes и взаимодействует с ним через стандартный ввод/вывод или сокет (соответственно), как показано на рис. 6.2 (не считая того, что дочерний процесс и субпроцессы выполняются на менеджерах узлов, а не на трекерах задач).

Обновления состояния

При работе под управлением YARN задача передает информацию о своем прогрессе и состоянии (включая счетчики) своему контроллеру приложений, содержащему сводную информацию о задании, каждые три секунды по связующему интерфейсу. Процесс изображен на рис. 6.5. Сравните с моделью MapReduce 1, в которой обновленная информация передавалась от дочернего процесса через трекер задач трекеру заданий для обобщения.

Клиент опрашивает контроллер приложений каждую секунду (периодичность задается свойством `mapreduce.client.progressmonitor.pollinterval`); полученная обновленная информация о прогрессе обычно выводится для просмотра пользователем.

В MapReduce 1 в веб-интерфейсе трекера заданий выводился список выполняемых заданий с информацией об их прогрессе. В YARN веб-интерфейс менеджера ресурсов выводит список всех приложений со ссылками на веб-интерфейсы соответствующих контроллеров приложений, в которых можно получить более подробную информацию о заданиях MapReduce (включая информацию о прогрессе).

Завершение заданий

Кроме периодических запросов к контроллеру приложений, клиент каждые пять секунд проверяет, не завершилось ли задание; для этого он вызывает метод `waitForCompletion()` объекта `Job`. Интервал опроса задается свойством конфигурации `mapreduce.client.completion.pollinterval`.

Также поддерживается оповещение о завершении задач обратными вызовами HTTP, как и в MapReduce 1. В MapReduce 2 обратный вызов инициируется контроллером приложений.

При завершении задания контроллер приложения и контейнеры задач удаляют свое рабочее состояние, после чего вызывается метод деинициализации объекта `OutputCommitter`. Информация о задании архивируется сервером истории заданий, где пользователи при желании смогут проанализировать ее.

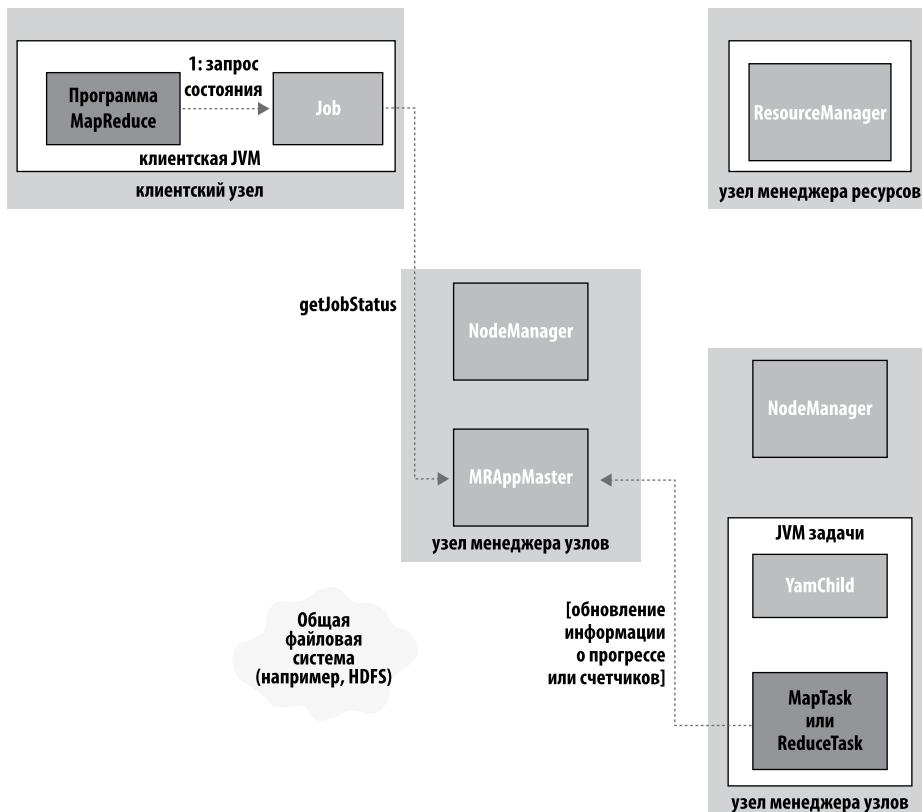


Рис. 6.5. Распространение обновленной информации состояния в системе MapReduce 2

Сбои

В реальном мире в пользовательским коде встречаются ошибки, процессы аварийно завершаются, а на машинах происходят сбои. Одно из важнейших достоинств Hadoop — способность выдерживать такие сбои, давая вашим заданиям возможность завершиться.

Сбои в классической модели MapReduce

В исполнительной среде MapReduce различаются три режима сбоев: сбой выполняемой задачи, сбой трекера задач и сбой трекера заданий.

Сбой задачи

Начнем со сбоев дочерних задач. Наиболее частая причина таких сбоев — иницирование исключения времени выполнения в пользовательском коде задачи отображения или свертки. В этом случае дочерняя JVM перед завершением работы сообщает об ошибке родительскому трекеру задач.

Ошибка в конечном итоге регистрируется в пользовательском журнале. Трекер задач помечает попытку выполнения задачи как сбойную и освобождает слот для выполнения другой задачи.

Для задач Streaming, если процесс Streaming завершается с ненулевым кодом завершения, он помечается как сбойный. Это поведение определяется свойством `stream.non.zero.exit.is.failure` (по умолчанию равно `true`).

Другой причиной сбоя может стать внезапная остановка дочерней JVM — например, из-за ошибки JVM, которая вызывает аварийное завершение работы JVM в конкретном наборе обстоятельств, выявленном пользовательским кодом MapReduce. В этом случае трекер задач отмечает, что процесс завершился, и помечает попытку как сбойную.

Проблема «зависших» задач решается иначе. Трекер задач замечает, что он давно не получал обновленной информации о прогрессе, и переходит к пометке задачи как сбойной. По истечении указанного периода дочерний процесс JVM будет автоматически уничтожен¹. Период тайм-аута, после которого задача считается сбойной, обычно составляет 10 минут, но он может настраиваться на уровне заданий (или на уровне кластеров) — нужное значение в миллисекундах задается свойству `mapred.task.timeout`.

Если задать нулевую продолжительность тайм-аута, то тайм-аут будет отключен, а долго выполняемые задачи никогда не будут помечаться как сбойные. В этом случае «зависшая» задача никогда не освободит свой слот, что со временем может

¹ Если «зависнет» процесс Streaming или Pipes, трекер задач уничтожит его (вместе с запущившей его виртуальной машиной Java) только в следующих обстоятельствах: либо свойству `mapred.task.tracker.task-controller` задано значение `org.apache.hadoop.mapred.LinuxTaskController`, либо используется контроллер задач по умолчанию (`org.apache.hadoop.mapred.DefaultTaskController`), а в системе доступна команда `setsid system` (так что дочерняя JVM и все запущенные ею процессы находятся в одной группе процессов). В любом случае в системе начинают накапливаться «бесхозные» процессы Streaming или Pipes, что отрицательно скажется на ее производительности.

привести к замедлению работы кластера. Соответственно, отключать тайм-аут не рекомендуется — лучше следить за тем, что задача периодически сообщает о прогрессе выполнения (см. «Как оценивается прогресс в MapReduce?», с. 263).

Получив оповещение о сбое попытки выполнения задачи (по периодическому сигналу трекера задач), трекер заданий заново планирует выполнение задачи. Трекер задач старается избегать повторного планирования задачи на том трекере, на котором ранее в ней произошел сбой. Кроме того, если при выполнении задачи сбой происходит четыре раза (или более), повторные попытки делаться не будут. Максимальное количество попыток выполнения задачи определяется свойством `mapred.map.max.attempts` для задач отображения и свойством `mapred.reduce.max.attempts` для задач свертки. По умолчанию при выполнении задачи происходит до четырех сбойных попыток (или столько раз, сколько задано в конфигурации), после чего сбойным считается само задание.

В некоторых приложениях отмена задания при сбоях нескольких задач нежелательна, так как результаты задания могут быть полезными даже при наличии сбоев. В таких ситуациях для задания можно установить максимальный процент задач, которые могут завершиться сбоем без сбоя всего задания. Задачами отображения и свертки можно управлять по отдельности при помощи свойств `mapred.max.map.failures.percent` и `mapred.max.reduce.failures.percent`.

Попытка выполнения задачи может быть *уничтожена*, что не считается сбоем. Причиной уничтожения может быть дублирование при спекулятивном выполнении (за дополнительной информацией обращайтесь к разделу «Спекулятивное выполнение», с. 288) или сбой на трекере задач, после которого трекер заданий помечает все попытки выполнения задач на нем как уничтоженные. Уничтоженные попытки выполнения задачи не учитываются в количестве попыток (заданных свойствами `mapred.map.max.attempts` и `mapred.reduce.max.attempts`), потому что сама задача в уничтожении не виновата.

Пользователи также могут уничтожать попытки выполнения задач или объявлять их сбойными из веб-интерфейса или командной строки (чтобы получить дополнительную информацию, введите команду `hadoop job`). Задания также могут уничтожаться аналогичным образом.

Сбой трекера задач

Если на трекере задач происходит сбой, из-за которого он перестает работать или работает очень медленно, трекер перестает отправлять периодические сигналы трекеру заданий (или отправляет их очень редко). Трекер заданий замечает отсутствие активности трекера задач, если он не получает ни одного периодического сигнала в течение 10 минут (значение в миллисекундах настраивается `mapred.tasktracker.expiry.interval`), и исключает его из пула трекеров задач, на которых

планируется выполнение задач. Трекер заданий организует повторное выполнение задач отображения из незавершенных заданий, успешно выполненных на этом трекере, так как их промежуточный вывод в локальной файловой системе сбийного трекера задач может быть недоступным для задач свертки. Все незавершенные задачи также планируются заново.

Трекер заданий может занести трекер задач в «черный список», даже если тот продолжает работать. Если на некотором трекере сбой происходит более чем в четырех задачах одного задания (количество определяется свойством `mapred.max.tracker.failures`), трекер заданий отмечает этот факт как неисправность (`fault`). Трекер задач заносится в «черный список» при достижении некоторого минимального порога неисправностей (значение по умолчанию равно 4, задается свойством `mapred.max.tracker.blacklists`), значительно превышающее среднее количество неисправностей трекеров задач в кластере.

Трекерам задач из «черного списка» не назначаются задачи, но они продолжают взаимодействовать с трекером задач. Неисправности со временем аннулируются (одна в день), так что трекеры задач смогут снова получать задания, просто продолжая работать. Если же проблема была исправлена (скажем, заменой оборудования), трекер задач исключается из «черного списка» трекера заданий после перезапуска и повторного присоединения к кластеру.

Сбой трекера заданий

Сбои трекеров заданий образуют самую серьезную категорию сбоев. В Hadoop не существует механизма преодоления сбоев трекеров задач, так что в этом случае сбоями становятся все выполняемые задания. Впрочем, этот режим сбоев маловероятен, потому что вероятность сбоя одной конкретной машины низка. К счастью, в YARN ситуация улучшается, так как одной из целей проектирования YARN было устранение единых точек отказа в MapReduce.

После перезапуска трекера заданий все задания, выполнявшиеся на момент его остановки, должны быть отправлены заново. Существует параметр конфигурации, пытающийся восстановить все выполняемые задания (`mapred.jobtracker.restart.recover`, по умолчанию восстановление отключено), однако он работает недостаточно надежно, поэтому использовать его не рекомендуется.

Сбои в YARN

Для программ MapReduce, работающих в YARN, необходимо учесть возможные сбои задач, контроллера приложений, менеджера узлов и менеджера ресурсов.

Сбой задачи

Сбой выполняемой задачи сходен с классическим случаем. Информация об исключениях времени выполнения и внезапных завершениях JVM передается контроллеру приложений, и попытка выполнения задачи помечается как сбойная. Аналогичным образом контроллер приложений узнает о «зависании» задач по отсутствию периодического сигнала по связующему каналу (продолжительность тайм-аута задается свойством `mapreduce.task.timeout`), а попытка выполнения задачи также помечается как сбойная.

Для определения сбойных задач используются те же свойства конфигурации, что и в классическом случае: задача помечается как сбойная после четырех попыток (количество определяется свойством `mapreduce.map.maxattempts` для задач отображения и свойством `mapreduce.reduce.maxattempts` для задач свертки). Задание будет считаться сбоящим при сбое более `mapreduce.map.failures.maxpercent` процентов задач отображения в задании или `mapreduce.reduce.failures.maxpercent` процентов задач свертки.

Сбой контроллера приложений

По аналогии с классической моделью MapReduce, дающей задачам несколько попыток успешного выполнения (на случай сетевых или аппаратных сбоев), приложения YARN тоже несколько раз проверяются на сбои. По умолчанию приложения помечаются как сбоящими в случае однократного сбоя, но порог можно увеличить при помощи свойства `yarn.resourcemanager.am.max-retries`.

Контроллер приложений отправляет периодические сигналы менеджеру ресурсов; в случае сбоя контроллера приложений менеджер ресурсов обнаруживает сбой и запускает новый экземпляр контроллера в новом контейнере (находящемся под управлением менеджера узлов). Контроллер приложений MapReduce может восстановить состояние задач, уже запущенных (сбоящим) приложением, чтобы их не пришлось выполнять заново. По умолчанию восстановление отключено, так что сбоящие контроллеры приложений заново запустят свои задачи, но восстановление можно включить, задав свойству `yarn.app.mapreduce.am.job.recovery.enable` значение `true`.

Клиент периодически запрашивает у контроллера приложений информацию о прогрессе; если в контроллере приложений происходит сбой, клиент должен найти новый экземпляр. В процессе инициализации задания клиент запрашивает у менеджера ресурсов адрес контроллера приложений и кэширует его, чтобы не перегружать менеджер ресурсов запросами каждый раз, когда ему потребуется опросить контроллер приложений. Но если в контроллере приложений происходит сбой, попытка получения обновленной информации состояния завершится

неудачей и клиент снова обратится к менеджеру ресурсов за адресом нового контроллера приложений.

Сбой менеджера узлов

Если сбой происходит на менеджере узлов, то он перестает отправлять периодические сигналы менеджеру ресурсов и исключается из пула доступных узлов последнего. Если за некоторый промежуток времени не поступит ни одного периодического сигнала, менеджер ресурсов считает менеджер узлов сбоящим; величина этого промежутка определяется свойством `yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms`, по умолчанию равным 600 000 (10 минутам).

Все задачи или контроллеры приложений, выполняемые на сбояном менеджере узлов, восстанавливаются с использованием механизмов, описанных в двух предыдущих разделах.

При высоком количестве сбоев в приложении менеджеры узлов могут быть занесены в «черный список» контроллером приложений. Для MapReduce контроллер приложений пытается заново спланировать задачи на других узлах, если на менеджере узлов произошло более трех сбоев задач. Пользователь может задать пороговую величину при помощи свойства `mapreduce.job.maxtaskfailures.per.tracker`.



Менеджер ресурсов не работает с «черными списками» (на момент написания книги), поэтому задачи новых заданий могут планироваться на сбоях узлах, даже если они были занесены в «черный список» контроллером приложений, управляющим предыдущим заданием.

Сбой менеджера ресурсов

Сбой менеджера ресурсов приводит к серьезным последствиям, потому что без него не могут запускаться ни задачи, ни контейнеры задач. Менеджер ресурсов изначально проектировался для восстановления из аварийных ситуаций, с использованием механизма контрольных точек для долгосрочного сохранения состояния (хотя на момент написания книги новейшая версия еще не содержала полноценной реализации).

После сбоя запускается новый экземпляр менеджера ресурсов (это делает администратор), который восстанавливает сохраненное состояние, включающее менеджеры узлов системы, а также работающие приложения. (Учтите, что задачи не являются частью состояния менеджера ресурсов, так как они находятся под управлением контроллера приложений.)

Механизм сохранения, используемый менеджером ресурсов, настраивается при помощи свойства `yarn.resourcemanager.store.class`. По умолчанию используется

реализация `org.apache.hadoop.yarn.server.resourcemanager.recovery.MemStore`, которая хранит данные в памяти, а следовательно, не обеспечивает высокой доступности. Однако сейчас разрабатывается система хранения данных на базе ZooKeeper, которая в будущем будет поддерживать надежное восстановление после сбоев менеджера ресурсов.

Планирование заданий

В ранних версиях Hadoop использовался очень простой метод планирования пользовательских заданий: они выполнялись в порядке отправки с использованием FIFO-планировщика. Обычно каждое задание использовало весь кластер, так что заданиям приходилось дожидаться своей очереди. Хотя совместное использование кластеров открывает выдающиеся возможности по предоставлению масштабных ресурсов большому количеству пользователей, проблема распределения ресурсов между пользователями требует более качественного планировщика. Реальные задания должны завершаться за разумное время, но при этом позволять пользователям, выдающим небольшие несистематические запросы, получать результаты за разумное время.

Позднее была добавлена возможность назначения приоритетов заданиям при помощи свойства `mapred.job.priority` или метода `setJobPriority()` объекта `JobClient` (в обоих случаях используются значения `VERY_HIGH`, `HIGH`, `NORMAL`, `LOW` или `VERY_LOW`). Выбирая следующее задание для выполнения, планировщик заданий останавливается на задании с наивысшим приоритетом. Однако FIFO-планировщик не поддерживает вытеснение, так что высокоприоритетное задание может быть заблокировано низкоприоритетным, долго выполняемым заданием, запущенным до планирования высокоприоритетного задания.

В поставку MapReduce в Hadoop включена подборка планировщиков. По умолчанию в MapReduce 1 используется исходный FIFO-планировщик на базе очереди, а также доступны многопользовательские планировщики Fair Scheduler и Capacity Scheduler.

Fair Scheduler

Планировщик Fair Scheduler стремится выделить каждому пользователю «справедливую» долю кластерной вычислительной мощности. Если выполняется всего одно задание, в его распоряжении оказывается весь кластер. По мере поступления новых заданий свободные слоты задач выделяются заданиям таким образом, чтобы

каждый пользователь получил справедливую долю ресурсов кластера. Короткие задания, принадлежащие одному пользователю, будут завершаться за разумное время даже во время выполнения большого задания другого пользователя, при этом большое задание тоже будет двигаться вперед.

Задания помещаются в пулы; по умолчанию каждому пользователю выделяется собственный пул. Пользователь, отправивший больше заданий, в среднем не получит в свое распоряжение больше ресурсов кластера. Также можно определять пулы с гарантированными минимальными возможностями по слотам отображения и свертки и задавать весовые коэффициенты для пулов.

Планировщик Fair Scheduler поддерживает вытеснение. Таким образом, если пул не получил свою справедливую долю ресурсов за определенный период времени, планировщик уничтожит задачи в перегруженных пулах, чтобы выделить больше слотов недогруженным пулам.

Fair Scheduler не входит в стандартную конфигурацию. Чтобы включить его, разместите JAR-файл в пути к классам Hadoop, скопировав его из каталога Hadoop *contrib/fairscheduler* в каталог *lib*. Затем задайте свойству *mapred.jobtracker.taskScheduler* значение *org.apache.hadoop.mapred.FairScheduler*.

Планировщик Fair Scheduler будет работать без дополнительной настройки, но, чтобы в полной мере использовать его возможности и научиться его настраивать (а также пользоваться веб-интерфейсом), обращайтесь к файлу *README* в каталоге *src/contrib/fairscheduler* дистрибутива.

Capacity Scheduler

Планировщик Capacity Scheduler использует несколько иной подход к много-пользовательскому планированию. Кластер образуется из нескольких очередей (по аналогии с пулами Fair Scheduler), которые могут складываться в иерархию (таким образом, очередь может быть родителем по отношению к другой очереди), при этом каждой очереди выделяется определенная мощность. В этом Capacity Scheduler похож на Fair Scheduler, не считая того, что в каждой очереди задания планируются с использованием дисциплины FIFO (с приоритетами). В сущности, Capacity Scheduler позволяет пользователям или организациям (определенным с использованием очередей) моделировать отдельный кластер MapReduce с FIFO-планированием для каждого пользователя или организации. Вместе с тем планировщик Fair Scheduler (который также поддерживает планирование заданий FIFO в пулах, что приближает его к Capacity Scheduler) обеспечивает справедливое распределение ресурсов внутри каждого пула, а выполняемые задания совместно используют ресурсы пула.

Тасовка и сортировка

MapReduce гарантирует, что входные данные каждой задачи свертки отсортированы по ключу. Процесс выполнения сортировки системой (и передачи выходных данных отображений на вход сверток) называется *тасовкой* (*shuffle*)¹. В этом разделе мы разберемся в том, как работает тасовка, так как хотя бы базовое представление об этом процессе пригодится в ходе оптимизации программ MapReduce. В коде тасовки постоянно вносятся всевозможные улучшения и усовершенствования, поэтому в следующем описании многие подробности опущены по необходимости (и могут изменяться со временем; описание приведено для версии 0.20). Во многих отношениях тасовка является «сердцем» MapReduce, и именно здесь происходит самое интересное.

На стороне отображения

Выходные данные, генерируемые функцией отображения, не записываются прямо на диск. Процесс более сложен; в нем задействована буферизация записи в памяти и предварительная сортировка по соображениям эффективности. На рис. 6.6 показано, что при этом происходит.

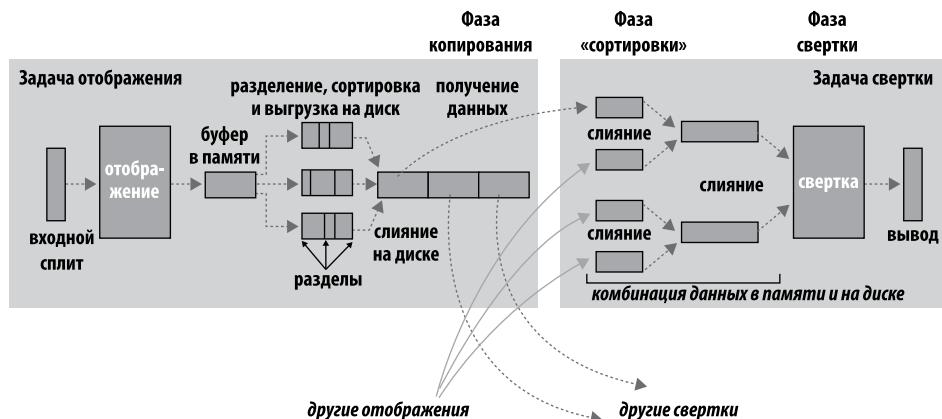


Рис. 6.6. Тасовка и сортировка в MapReduce

¹ Вообще говоря, термин «тасовка» неточен, потому что в некоторых контекстах он относится только к той части процесса, в которой выходные данные отображений передаются задачам свертки. В этом разделе мы будем считать, что он обозначает весь процесс — от той точки, в которой отображение производит выходные данные, до точки, в которой свертка получает ввод.

У каждой задачи отображения имеется циклический буфер, в который она записывает свои выходные данные. По умолчанию размер буфера составляет 100 Мбайт, однако размер может настраиваться изменением свойства `io.sort.mb`. Когда содержимое буфера достигает определенного порога (`io.sort.spill.percent`; значение по умолчанию равно 0.80, то есть 80%), фоновый программный поток начинает выгружать содержимое на диск. Выходные данные отображения продолжают записываться в буфер, пока не начнется выгрузка, но если буфер в это время заполнится, то отображение блокируется до завершения выгрузки. Выгрузка осуществляется по циклическому принципу в каталоги, заданные свойством `mapred.local.dir` (находящиеся в каталоге задания).

Перед записью на диск программный поток сначала делит данные на разделы, соответствующие сверткам, которым они в конечном итоге будут переданы. В каждом разделе фоновый программный поток сортирует данные по ключу, а если определена комбинирующая функция — она выполняется для выходных данных сортировки. Выполнение комбинирующей функции делает вывод отображений более компактным, что уменьшает объем данных, записываемых на локальный диск и передаваемых свертке.

Каждый раз, когда буфер памяти достигает порога выгрузки, создается новый файл выгрузки. Таким образом, после того как задача отображения выведет свою последнюю выходную запись, файлов выгрузки может быть несколько. Перед завершением задачи файлы выгрузки сливаются в один разделенный и отсортированный выходной файл. Максимальное количество одновременно сливаемых потоков определяется свойством конфигурации `io.sort.factor`; значение по умолчанию равно 10.

При наличии минимум трех файлов выгрузки (количество определяется свойством `min.num.spills.for.combine`) комбинирующая функция выполняется снова перед записью выходного файла. Напомним, что комбинирующие функции могут выполняться повторно для вывода, не оказывая влияния на конечный результат. Для одного-двух файлов выгрузки потенциальное сокращение размера вывода не компенсирует затрат на выполнение комбинирующей функции, поэтому она повторно не выполняется.

Часто выходные данные отображений стоит сжимать при записи на диск, потому что сжатие ускоряет запись, экономит дисковое пространство и сокращает объем данных, передаваемых свертке. По умолчанию выходные данные не сжимаются, но сжатие легко включается заданием свойству `mapred.compress.map.output` значения `true`. Используемая библиотека сжатия задается свойством `mapred.map.output.compression.codec`; за дополнительной информацией о форматах сжатия обращайтесь к разделу «Сжатие», с. 128.

Свертки получают доступ к разделам выходного файла через HTTP. Максимальное количество рабочих потоков, используемых для предоставления разделов

файла, определяется свойством `tasktracker.http.threads`; настройка действует на уровне трекера задач, а не на уровне слота задачи отображения. Значение по умолчанию 40 иногда увеличивается для очень больших кластеров с крупномасштабными заданиями.

В MapReduce 2 это свойство не действует, потому что максимальное количество программных потоков задается автоматически в зависимости от количества процессоров на машине. (MapReduce 2 использует инфраструктуру Netty, у которой по умолчанию количество программных потоков может быть вдвое больше количества процессоров.)

На стороне свертки

Обратимся к другой стороне процесса — свертке. Выходной файл отображения находится на локальном диске машины, выполнившей задачу отображения (хотя вывод отображений всегда записывается на локальный диск, вывод сверток может и не записываться), но теперь он необходим на машине, которая будет выполнять задачу свертки для раздела. Кроме того, задаче свертки необходим вывод отображения для ее конкретного раздела от нескольких задач отображений из кластера. Задачи отображения могут завершаться в разное время, поэтому задача свертки начинает копировать их выходные данные по мере завершения. Это *фаза копирования* задачи свертки. Задача свертки располагает несколькими копирующими программными потоками, чтобы выходные данные отображений могли загружаться параллельно. По умолчанию используются пять программных потоков, но это количество можно изменить, задав свойство `mapred.reduce.parallel.copies`.



Как задачи свертки узнают, от каких машин следует получать данные? Успешно завершающиеся задачи отображения оповещают об изменении своего состояния родительский трекер задач, который оповещает трекер задачий. (В MapReduce 2 задачи оповещают контроллер приложений напрямую.) Для передачи оповещений используется механизм периодических сигналов. Таким образом, для конкретного задания трекер задачий (или контроллер приложений) располагает информацией о соответствии между выходными данными отображений и хостами. Программный поток в задаче свертки периодически запрашивает у контроллера данные хостов, содержащих вывод отображений, пока не получит их все.

Хосты не удаляют выходные данные отображений с диска сразу же после их получения первой сверткой, потому что при выполнении свертки может произойти сбой. Вместо этого они ждут, пока трекер задачий (или контроллер приложений) не разрешит удалить данные; это происходит после завершения задания.

Выходные данные отображений копируются в память JVM задачи свертки, если они достаточно малы (размер буфера определяется свойством `mapred.job.shuffle.input.buffer.percent`, определяющим часть кучи, используемую для этой цели); в противном случае они копируются на диск. Когда буфер в памяти достигает порогового размера (определенного свойством `mapred.job.shuffle.merge.percent`) или достигает порогового числа выводов отображений (`mapred.inmem.merge.threshold`), происходит слияние данных и выгрузка их на диск. Если комбинирующая функция задана, она выполняется во время слияния для сокращения объема данных, записываемых на диск.

По мере накопления копий на диске фоновый программный поток сливает их в большие, отсортированные файлы. Это экономит время по сравнению с выполнением слияния в будущем. Все выходные данные отображений, которые были сжаты (задачей отображения), должны быть распакованы в памяти для выполнения слияния.

Когда все выходные данные отображений будут скопированы, задача свертки переходит в фазу сортировки (которую было бы правильнее называть «фазой слияния», потому что сортировка была выполнена на стороне отображения). В этой фазе происходит слияние выходных данных отображений с сохранением порядка сортировки. Слияние осуществляется в несколько *раундов*. Например, при 50 выводах отображений и с коэффициентом слияния 10 (значение по умолчанию, определяемое свойством `io.sort.factor`, как и для слияния в отображениях) будет 5 раундов. За каждый раунд 10 файлов сливаются в один, и в конечном итоге будет создано 5 промежуточных файлов.

Вместо выполнения последнего раунда, в котором эти 5 файлов будут слиты в один отсортированный файл, процесс слияния избегает лишнего обращения к диску, напрямую передавая данные функции свертки. Передача данных составляет последнюю фазу: *фазу свертки*. Данные для завершающего слияния могут храниться как в памяти, так и в сегментах на диске.



Количество файлов, подвергаемых слиянию в каждом раунде, не настолько тривиально, как в приведенном примере. Целью является слияние минимального количества файлов для достижения заданного коэффициента слияния к последнему раунду. Таким образом, для 40 файлов слияние не будет объединять по 10 файлов в 4 раунда, чтобы получить 4 файла. Вместо этого за первый раунд будут объединены только 4 файла, а в последующие три — по 10 файлов. 4 слитых файла и 6 (еще не слитых) составляют 10 файлов для последнего раунда. Процесс изображен на рис. 6.7.

Количество раундов при этом не изменяется; это оптимизация, минимизирующая объем данных, записываемых на диск, потому что результаты слияния в последнем раунде всегда передаются напрямую свертке.

В фазе свертки функция свертки вызывается для каждого ключа в отсортированных выходных данных. Выходные данные этой фазы записываются непосредственно в выходную файловую систему, чаще всего в HDFS. В случае HDFS, поскольку трекер задач (или менеджер узлов) совмещен с узлом данных, первая реплика блока будет записана на локальный диск.

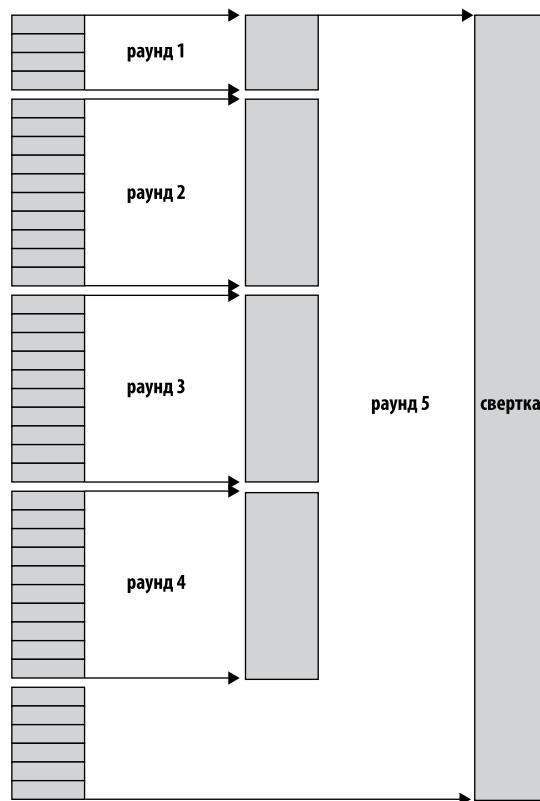


Рис. 6.7. Эффективное слияние 40 частей файла с коэффициентом слияния 10

Настройка конфигурации

Сейчас вы уже знаете достаточно для того, чтобы понять, как настроить тасовку для улучшения производительности MapReduce. Соответствующие настройки, которые могут использоваться на уровне заданий (кроме тех случаев, где это явно указано в тексте), перечислены в табл. 6.1 и 6.2 вместе со значениями по умолчанию, которые достаточно хорошо подходят для заданий общего назначения.

Общий принцип заключается в том, чтобы предоставить тасовке как можно больше памяти. Однако при этом также необходимо позаботиться и о том, чтобы у функций отображения и свертки тоже было достаточно памяти для работы. Вот почему следует писать функции отображения и свертки так, чтобы они использовали как можно меньше памяти — и уж конечно, они не должны использовать неограниченный объем памяти (например, при накоплении значений в ассоциативном массиве).

Объем памяти, выделяемой JVM, в которой работают задачи отображения и свертки, задается свойством `mapred.child.java.opts`. Постарайтесь сделать его как можно больше, чтобы максимизировать объем памяти узлов задач; некоторые ограничения, которые при этом следует учитывать, описаны в разделе «Память», с. 396.

На стороне отображения лучшая производительность достигается предотвращением множественных выгрузок на диск; оптимальное количество — одна выгрузка. Если вы можете оценить размер выходных данных отображений, задайте свойства `io.sort.*` так, чтобы свести к минимуму количество выгрузок. В частности, постарайтесь увеличить `io.sort.mb`, если это возможно. В MapReduce имеется счетчик (см. «Счетчики», с. 337) для подсчета общего количества записей, выгруженных на диск в процессе задания. Он может помочь при настройке. Учтите, что в счетчик включаются выгрузки как на стороне отображения, так и на стороне свертки.

На стороне свертки наилучшая производительность достигается в том случае, если промежуточные данные находятся полностью в памяти. По умолчанию это не делается, потому что вся память резервируется для функции свертки. Но если функция свертки предъявляет малые требования к памяти, задайте `mapred.inmem.merge.threshold` значение 0, а `mapred.job.reduce.input.buffer.percent` значение 1.0 (или меньше — см. табл. 6.2); возможно, это обеспечит прирост производительности.

Hadoop по умолчанию использует размер буфера 4 Кбайт. В общем случае этого мало, и размер буфера для кластера стоит увеличить (при помощи свойства `io.file.buffer.size`; также см. «Другие свойства Hadoop», с. 408).

В апреле 2008 года технология Hadoop победила в категории эталонных тестов террабайтной сортировки общего назначения, и одной из оптимизаций стало именно хранение промежуточных данных в памяти на стороне свертки.

Таблица 6.1. Свойства оптимизации на стороне отображения

Имя	Тип	Значение по умолчанию	Описание
io.sort.mb	int	100	Размер буфера памяти, используемого при сортировке выходных данных отображения (в мегабайтах)
io.sort.record.percent	float	0,05	Часть io.sort.mb, зарезервированная для хранения границ записей выходных данных отображения. Оставшееся пространство используется для хранения самих выходных данных отображения. В версиях 1.x и выше это свойство было исключено, так как код тасовки после доработки стал лучше справляться с использованием всей доступной памяти для хранения вывода отображений и служебной информации
io.sort.spill.percent	float	0,80	Порог использования буфера выходных данных отображения и индекса границ записей, после которого начинается процесс выгрузки на диск
io.sort.factor	int	10	Максимальное количество потоков, сливаемых одновременно при сортировке файлов. Свойство также используется при свертке. На практике очень часто увеличивается до 100
min.num.spills.for.combine	int	3	Минимальное количество файлов выгрузки, необходимое для выполнения комбинирующей функции (если она определена)
mapred.compress.map.output	boolean	false	Сжатие выходных данных отображения
mapred.map.output.compression.codec	имя класса	org.apache.hadoop.io.compress.DefaultCodec	Кодек сжатия, применяемый к выходным данным отображения
tasktracker.http.threads	int	40	Количество рабочих программных потоков трекера задач, предоставляющих выходные данные отображения задачам свертки. Свойство действует на уровне кластера и не может задаваться для отдельных заданий. Не поддерживается в MapReduce 2

Таблица 6.2. Свойства оптимизации на стороне свертки

Имя	Тип	Значение по умолчанию	Описание
mapred.reduce.parallel.copies	int	5	Количество программных потоков, используемых для копирования выходных данных отображения в задаче свертки
mapred.reduce.copy.backoff	int	300	Максимальный промежуток времени (в секундах), затраченный на передачу одного вывода отображения задаче свертки, после которого он помечается как сбойный. Задача свертки может повторить попытку передачи в пределах этого промежутка (с применением экспоненциальной отсрочки)
io.sort.factor	int	10	Максимальное количество потоков, сливающихся одновременно при сортировке файлов. Свойство также используется при отображении
mapred.job.shuffle.input.buffer.percent	float	0,70	Часть общего размера кучи, выделяемая под буфер выходных данных отображения в фазе копирования процесса тасовки
mapred.job.shuffle.merge.percent	float	0,66	Порог использования буфера выходных данных отображения (определенного свойством mapred.job.shuffle.input.buffer.percent), после которого начинается процесс слияния выходных данных и выгрузки их на диск
mapred.in-mem.merge.threshold	int	1000	Пороговое количество выходных данных отображения, после которого начинается процесс слияния выходных данных и выгрузки их на диск. Значение 0 и менее означает, что порог не определен, а поведением выгрузки управляет исключительно mapred.job.shuffle.merge.percent
mapred.job.reduce.input.buffer.percent	float	0,0	Часть общей кучи, используемая для хранения выходных данных отображения в памяти во время свертки. Чтобы началась фаза свертки, размер выходных данных отображения в памяти не должен превышать заданную величину. По умолчанию все выходные данные отображения сливаются на диске перед началом свертки, чтобы задаче свертки было доступно как можно больше памяти. Но если в вашем случае свертка требует меньше памяти, значение может быть увеличено для сокращения количества обращений к диску

Выполнение задач

В начале этой главы, в разделе «Выполнение задания MapReduce» уже был описан процесс выполнения задач системой MapReduce в контексте задания. В этом разделе мы рассмотрим некоторые возможности управления выполнением задач, доступные для пользователей MapReduce.

Среда выполнения задач

Hadoop предоставляет задачам отображения и свертки информацию о среде, в которой они выполняются. Например, задача отображения может узнать имя обрабатываемого файла (см. «Информация о файле в задаче отображения», с. 317), а задача отображения или свертки может узнать номер попытки задачи. Свойства из табл. 6.3 доступны в конфигурации задачи, для получения которой в старом MapReduce API следовало предоставить реализацию метода `configure()` для `Mapper` или `Reducer`, которой конфигурация передавалась в аргументе. В новом API к этим свойствам можно обращаться из объекта контекста, передаваемого всем методам `Mapper` или `Reducer`.

Таблица 6.3. Свойства окружения задачи

Имя	Тип	Описание	Пример
mapred.job.id	String	Идентификатор задания (описание формата приведено во врезке «Идентификаторы задания, задачи и попытки», с. 229)	job_200811201130_0004
mapred.tip.id	String	Идентификатор задачи	task_200811201130_-0004_m_000003
mapred.task.id	String	Идентификатор попытки выполнения задачи (не идентификатор задачи!)	attempt_200811201130_0004_m_000003_0
mapred.task.partition	int	Индекс задачи в задании	3
mapred.task.is.map	boolean	Признак задачи отображения	true

Переменные окружения Streaming

Для программ Streaming Hadoop устанавливает параметры конфигурации задания в виде переменных окружения. При этом символы, не являющиеся алфавитно-цифровыми, заменяются подчеркиваниями, чтобы имена оставались действительными. Следующее выражение Python демонстрирует обращение к значению свойства `mapred.job.id` в сценарии Streaming, написанном на Python:

```
os.environ["mapred_job_id"].
```

Чтобы задать переменные окружения для процессов Streaming, запущенных MapReduce, передайте параметр `-cmdenv` программе запуска Streaming (по одному для каждой задаваемой переменной). Например, следующая команда задает переменную окружения `MAGIC_PARAMETER`:

```
-cmdenv MAGIC_PARAMETER=abraacadabra.
```

Спекулятивное выполнение

Модель MapReduce проектировалась для разбиения заданий на задачи и параллельного выполнения задач с целью сокращения времени выполнения (по сравнению с последовательным выполнением задач).

Из-за этого время выполнения задания начинает зависеть от медленных задач, потому что всего одна долго выполняемая задача способна существенно замедлить выполнение всего задания. Если задание состоит из сотен и тысяч задач, вероятность появления медленных задач становится более чем реальной.

Задачи могут медленно выполняться по разным причинам, включая аппаратную деградацию или неправильную конфигурацию программного обеспечения. Диагностика усложняется тем, что задачи по-прежнему успешно завершаются, хотя их выполнение занимает больше времени, чем ожидалось. Hadoop не пытается диагностировать и решать проблемы медленного выполнения задач; вместо этого Hadoop старается определить, когда задача выполняется медленнее ожидаемого, и запускает другую эквивалентную задачу в качестве резервной. Этот метод называется *спекулятивным выполнением* задач.

Важно понять, что спекулятивное выполнение не сводится к запуску двух дубликатов более или менее одновременно, чтобы они могли конкурировать друг с другом. Такой подход был бы расточительной тратой ресурсов кластера. Спекулятивная задача запускается только после запуска всех задач, входящих в задание, и только для задач, которые проработали некоторое время (не меньше минуты), но в отношении прогресса в среднем отстали от других задач. Когда задача завершается успешно, все выполняемые дубликаты уничтожаются, потому что они становятся

ненужными. Итак, если исходная задача завершается раньше спекулятивной, то уничтожается спекулятивная задача; если же первой завершится спекулятивная задача, то уничтожается исходная задача.

Спекулятивное выполнение — оптимизация, а не механизм повышения надежности выполнения заданий. Если в коде присутствуют ошибки, иногда приводящие к «зависанию» или замедлению выполнения задачи, не стоит рассчитывать на то, что спекулятивное выполнение поможет обойти эти проблемы; скорее всего, те же ошибки проявятся и в спекулятивной задаче. Чтобы задача не «зависала» и не замедлялась, исправьте ошибку.

Спекулятивное выполнение включено по умолчанию. Его можно включать или отключать независимо для задач отображения или задач свертки, на уровне кластера или на уровне задания. Соответствующие свойства перечислены в табл. 6.4.

Таблица 6.4. Свойства спекулятивного выполнения

Имя	Тип	Значение по умолчанию	Описание
mapred.map.tasks.speculative.execution	boolean	true	Флаг, разрешающий запуск дополнительных экземпляров задачи отображения, которая прогрессирует слишком медленно
mapred.reduce.tasks.speculative.execution	boolean	true	Флаг, разрешающий запуск дополнительных экземпляров задачи свертки, которая прогрессирует слишком медленно
yarn.app.mapreduce.am.job.speculator.class	класс	org.apache.hadoop.mapreduce.v2.app.speculate.DefaultSpeculator	Класс Speculator, реализующий политику спекулятивного выполнения (только для MapReduce 2)
yarn.app.mapreduce.am.job.task.estimator.class	класс	org.apache.hadoop.mapreduce.v2.app.speculate.LegacyTaskRuntimeEstimator	Реализация TaskRuntimeEstimator, используемая экземплярами Speculator, представляющими оценки времени выполнения задач (только для MapReduce 2)

Когда отключается спекулятивное выполнение? Его целью является сокращение времени выполнения заданий, но за это приходится расплачиваться эффективностью кластера. В занятом кластере спекулятивное выполнение сокращает общую пропускную способность, поскольку избыточные задачи выполняются для

ускорения выполнения одного задания. По этой причине некоторые администраторы кластеров предпочитают отключать спекулятивное выполнение в кластерах, чтобы пользователи явно включали его для конкретных заданий. Это особенно актуально для старых версий Hadoop, в которых механизм спекулятивного выполнения мог слишком агрессивно действовать при планировании спекулятивных задач.

Существуют веские доводы в пользу отключения спекулятивного выполнения для задач свертки, поскольку каждый дубликат задачи свертки должен получать те же выходные данные отображений, что и исходная задача, а это приводит к существенному повышению сетевого трафика в кластере.

Также отключение спекулятивного выполнения имеет смысл для задач, не обладающих свойством идемпотентности. Однако во многих случаях можно написать задачу, которая является идемпотентной и использует `OutputCommitter` для перемещения своего вывода в нужное место при успешном завершении задачи. Этот метод более подробно рассматривается в следующем разделе.

OutputCommitter

Hadoop MapReduce использует протокол закрепления (`commit`), гарантирующий, что задания и задачи либо завершаются успешно, либо корректно отрабатывают сбой. Поведение реализуется реализацией `OutputCommitter`, используемой для задания. В старом MapReduce API она назначается вызовом `setOutputCommitter()` для объекта `JobConf` или заданием свойства `mapred.output.committer.class` в конфигурации. В новом MapReduce API реализация `OutputCommitter` определяется методом `getOutputCommitter()` объекта `OutputFormat`. По умолчанию используется реализация `FileOutputCommitter`, подходящая для заданий MapReduce, работающих с файлами. Вы можете настроить существующую реализацию `OutputCommitter` и даже написать новую реализацию, если вам понадобится выполнять специальную подготовку или deinициализацию для заданий и задач.

`OutputCommitter` API выглядит следующим образом (и в старом, и в новом MapReduce API):

```
public abstract class OutputCommitter {  
  
    public abstract void setupJob(JobContext jobContext) throws IOException;  
    public void commitJob(JobContext jobContext) throws IOException { }  
    public void abortJob(JobContext jobContext, JobStatus.State state)  
        throws IOException { }  
  
    public abstract void setupTask(TaskAttemptContext taskContext)  
        throws IOException;
```

```
public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
    throws IOException;
public abstract void commitTask(TaskAttemptContext taskContext)
    throws IOException;
public abstract void abortTask(TaskAttemptContext taskContext)
    throws IOException;
}
```

Метод `setupJob()` вызывается перед запуском задания; обычно он используется для выполнения инициализации. Для `FileOutputCommitter` метод создает каталог выходных данных `${mapred.output.dir}` и временное рабочее пространство для вывода задач `${mapred.output.dir}/_temporary` .

Если задание завершается успешно, вызывается метод `commitJob()`, который в файловой реализации по умолчанию удаляет временное рабочее пространство и создает в каталоге выходных данных пустой скрытый файл-маркер с именем `_SUCCESS`, присутствие которого сообщает клиентам файловой системы об успешном завершении задания. Если задание не завершилось успешно, вызывается метод `abortJob()` с объектом состояния, указывающим, произошел ли в задании сбой или оно было уничтожено (пользователем, например). В реализации по умолчанию при этом удаляется временное рабочее пространство задания.

Операции на уровне задач выглядят аналогично. Метод `setupTask()` вызывается перед выполнением задачи; реализация по умолчанию не делает ничего.

Фаза закрепления для задач не является обязательной; ее можно отключить, вернув `false` из `needsTaskCommit()`. В этом случае инфраструктура не выполняет для задачи протокол распределенного закрепления, и ни один из методов `commitTask()` и `abortTask()` не вызывается. `FileOutputCommitter` пропускает фазу закрепления, если задача не записала никакие выходные данные.

В случае успешного выполнения вызывается метод `commitTask()`, который в реализации по умолчанию перемещает временный каталог выходных данных задачи (в имени которого содержится идентификатор попытки задачи, чтобы предотвратить возможные конфликты между попытками) в итоговый выходной каталог `${mapred.output.dir}` . В противном случае инфраструктура вызывает метод `abortTask()`, который удаляет временный каталог выходных данных задачи.

Инфраструктура гарантирует, что в случае нескольких попыток выполнения конкретной задачи закреплена будет только одна попытка; остальные попытки отменяются. Такая ситуация может возникнуть из-за того, что первая попытка по какой-то причине оказалась сбоящей — тогда она отменяется, а закрепляется другая, успешная попытка. Также возможна ситуация, при которой две попытки выполняются параллельно как спекулятивные дубликаты; в этом случае попытка, завершившаяся первой, будет закреплена, а другая попытка отменится.

Файлы побочных эффектов

Обычным способом записи выходных данных задачами отображения и свертки является накопление пар «ключ-значение» с использованием `OutputCollector`. Некоторым приложениям требуется большая гибкость, чем модель пар «ключ-значение», поэтому такие приложения записывают выходные файлы прямо из задач отображения или свертки в распределенную файловую систему – например, в HDFS (другие способы описаны в разделе «Множественный вывод», с. 331).

Необходимо проследить за тем, чтобы множественные экземпляры одной задачи не пытались выполнять запись в один файл. Как было показано в предыдущем разделе, протокол `OutputCommitter` решает эту проблему. Если приложения записывают файлы побочных эффектов (*side-effect files*) в рабочие каталоги своих задач, то побочные файлы успешно завершенных задач будут автоматически переведены в выходной каталог, а побочные файлы сбойных задач удаляются.

Задача может определить свой рабочий каталог, прочитав значение свойства `mapred.work.output.dir` из своего файла конфигурации. Кроме того, программа MapReduce, использующая Java API, может вызвать статический метод `getWorkOutputPath()` класса `FileOutputFormat` для получения объекта `Path`, представляющего рабочий каталог. Инфраструктура создает рабочий каталог перед выполнением задачи, так что вам его создавать не обязательно.

Рассмотрим простой пример: программу для преобразования графических файлов из одного формата в другой. В одном из способов создается задание, состоящее только из отображений, при этом каждое отображение получает набор преобразуемых файлов (вероятно, с использованием `NLineInputFormat`; см. «`NLineInputFormat`», с. 324). Если задача отображения записывает преобразованные файлы в свой рабочий каталог, они будут перенесены в выходной каталог при успешном завершении задачи.

Повторное использование JVM задач

Hadoop выполняет задачи в своей виртуальной машине Java (JVM), чтобы отделить их от других выполняемых задач. Запуск новой JVM для каждой задачи может занимать около секунды; для заданий, выполняемых по минуте и более, это несущественно. Однако для заданий с большим количеством непродолжительных задач (обычно это задачи отображения) или продолжительной инициализацией повторное использование JVM для последующих задач может повысить производительность¹.

¹ Повторное использование JVM в настоящее время не поддерживается в MapReduce 2.

При включении повторного использования JVM задачи не выполняются одновременно в одной виртуальной машине; вместо этого JVM выполняет задачи последовательно. Трекеры задач могут выполнять более одной задачи, но это всегда делается в разных JVM. Свойства, управляющие количеством слотов задач отображения и слотов задач свертки на трекер задач, обсуждаются в разделе «Память», с. 396.

Повторным использованием JVM задач управляет свойство `mapred.job.reuse.jvm.num.tasks`. Оно задает максимальное количество задач, выполняемых для указанного задания в каждой запущенной виртуальной машине Java; значение по умолчанию равно 1 (табл. 6.5). Задачи отображения и свертки при этом не различаются; тем не менее задачи разных заданий всегда выполняются в разных JVM. Для настройки свойства также может использоваться метод `setNumTasksToExecutePerJvm()` класса `JobConf`.

Таблица 6.5. Свойства повторного использования JVM задач

Имя	Тип	Значение по умолчанию	Описание
<code>mapred.job.reuse.jvm.num.tasks</code>	int	1	Максимальное количество задач, выполняемых для указанного задания для каждой JVM трекера задач. Значение -1 означает отсутствие ограничений, то есть одна JVM может использоваться для всех задач, входящих в задание

Счетные задачи также выигрывают от повторного использования JVM за счет использования оптимизаций времени выполнения, применяемых HotSpot JVM. Проработав в течение некоторого времени, HotSpot JVM собирает достаточно информации для обнаружения в коде участков, критичных для производительности, и динамически транслирует байт-коды Java таких участков в машинный код. Оптимизация хорошо работает для продолжительных процессов, но JVM, работающие по несколько секунд или минут, не смогут в полной мере использовать преимущества HotSpot. В таких ситуациях стоит подумать о включении повторного использования JVM задач.

Пропуск некорректных записей

Большие наборы данных редко бывают идеальными. В них часто встречаются поврежденные записи. В них встречаются записи, хранимые в другом формате. В них

часто встречаются пропущенные поля. В идеальном мире ваш код будет корректно обрабатывать все перечисленные ситуации. На практике часто бывает разумнее проигнорировать записи-нарушители. В зависимости от выполняемого анализа, если проблемы встречаются только в малом проценте записей, пропуск этих записей вряд ли значительно повлияет на результат. Но если задача, столкнувшись с некорректной записью, выдает исключение времени выполнения — происходит сбой. Трекер заданий пытается снова выполнить сбойные задачи (так как сбой может быть обусловлен аппаратным отказом или другой причиной, неподконтрольной для задачи), но в случае четырехкратного сбоя все задание помечается как сбойное (см. «Сбой задачи», с. 272). Если исключение происходит из-за данных, повторный запуск задачи не поможет, потому что каждый раз сбой будет происходить по тому же сценарию.



При использовании TextInputFormat (см. «TextInputFormat», с. 322) можно задать максимальную ожидаемую длину строки, чтобы в определенной степени защититься от некорректных данных. Повреждение файла может проявиться в появлении очень длинных строк, приводящих к ошибкам нехватки памяти и последующим сбоям задачи. Задавая mapred.linerecordreader.maxlength значение в байтах, защищающее от нарушения границ памяти (и превышающее длину строк в ваших входных данных), объект чтения будет пропускать (длинные) поврежденные строки без сбоя задачи.

Лучше всего обрабатывать некорректные записи в коде отображения и свертки. Вы можете обнаружить некорректную запись и проигнорировать ее или же аварийно завершить задание с выдачей исключения. Также можно подсчитать общее количество некорректных записей в задании при помощи счетчиков, чтобы понять, насколько распространена проблема.

В отдельных случаях проблема оказывается неразрешимой из-за ошибки в сторонней библиотеке, которую не удается обойти в коде отображения или свертки. В таких случаях можно воспользоваться режимом автоматического пропуска некорректных записей Hadoop¹.

В режиме пропуска (*skipping mode*) задачи передают информацию об обрабатываемых записях трекеру задач. При сбое задачи трекер задач пытается снова выполнить ее, пропуская записи, вызвавшие сбой. Из-за дополнительного сетевого трафика и необходимости хранения служебной информации о диапазонах

¹ В новом MapReduce API этот режим не поддерживается.

некорректных записей режим пропуска включается для задачи только после двухкратного сбоя.

Итак, для задачи, у которой стабильно происходит сбой на некорректной записи, трекер задач выполняет следующие попытки со следующими результатами:

1. Сбой задачи.
2. Сбой задачи.
3. Включение режима пропуска. В задаче снова происходит сбой, но сбойная запись сохраняется трекером задач.
4. Режим пропуска остается включенным. Задача пропускает некорректную запись, приведшую к сбою на предыдущей попытке, и успешно работает.

Режим пропуска отключен по умолчанию; он включается независимо для задач отображения и свертки при помощи класса `SkipBadRecords`. Следует помнить, что режим пропуска способен обнаруживать только одну некорректную запись на попытку выполнения задачи, поэтому этот механизм подходит только для обнаружения относительно редких некорректных записей (в количестве, скажем, нескольких на задачу). Возможно, стоит увеличить максимальное количество попыток (свойства `mapred.map.max.attempts` и `mapred.reduce.max.attempts`), чтобы предоставить режиму пропуска достаточно попыток для обнаружения и пропуска всех некорректных записей во входном сплите.

Некорректные записи, обнаруженные Hadoop, сохраняются в виде последовательных файлов в подкаталоге `_logs/skip` выходного каталога задания. Файлы можно просмотреть для диагностических целей после завершения задания (например, с использованием команды `hadoop fs -text`).

7

Типы и форматы MapReduce

MapReduce реализует простую модель обработки данных: входные и выходные данные функций отображения и свертки представляют собой пары «ключ-значение». В этой главе подробно рассматривается модель MapReduce и, в частности, возможности использования в ней данных разных форматов, от простого текста до структурированных двоичных объектов.

Типы MapReduce

Функции отображения и свертки в Hadoop MapReduce имеют общую форму:

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

В общем случае типы входного ключа и значения отображения (K1 и V1) отличны от выходных типов отображения (K2 и V2). Входные данные свертки должны совпадать с выходными типами отображения, хотя выходные типы свертки снова могут различаться (K3 и V3). Вот как эта общая форма представлена в Java API:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

    public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
        // ...
    }

    protected void map(KEYIN key, VALUEIN value,
```

```
        Context context) throws IOException, InterruptedException {  
    // ...  
}  
}  
  
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
  
    public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>  
    {  
        // ...  
    }  
  
    protected void reduce(KEYIN key, Iterable<VALUEIN> values,  
                         Context context) throws IOException,  
                         InterruptedException {  
        // ...  
    }  
}
```

Объекты контекста используются для генерирования пар «ключ-значение», поэтому они параметризуются по выходным типам, чтобы сигнатура метода `write()` выглядела следующим образом:

```
public void write(KEYOUT key, VALUEOUT value)  
    throws IOException, InterruptedException
```

Так как `Mapper` и `Reducer` являются отдельными классами, параметры типов имеют разную область видимости, и фактический аргумент типа `KEYIN` (допустим) в `Mapper` может отличаться от типа одноименного параметра (`KEYIN`) в `Reducer`. Так, в примере с определением максимальной температуры из предыдущих глав `KEYIN` заменяется типом `LongWritable` для `Mapper` и типом `Text` для `Reducer`.

Кроме того, хотя выходные типы отображения должны соответствовать входным типам свертки, компилятор Java это требование не проверяет.

Имена параметров типов отличаются от имен абстрактных типов (`KEYIN/K1` и т. д.), но форма остается неизменной.

Если в задании используется комбинирующая функция, она пишется в той же форме, что и функция свертки (и является реализацией `Reducer`), за исключением того, что его выходными типами являются типы промежуточного ключа и значения (`K2` и `V2`), чтобы они могли передаваться на вход функции свертки:

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

Часто комбинирующая функция и функция свертки совпадают; в этом случае K3 совпадает с K2, а V3 совпадает с V2.

Функция разделения работает с типами промежуточного ключа и значения (K2 и V2) и возвращает индекс раздела. На практике раздел определяется исключительно по ключу (значение игнорируется):

```
partition: (K2, V2) → integer.
```

Или на языке Java:

```
public abstract class Partitioner<KEY, VALUE> {  
    public abstract int getPartition(KEY key, VALUE value, int numPartitions);  
}
```

СИГНАТУРЫ MAPREDUCE В СТАРОМ API

В старом API сигнатуры были очень похожими. В них даже использовались имена параметров типов K1, V1 и т. д., хотя ограничения типов в старом и новом API полностью совпадают.

```
public interface Mapper<K1, V1, K2, V2> extends JobConfigurable, Closeable {  
  
    void map(K1 key, V1 value, OutputCollector<K2, V2> output,  
             Reporter reporter)  
        throws IOException;  
    }  
    public interface Reducer<K2, V2, K3, V3> extends JobConfigurable, Closeable {  
  
        void reduce(K2 key, Iterator<V2> values,  
                    OutputCollector<K3, V3> output, Reporter reporter) throws IOException;  
    }  
    public interface Partitioner<K2, V2> extends JobConfigurable {  
        int getPartition(K2 key, V2 value, int numPartitions);  
    }
```

Впрочем, довольно теории. Как вся эта информация помогает в настройке заданий MapReduce? В табл. 7.1 приведена сводка параметров конфигурации для нового API (а в табл. 7.2 — для старого API). Сводка разделена на свойства, определяющие типы, и те, которые должны быть совместимы с настроенными типами.

Входные типы задаются входным форматом. Так, например, для `TextInputFormat` генерируются ключи типа `LongWritable` и значения типа `Text`. Другие типы задаются явно вызовами методов `Job` (или `JobConf` в старом API). Если промежуточные типы не заданы явно, они по умолчанию совпадают с (итоговыми) выходными типами, которыми также по умолчанию являются `LongWritable` и `Text`. Так, если K2

и K3 совпадают, вам не нужно вызывать `setMapOutputKeyClass()`, потому что в качестве выходного будет выбран тип, назначенный вызовом `setOutputKeyClass()`. А если V2 и V3 совпадают, достаточно использовать `setOutputValueClass()`.

Само существование методов для задания промежуточных и итоговых выходных типов может показаться странным. В конце концов, почему нельзя определить типы по сочетанию отображения и свертки? Ответ связан с ограничениями механизма параметризации Java: стирание типов (type erasure) означает, что информация типа не всегда доступна во время выполнения, поэтому Hadoop приходится передавать ее явно. Это также означает, что задание MapReduce может настраиваться с несовместимыми типами, потому что конфигурация не проверяется во время компиляции. Настройки, совместимые с типами MapReduce, перечислены в нижней части табл. 7.1. Конфликты типов обнаруживаются во время выполнения задания, поэтому стоит прогнать тестовое задание на небольшом объеме данных, чтобы обнаружить и исправить все несовместимости типов.

Задание MapReduce по умолчанию

Что произойдет, если запустить MapReduce без назначения задачи отображения или свертки? Давайте попробуем выполнить следующую минимальную программу MapReduce:

```
public class MinimalMapReduce extends Configured implements Tool {  
  
    @Override  
    public int run(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.err.printf("Usage: %s [generic options] <input> <output>\n",  
                getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.err);  
            return -1;  
        }  
  
        Job job = new Job(getConf());  
        job.setJarByClass(getClass());  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        return job.waitForCompletion(true) ? 0 : 1;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new MinimalMapReduce(), args);  
        System.exit(exitCode);  
    }  
}
```

Таблица 7.1. Конфигурация типов MapReduce в новом API

Свойство	Set-метод Job	Входные типы			Промежуточные типы		Выходные типы	
		K1	V1	K2	V2	K3	V3	
Свойства настройки типов								
mapreduce.job.inputformat.class	setInputFormatClass()	*	*					
mapreduce.map.output.key.class	setMapOutputKeyClass()			*				
mapreduce.map.output.value.class	setMapOutputValueClass()			*				
mapreduce.job.output.key.class	setOutputKeyClass()			*				
mapreduce.job.output.value.class	setOutputValueClass()			*				
Свойства, соответствующие типам								
mapreduce.job.map.class	setMapperClass()	*	*	*	*	*	*	
mapreduce.job.combine.class	setCombinerClass()			*		*		
mapreduce.job.partitioner.class	setPartitionerClass()			*		*		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			*				
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			*				
mapreduce.job.reduce.class	setReducerClass()			*		*	*	*
mapreduce.job.outputformat.class	setOutputFormatClass()			*		*	*	

Таблица 7.2. Конфигурация типов MapReduce в старом API

Свойство	Set-метод JobConf	Входные типы	Промежуточные типы	Выходные типы
		K1	V1	K2
Свойства настройки типов				
mapred.input.format.class	setInputFormat()	*	*	
mapred.mapoutput.key.class	setMapOutputKeyClass()		*	
mapred.mapoutput.value.class	setMapOutputValueClass()		*	
mapred.output.key.class	setOutputKeyClass()		*	
mapred.output.value.class	setOutputValueClass()		*	
Свойства, соответствующие типам				
mapred.mapper.class	setMapperClass()	*	*	*
mapred.map.runner.class	setMapRunnerClass()	*	*	*
mapred.combiner.class	setCombinerClass()	*	*	*
mapred.partitioner.class	setPartitionerClass()	*	*	*
mapred.output.key.comparator.class	setOutputKeyComparatorClass()		*	
mapred.output.value.groupfn.class	setOutputValueGroupingComparator()		*	
mapred.reducer.class	setReducerClass()	*	*	*
mapred.output.format.class	setOutputFormat()		*	*

Из всех параметров конфигурации задаются только входной и выходной каталоги. Запустим MapReduce для подмножества метеорологических данных следующей командой:

```
% hadoop MinimalMapReduce "input/ncdc/all/190{1,2}.gz" output
```

В выходном каталоге появляется файл с именем *part-r-00000*. Вот как выглядят его начальные строки (вывод усечен по ширине страницы):

```
0→00290290709999919010106004+64333+023450FM-12+000599999V0202701N01591...
0→0035029070999991902010106004+64333+023450FM-12+000599999V0201401N01181...
135→0029029070999991901010113004+64333+023450FM-12+000599999V0202901N008...
141→0035029070999991902010113004+64333+023450FM-12+000599999V0201401N011...
270→0029029070999991901010120004+64333+023450FM-12+000599999V0209991C000...
282→0035029070999991902010120004+64333+023450FM-12+000599999V0201401N013...
```

Каждая строка состоит из целого числа, за которым следует символ табуляции и исходная запись метеорологических данных. Программа, конечно, бесполезная, но понимание того, как она генерирует свои данные, дает представление о настройках по умолчанию, которые используются Hadoop при запуске заданий MapReduce. В листинге 7.1 приведена программа, которая выдает точно такой же результат, что и `MinimalMapReduce`, но явно задает параметрам значения по умолчанию.

Листинг 7.1. Минимальная программа MapReduce с явно заданными значениями по умолчанию

```
public class MinimalMapReduceWithDefaults extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
        if (job == null) {
            return -1;
        }

        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(Mapper.class);

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

        job.setPartitionerClass(HashPartitioner.class);

        job.setNumReduceTasks(1);
        job.setReducerClass(Reducer.class);
```

```
job.setOutputKeyClass(LongWritable.class);
job.setOutputValueClass(Text.class);
job.setOutputFormatClass(TextOutputFormat.class);

return job.waitForCompletion(true) ? 0 : 1;
}

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

Мы упростили несколько первых строк метода `run()`, выделив логику вывода справки и назначения входных/выходных путей во вспомогательный метод. Почти все управляющие программы MapReduce получают эти два аргумента (входной и выходной путь), так что выделение шаблонного кода не повредит. Ниже приведены методы класса `JobBuilder`:

```
public static Job parseInputAndOutput(Tool tool, Configuration conf,
    String[] args) throws IOException {

    if (args.length != 2) {
        printUsage(tool, "<input> <output>");
        return null;
    }
    Job job = new Job(conf);
    job.setJarByClass(tool.getClass());
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    return job;
}

public static void printUsage(Tool tool, String extraArgsUsage) {
    System.err.printf("Usage: %s [genericOptions] %s\n\n",
        tool.getClass().getSimpleName(), extraArgsUsage);
    GenericOptionsParser.printGenericCommandUsage(System.err);
}
```

Вернемся к примеру `MinimalMapReduceWithDefaults` из листинга 7.1. Хотя в конфигурации по умолчанию устанавливаются много других параметров, для запуска заданий наиболее важными являются строки, выделенные жирным шрифтом.

Для представления в качестве формата входных данных по умолчанию выбирается объект `TextInputFormat`, производящий ключи типа `LongWritable` (смещение

в начале строки файла) и значения типа `Text` (строка текста). Это объясняет, откуда взялись целые числа в итоговых выходных данных: это смещения строк.

В качестве отображения по умолчанию используется класс `Mapper`, который записывает входной ключ и значение в выходной поток без изменений:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
  
    protected void map(KEYIN key, VALUEIN value,  
                      Context context) throws IOException, InterruptedException {  
        context.write((KEYOUT) key, (VALUEOUT) value);  
    }  
}
```

`Mapper` — параметризованный тип, что позволяет ему работать с любыми типами ключей и значений. В нашем примере входной и выходной ключи отображения имеют тип `LongWritable`, а входное и выходное значения отображения — тип `Text`.

По умолчанию используется разделитель (`partitioner`) `HashPartitioner`, хеширующий ключ записи для определения раздела, к которому принадлежит запись. Каждый раздел обрабатывается задачей свертки, поэтому количество разделов соответствует количеству задач свертки в задании:

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {  
  
    public int getPartition(K key, V value,  
                           int numReduceTasks) {  
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;  
    }  
}
```

Хеш-код ключа преобразуется в неотрицательное целое посредством объединения его поразрядной операцией AND с наибольшим целым числом. Затем результат делится нацело на количество разделов, а остаток от целочисленного деления определяет индекс раздела, к которому относится запись.

По умолчанию используется одна задача свертки, а следовательно — один раздел. В этом случае действие несущественно, так как все данные попадают в один раздел. Тем не менее вы должны понимать поведение `HashPartitioner` в ситуациях с несколькими задачами свертки. С качественной хеш-функцией ключа записи будут равномерно распределяться между задачами; при этом все записи с одним ключом будут обрабатываться одной задачей свертки.

Возможно, вы заметили, что мы не задали количество задач отображения. Дело в том, что оно равно количеству сплитов, в которые преобразуются входные

данные, а количество сплитов определяется размером входных данных и размером блока (если файл хранится в HDFS). Параметры управления размером сплита рассматриваются в разделе «Входные сплиты FileInputFormat», с. 313.

ВЫБОР КОЛИЧЕСТВА ЗАДАЧ СВЕРТКИ

По умолчанию в Hadoop используется одна задача свертки, и это обстоятельство часто сбивает с толку новых пользователей. Почти во всех реальных задачах количество сверток следует увеличить; в противном случае задание будет работать очень медленно, так как все промежуточные данные будут проходить через одну задачу свертки. (Учтите, что при запуске под управлением локального исполнителя заданий поддерживается только нуль или одна задача свертки.)

Оптимальное количество сверток связано с общим количеством разрешенных слотов свертки в вашем кластере. Общее количество слотов вычисляется умножением количества узлов в кластере на количество слотов в одном узле (которое определяется значением свойства mapred.tasktracker.reduce.tasks.maximum – см. «Настройки окружения», с. 396).

На практике часто используется количество сверток, немногим меньшее общего количества слотов; в этом случае выполнение задач свертки образует одну «волну» (а несколько сбоев не приводят к увеличению общего времени выполнения задания). Если задачи свертки очень велики, можно увеличить количество сверток (например, для выполнения в две «волны»), чтобы задачи стали более детализированными, а сбои не оказывали значительного влияния на время выполнения задания.

По умолчанию для представления свертки используется **Reducer** — также параметризованный тип, который просто записывает все входные данные в свой вывод:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
  
    protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context  
        Context context) throws IOException,  
        InterruptedException {  
        for (VALUEIN value: values) {  
            context.write((KEYOUT) key, (VALUEOUT) value);  
        }  
    }  
}
```

Для этого задания выходной ключ относится к типу `LongWritable`, а выходное значение — к типу `Text`. Собственно, все ключи этой программы MapReduce относятся к типу `LongWritable`, а значения — к типу `Text`, поскольку функции отображения и свертки являются тождественными, а значит, по определению сохраняют тип. Однако в большинстве программ MapReduce не используются постоянные типы ключей и значений, поэтому вы должны настроить задание для объявления используемых типов. О том, как это делается, рассказано в предыдущем разделе.

Прежде чем передаваться на свертку, записи сортируются системой MapReduce. В нашем случае ключи сортируются в числовом виде, что приводит к чередованию строк разных входных файлов в одном объединенном выходном файле.

По умолчанию используется формат вывода `TextOutputFormat`, при котором записи выводятся по одной на строку, с преобразованием ключей и значений в строки и разделением их символом табуляции. Вот откуда в наших выходных данных появились табуляции: это функциональность `TextOutputFormat`.

Задание Streaming по умолчанию

Задание Streaming по умолчанию похоже на свой Java-эквивалент, но не идентично ему. Минимальная форма выглядит так:

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*Streaming.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper /bin/cat
```

Учтите, что вы должны предоставить функцию отображения; используемое по умолчанию тождественное отображение не подходит. Это объясняется тем, что входной формат по умолчанию `TextInputFormat` генерирует ключи `LongWritable` и значения `Text`. Однако выходные ключи и значения Streaming (в том числе ключи и значения отображений) всегда относятся к типу `Text`¹. Тождественное отображение не может превратить ключи `LongWritable` в ключи `Text`, поэтому происходит сбой.

Когда мы указываем отображение, реализованное не на Java, со входным форматом `TextInputFormat`, Streaming действует особым образом. Процессу отображения передается только значение без ключа (для других форматов входных данных аналогичного эффекта можно добиться, задав `stream.map.input.ignoreKey` значение `true`). Это весьма полезная возможность, потому что ключ всего лишь содержит смещение строки в файле, а значение — саму строку, которая представляет интерес

¹ Кроме использования в двоичном режиме (недоступном в 1.x) с параметрами `-io rawbytes` или `-io typedbytes`. По умолчанию используется текстовый режим (`-io text`).

для большинства приложений. Общим эффектом этого задания будет сортировка входных данных.

С явным заданием многих значений по умолчанию команда принимает следующий вид (учтите, что Streaming использует классы старого MapReduce API):

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*Streaming.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -inputformat org.apache.hadoop.mapred.TextInputFormat \
  -mapper /bin/cat \
  -partitioner org.apache.hadoop.mapred.lib.HashPartitioner \
  -numReduceTasks 1 \
  -reducer org.apache.hadoop.mapred.lib.IdentityReducer \
  -outputformat org.apache.hadoop.mapred.TextOutputFormat
```

В аргументах *mapper* и *reducer* передается команда или класс Java. Также возможно (но не обязательно) задать комбинирующую функцию при помощи аргумента *-combiner*.

Ключи и значения в Streaming

Приложение Streaming может управлять разделителем, используемым при преобразовании пары «ключ-значение» в серию байтов и передаче ее процессу отображения или свертки через стандартный ввод. По умолчанию используется символ табуляции, но желательно иметь возможность сменить его, если ключи или значения сами содержат символы табуляции.

Когда отображение или свертка записывает пары «ключ-значение», они также могут разделяться настраиваемым разделителем. Более того, ключ вывода может состоять не только из первого, но и из первых *n* полей (определяемых свойством *stream.num.map.output.key.fields* или *stream.num.reduce.output.key.fields*), при этом значение состоит из остальных полей. Например, если выходные данные процесса Streaming состоят из символов *a,b,c* (с запятой-разделителем), то при *n=2* ключом будет подстрока *a,b*, а значением — *c*.

Разделители могут назначаться независимо для отображений и сверток. Соответствующие свойства перечислены в табл. 7.3 и представлены на диаграмме передачи данных на рис. 7.1.

Эти настройки не влияют на форматы входных и выходных данных. Например, если свойству *stream.reduce.output.field.separator* задано значение *:*, а потоковый процесс свертки запишет в стандартный вывод строку *a:b*, Streaming-свертка извлечет из строки *a* как ключ, а *b* как значение. Со стандартным форматом *TextOutputFormat* эта запись будет записана в выходной файл с разделением *a* и *b*.

табуляцией. Разделитель, используемый `TextOutputFormat`, задается при помощи свойства `mapred.textoutputformat.separator`.

Таблица 7.3. Свойства разделителей Streaming

Имя свойства	Тип	Значение по умолчанию	Описание
<code>stream.map.input.field.separator</code>	<code>String</code>	<code>\t</code>	Разделитель, используемый при передаче входных строк ключа и значения в виде потока байтов потоковому процессу отображения
<code>stream.map.output.field.separator</code>	<code>String</code>	<code>\t</code>	Разделитель, используемый при разбиении вывода потокового процесса отображения на строки ключа и значения для вывода отображения
<code>stream.num.map.output.key.fields</code>	<code>int</code>	<code>1</code>	Количество полей, разделяемых <code>stream.map.output.field.separator</code> , образующих выходной ключ отображения
<code>stream.reduce.input.field.separator</code>	<code>String</code>	<code>\t</code>	Разделитель, используемый при передаче входных строк ключа и значения в виде потока байтов потоковому процессу свертки
<code>stream.reduce.output.field.separator</code>	<code>String</code>	<code>\t</code>	Разделитель, используемый при разбиении вывода потокового процесса свертки на строки ключа и значения для окончательного вывода свертки
<code>stream.num.reduce.output.key.fields</code>	<code>int</code>	<code>1</code>	Количество полей, разделяемых <code>stream.reduce.output.field.separator</code> , образующих выходной ключ свертки

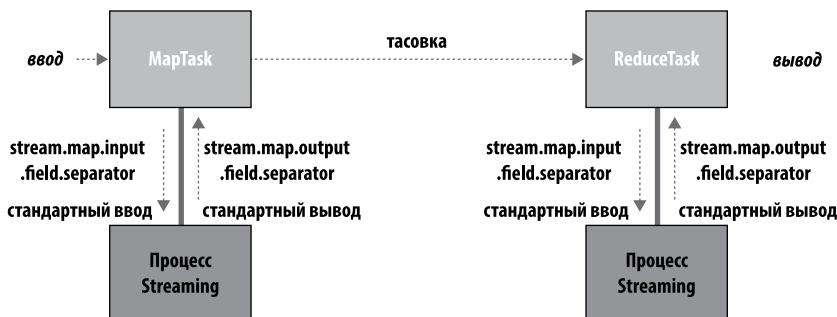


Рис. 7.1. Использование разделителей в Streaming-заданиях MapReduce

Список параметров конфигурации Streaming находится на веб-сайте Hadoop по адресу <http://hadoop.apache.org/mapreduce/docs/current/streaming.html#Configurable+parameters>.

Форматы входных данных

Hadoop может обрабатывать разные типы форматов данных, от неструктурированных текстовых файлов до баз данных. В этом разделе рассматриваются разные доступные форматы.

Входные сплиты и записи

Как упоминалось в главе 2, входной сплит представляет собой фрагмент входных данных, обрабатываемый одной задачей отображения. Каждое отображение обрабатывает один сплит. Сплит делится на записи, отображение поочередно обрабатывает каждую запись — пару «ключ/значение». Сплиты и записи определяются на логическом уровне: скажем, ничто не требует их привязки к файлам, хотя такое воплощение и является самым распространенным. В контексте базы данных сплит может соответствовать диапазону строк из таблицы, а запись — одной строке этого диапазона (именно это делает `DBInputFormat`, входной формат для чтения данных из реляционной базы данных).

Входные сплиты представлены классом Java `InputSplit` (который, как и все классы, упомянутые в этом разделе, входит в пакет `org.apache.hadoop.mapreduce`)¹:

```
public abstract class InputSplit {  
    public abstract long getLength() throws IOException, InterruptedException;  
    public abstract String[] getLocations() throws IOException,  
        InterruptedException;  
}
```

С `InputSplit` связана длина в байтах и набор мест хранения данных, которые представляют собой обычные строки с именами хостов. Сплит не хранит входные данные; это всего лишь ссылка на них. Информация о местах хранения используется системой MapReduce для размещения задач отображения как можно ближе к данным сплита, а размер используется для упорядочения сплитов, чтобы самые большие обрабатывались в первую очередь для минимизации времени выполнения задания.

¹ Их аналоги для старого MapReduce API находятся в `org.apache.hadoop.mapred`.

Вам как разработчику приложений MapReduce не придется напрямую взаимодействовать с реализациями `InputSplit`, так как они создаются классом `InputFormat`. `InputFormat` отвечает за создание входных сплитов и разбиение их на записи. Прежде чем переходить к конкретным примерам `InputFormat`, давайте кратко рассмотрим использование этого класса в MapReduce. Его интерфейс выглядит так:

```
public abstract class InputFormat<K, V> {  
    public abstract List<InputSplit> getSplits(JobContext context)  
        throws IOException, InterruptedException;  
  
    public abstract RecordReader<K, V>  
        createRecordReader(InputSplit split,  
                           TaskAttemptContext context) throws IOException,  
                                         InterruptedException;  
}
```

Клиент, запускающий задание, вычисляет сплиты вызовом `getSplits()` и передает их трекеру задач. Трекер использует информацию о местах хранения данных для планирования задач отображения, которые будут обрабатывать их на трекерах задач. На трекере задач задача отображения передает сплит методу `createRecordReader()` объекта `InputFormat`, чтобы получить объект `RecordReader` для этого сплита. По сути, `RecordReader` представляет собой обычный итератор для перебора записей; задача отображения использует его для генерирования пар «ключ-значение», передаваемых функции отображения. Пример встречается в методе `run()` класса `Mapper`:

```
public void run(Context context) throws IOException, InterruptedException {  
    setup(context);  
    while (context.nextKeyValue()) {  
        map(context.getCurrentKey(), context.getCurrentValue(), context);  
    }  
    cleanup(context);  
}
```

После вызова `setup()` метод `nextKeyValue()` многократно вызывается для объекта `Context` (который делегирует выполнение одноименному методу `RecordReader`) для заполнения объектов ключей и значений. Ключи и значения получаются от `RecordReader` при помощи `Context` и передаются методу `map()` для обработки.

Когда `RecordReader` доберется до конца потока, метод `nextKeyValue()` возвращает `false`, задача отображения выполняет свой метод `cleanup()`, после чего завершается.

Наконец, обратите внимание на то, что метод `run()` класса `Mapper` объявлен открытым (`public`) и может изменяться пользователями. Реализация `Multi-`

`threadedMapper` запускает отображения параллельно в настраиваемом количестве программных потоков (задается свойством `mapreduce.mapper.multi-threadedmapper.threads`). Для большинства задач обработки данных многопоточность не дает преимуществ перед реализацией по умолчанию. Однако для отображений, в которых обработка каждой записи занимает много времени (например, из-за необходимости обращений к внешним серверам), она позволяет нескольким отображениям выполняться на одной JVM с минимальной конкуренцией.



В коде это не показано, но по соображениям эффективности реализации `RecordReader` возвращают те же объекты ключа и значения при каждом вызове `getCurrentKey()` и `getCurrentValue()`. Вызов `nextKeyValue()` изменяет только содержимое этих объектов. Это обстоятельство может удивить пользователей, которые ожидают, что ключи и значения неизменяемы и не должны использоваться повторно. Сохранение ссылки на объект ключа или значения за пределами метода `map()` создаст проблемы, так как его содержимое может неожиданно измениться. Если вам понадобится сохранить такую ссылку, создайте копию нужного объекта. Например, для объекта `Text` можно воспользоваться копирующим конструктором: `new Text(value)`.

Похожая ситуация возникает и для сверток. В этом случае объекты значений в итераторе свертки используются повторно, поэтому вам придется копировать все объекты, которые должны сохраняться между вызовами итератора (см. листинг 8.14).

FileInputFormat

`FileInputFormat` – базовый класс для всех реализаций `InputFormat`, использующих файлы в качестве источника данных (рис. 7.2). Он предоставляет место для определения файлов, включаемых в качестве входных данных задания, и реализацию генерирования сплитов для входных файлов. Разбиение сплитов на записи выполняется субклассами.

Входные пути FileInputFormat

Входные данные задаются набором путей, что позволяет чрезвычайно гибко ограничивать входные данные задания. `FileInputFormat` предоставляет четыре статических вспомогательных метода для задания входных путей `Job`:

```
public static void addInputPath(Job job, Path path)
public static void addInputPaths(Job job, String commaSeparatedPaths)
```

продолжение ↗

```
public static void setInputPaths(Job job, Path... inputPaths)
public static void setInputPaths(Job job, String commaSeparatedPaths)
```

Методы `addInputPath()` и `addInputPaths()` включают путь (или пути) в список входных данных. Повторный вызов этих методов позволяет построить список путей. Методы `setInputPaths()` задают полный список путей за один вызов (с заменой всех путей, заданных для `Job` в предыдущих вызовах).

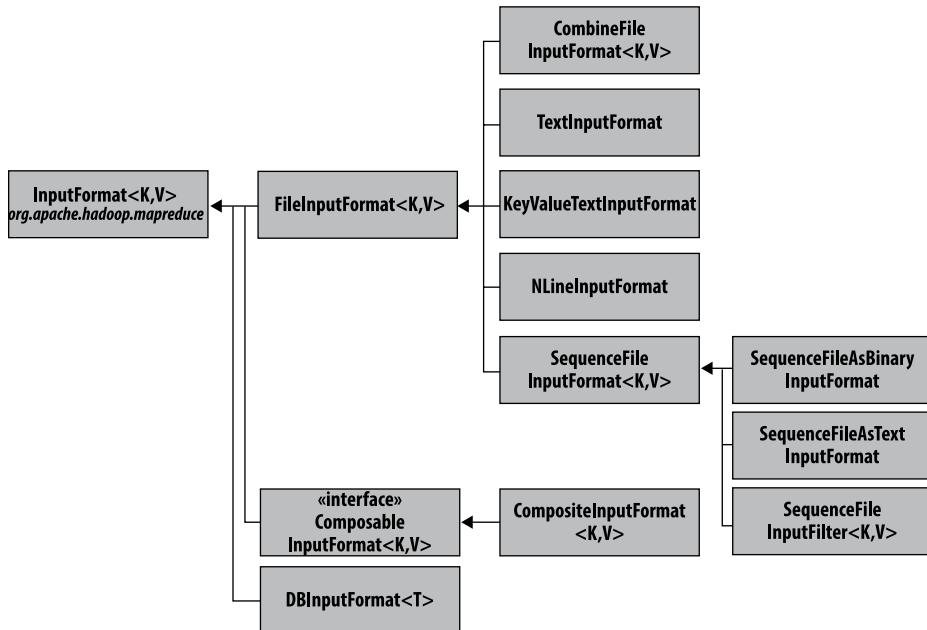


Рис. 7.2. Иерархия форматов входных данных

Путь может представлять файл, каталог или при использовании метасимволов — набор файлов и каталогов. Путь, представляющий каталог, включает в себя все файлы этого каталога как входные данные для задания. За дополнительной информацией о метасимволах обращайтесь к разделу «Шаблоны имён файлов», с. 106.



Содержимое каталога, заданного в качестве входного пути, не обрабатывается рекурсивно. Более того, каталог должен содержать только файлы. Если в нем содержится подкаталог, то последний будет интерпретирован как файл, что приведет к ошибке. В таких случаях следует использовать файловый шаблон или фильтр, выбирающий только файлы из каталога. Также можно задать свойству `mapred.input.dir.recursive` значение `true`, чтобы входной каталог читался в рекурсивном режиме.

Методы `add` и `set` позволяют задавать только включаемые файлы. Чтобы исключить некоторые файлы из ввода, установите фильтр при помощи метода `setInputPathFilter()` объекта `FileInputFormat`:

```
public static void setInputPathFilter(Job job, Class<? extends PathFilter> filter)
```

Фильтры более подробно описаны в разделе «`PathFilter`», с. 108.

Даже если фильтр не установлен, `FileInputFormat` использует фильтр по умолчанию, который исключает скрытые файлы (с именами, начинающимися с точки или символа подчеркивания). Фильтр, установленный вызовом `setInputPathFilter()`, действует как дополнение к фильтру по умолчанию. Иначе говоря, пройдут только не-скрытые файлы, принятые вашим фильтром.

Пути и фильтры также могут настраиваться свойствами конфигурации (табл. 7.4), что может быть удобно для приложений `Streaming` и `Pipes`. В интерфейсах как `Streaming`, так и `Pipes` пути могут задаваться параметром `-input`, поэтому задавать их напрямую обычно не обязательно.

Таблица 7.4. Свойства входных путей и фильтров

Имя свойства	Тип	Значение по умолчанию	Описание
<code>mapred.input.dir</code>	Список путей, разделенных запятыми	Нет	Входные файлы задания. Внутренние запятые в путях должны экранироваться обратной косой чертой. Например, шаблон {a,b} принимает вид {a\,b}
<code>mapred.input.pathFilter.class</code>	Имя класса <code>PathFilter</code>	Нет	Фильтр, применяемый к входным файлам задания

Входные сплиты `FileInputFormat`

Как `FileInputFormat` преобразует заданный набор файлов в сплиты? `FileInputFormat` разбивает только большие файлы (то есть файлы, размер которых превышает размер блока HDFS). Размер сплита обычно совпадает с размером блока HDFS, для большинства приложений это нормально; однако этим значением можно управлять при помощи различных свойств Hadoop, перечисленных в табл. 7.5.

Таблица 7.5. Свойства управления размером сплита

Имя свойства	Тип	Значение по умолчанию	Описание
mapred.min.split.size	int	1	Минимальный действительный размер файлового сплита в байтах
mapred.max.split.size*	long	Long.MAX_VALUE, то есть 9223372036854775807	Максимальный действительный размер файлового сплита в байтах
dfs.block.size	long	64 Мбайт, то есть 67108864	Размер блока HDFS в байтах

* Свойство не поддерживается в старом MapReduce API (за исключением *CombineFileInputFormat*). Вместо этого оно вычисляется косвенно, делением общего размера входных данных задания на ориентировочное количество задач отображения, задаваемое свойством *mapred.map.tasks* (или методом *setNumMapTasks()* объекта *JobConf*). Так как значение *mapred.map.tasks* по умолчанию равно 1, максимальный размер сплита равен размеру входных данных.

Минимальный размер сплита обычно составляет 1 байт, хотя у некоторых форматов устанавливается другая граница. (Например, последовательные файлы вставляют в поток маркеры синхронизации, так что минимальный размер сплита должен быть достаточно большим, чтобы в каждом сплите присутствовала точка синхронизации, по которой объект чтения сможет повторно синхронизироваться по границе записи.)

Приложения могут устанавливать минимальный размер сплита. Если значение превышает размер блока, сплит будет занимать несколько блоков. При использовании HDFS веских причин для этого нет, потому что это увеличит количество блоков, не локальных по отношению к задаче отображения.

Максимальный размер сплита по умолчанию равен максимальному значению, которое может быть представлено типом Java *long*. Значение действует только в том случае, если оно меньше размера блока.

Размер сплита вычисляется по следующей формуле (см. описание метода *computeSplitSize()* в *FileInputFormat*):

```
max(minimumSize, min(maximumSize, blockSize)).
```

По умолчанию

```
minimumSize < blockSize < maximumSize.
```

Соответственно, размер сплита равен `blockSize`. Различные настройки этих параметров и их влияние на итоговый размер сплита приведены в табл. 7.6.

Таблица 7.6. Примеры управления размером сплита

Минимальный размер сплита	Максимальный размер сплита	Размер блока	Размер сплита	Комментарий
1 (по умолчанию)	Long.MAX_VALUE (по умолчанию)	64 Мбайт (по умолчанию)	64 Мбайт	По умолчанию размер сплита совпадает со стандартным размером блока
1 (по умолчанию)	Long.MAX_VALUE (по умолчанию)	128 Мбайт	128 Мбайт	Наиболее естественный способ увеличения размера сплита — увеличение блоков в HDFS (либо изменением <code>dfs.blocksize</code> , либо на уровне файлов во время их конструирования)
128 Мбайт	Long.MAX_VALUE (по умолчанию)	64 Мбайт (по умолчанию)	128 Мбайт	Назначение минимального размера сплита, превышающего размер блока, увеличивает размер сплита — но за счет локальности данных.
1 (по умолчанию)	32 Мбайт	64 Мбайт (по умолчанию)	32 Мбайт	Назначение максимального размера сплита, меньшего размера блока, уменьшает размер сплита.

Малые файлы и `CombineFileInputFormat`

Hadoop лучше работает с небольшим количеством больших файлов, чем с множеством мелких. Одна из причин заключается в том, что `FileInputFormat` генерирует сплиты так, что каждый сплит занимает весь файл или его часть. Если файл очень мал (то есть значительно меньше блока HDFS) и файлов много, каждая задача отображения обрабатывает очень небольшой объем входных данных. Задач будет очень много (по одной на файл), и каждая задача требует лишних затрат ресурсов. Сравните файл 1 Гбайт, разбитый на 16 64-мегабайтных блоков и 10 000 100-килобайтных файлов. Каждый из 10 000 файлов использует по одной задаче отображения, и выполнение задания может замедлиться в десятки и сотни раз по сравнению с эквивалентной задачей с одним входным файлом и 16 задачами отображения.

Ситуацию несколько разряжает класс `CombineFileInputFormat`, спроектированный для эффективной работы с малыми файлами. Если `FileInputFormat` создает по одному сплиту на файл, `CombineFileInputFormat` упаковывает много файлов в один сплит, чтобы у каждой задачи отображения было больше данных для обработки. Важно то, что `CombineFileInputFormat` учитывает локальность узлов и сегментов при принятии решений о том, какие блоки должны размещаться в одном сплите, поэтому по скорости обработки входных данных не уступает типичному заданию MapReduce.

Конечно, сценария с множеством малых файлов следует по возможности избегать, потому для MapReduce критична скорость передачи данных дисков в кластере, а обработка множества малых файлов увеличивает количество операций позиционирования, необходимых для запуска задания. Кроме того, хранение большого количества малых файлов в HDFS неэффективно расходует память узла имен. Один из способов заключается в слиянии множества мелких файлов в более крупные файлы с использованием `SequenceFile`; в этом случае ключами могут быть имена файлов (или константа — например, `NullWritable`, если ключ не нужен), а значениями — содержимое файлов. Пример приведен в листинге 7.4. А если у вас уже имеется множество мелких файлов в HDFS, попробуйте использовать `CombineFileInputFormat`.



Класс `CombineFileInputFormat` хорошо работает не только с малыми файлами; при обработке больших файлов он тоже полезен. По сути, `CombineFileInputFormat` отделяет объем данных, потребляемых задачей отображения, от размера файлового блока в HDFS.

Если ваши отображения способны обработать блок за считанные секунды, вы можете использовать `CombineFileInputFormat` с максимальным размером сплита, кратным числу блоков (для чего свойству `mapred.max.split.size` задается значение в байтах), чтобы каждая задача отображения обрабатывала более одного блока. Вместе с тем выполнение пропорционально меньшего количества отображений снижает общее время обработки, что приводит к сокращению затрат ресурсов на хранение служебной информации и ускорению запуска.

Поскольку `CombineFileInputFormat` является абстрактным классом, не имеющим конкретных классов (в отличие от `FileInputFormat`), вам придется немного потрудиться, чтобы использовать его. (Будем надеяться, что в будущем в библиотеку будут добавлены стандартные реализации.) Например, чтобы получить эквивалент `TextInputFormat` на базе `CombineFileInputFormat`, следует создать конкретный субкласс `CombineFileInputFormat` и реализовать метод `getRecordReader()`.

Предотвращение разбиения

В некоторых приложениях лучше запретить разбиение файлов, так как это позволит одной задаче отображения полностью обработать каждый входной файл. Например, чтобы проверить, отсортированы ли все записи в файле, можно просто перебрать записи по порядку и убедиться в том, что каждая запись не меньше предыдущей. При реализации в виде задачи отображения этот алгоритм будет работать только в том случае, если одно отображение обрабатывает весь файл¹.

Существует пара способов предотвратить разбиение существующего файла. Первый (решение «на скорую руку») основан на увеличении минимального размера сплита до размера, превышающего размер наибольшего файла в системе. Для этого можно задать ему максимальное значение `Long.MAX_VALUE`. Во втором субклассируется конкретный субкласс `FileInputFormat`, который вы хотите использовать, а метод `isSplitable()` переопределяется так, чтобы он возвращал `false`. Например, версия `TextInputFormat` с запрещенным разбиением выглядит так:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
public class NonSplittableTextInputFormat extends TextInputFormat {
    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }
}
```

Информация о файле в задаче отображения

Задача отображения, обрабатывающая файловый входной сплит, может получить информацию о сплите вызовом метода `getInputSplit()` объекта `Context` из `Mapper`. Если формат входных данных является производным от `FileInputFormat`, объект `InputSplit`, возвращаемый методом, может быть преобразован в `FileSplit` для обращения к информации о файле (средства получения информации перечислены в табл. 7.7).

В старом MapReduce API, Streaming и Pipes, та же информация о сплите доступна в свойствах, которые могут читаться из конфигурации отображения. (В старом MapReduce API для этого в вашу реализацию `Mapper` включается реализация `configure()` для получения доступа к объекту `JobConf`.)

Кроме свойств, перечисленных в табл. 7.7, для всех задач отображения и свертки доступны свойства, перечисленные в разделе «Среда выполнения задач», с. 287.

¹ Так реализовано отображение в `SortValidator.RecordStatsChecker`.

Таблица 7.7. Свойства для получения информации о файловых сплитах

Метод FileSplit	Имя свойства	Тип	Описание
getPath()	map.input.file	Path/String	Минимальный действительный размер файлового сплита в байтах
getStart()	map.input.start	long	Смещение начала сплита от начала файла (в байтах)
getLength()	map.input.length	long	Длина сплита в байтах

В следующем разделе показано, как использовать `FileSplit` для обращения к имени файла сплита.

Обработка всего файла как записи

Похожее требование, которое иногда встречается для отображений, — полный доступ ко всему содержимому файла. Запрет разбиения файла идет в правильном направлении, но вам также понадобится объект `RecordReader`, предоставляющий содержимое файла как значение записи. Класс `WholeFileInputFormat` из листинга 7.2 показывает, как это делается.

Листинг 7.2. Реализация InputFormat для чтения всего файла как записи

```
public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(
        InputSplit split, TaskAttemptContext context) throws IOException,
        InterruptedException {
        WholeFileRecordReader reader = new WholeFileRecordReader();
        reader.initialize(split, context);
        return reader;
    }
}
```

`WholeFileInputFormat` определяет формат, в котором ключ не используется (`NullWritable`), а значение соответствует содержимому файла (`BytesWritable`).

Класс определяет два метода. Во-первых, формат обязательно должен указать, что входные файлы не должны подвергаться разбиению; для этого переопределенный метод `isSplitable()` возвращает `false`. Во-вторых, мы определяем реализацию `createRecordReader()`, возвращающую пользовательскую реализацию `RecordReader` из листинга 7.3.

Листинг 7.3. Реализация `RecordReader`, используемая `WholeFileInputFormat` для чтения всего файла как записи

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context)
        throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLength()];
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);
            FSDataInputStream in = null;
            try {
                in = fs.open(file);
                IOUtils.readFully(in, contents, 0, contents.length);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
            processed = true;
            return true;
        }
        return false;
    }

    @Override
    public NullWritable getCurrentKey() throws IOException, InterruptedException {
        продолжение ⇨
```

Листинг 7.3 (продолжение)

```

        return NullWritable.get();
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException,
        InterruptedException {
        return value;
    }

    @Override
    public float getProgress() throws IOException {
        return processed ? 1.0f : 0.0f;
    }

    @Override
    public void close() throws IOException {
        // Ничего не делает
    }
}

```

`WholeFileRecordReader` отвечает за получение объекта `FileSplit` и преобразование его в одну запись с пустым ключом и значением, состоящим из байтов файла. Так как запись всего одна, она либо была обработана `WholeFileRecordReader`, либо не была; соответственно, в класс включается флаг `processed`. Если файл не был обработан при вызове метода `nextKeyValue()`, мы открываем файл, создаем байтовый массив, длина которого соответствует длине файла, и используем класс Hadoop `IOUtils` для копирования файла в байтовый массив. Затем массив присваивается экземпляру `BytesWritable`, переданному методу `next()`, и возвращаем `true`, чтобы сообщить о том, что запись была прочитана.

Далее идут элементарные служебные методы для обращения к текущим типам ключа и значения, получения информации о прогрессе чтения, а также метод `close()`, который вызывается инфраструктурой MapReduce при завершении чтения.

Чтобы продемонстрировать использование `WholeFileInputFormat`, рассмотрим задание MapReduce для упаковки малых файлов в последовательные файлы. Ключом является имя исходного файла, а значением — его содержимое. Программа представлена в листинге 7.4.

Листинг 7.4. Программа MapReduce для упаковки набора малых файлов в объект `SequenceFile`

```

public class SmallFilesToSequenceFileConverter extends Configured
    implements Tool {

    static class SequenceFileMapper

```

```
extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {

    private Text filenameKey;

    @Override
    protected void setup(Context context) throws IOException,
        InterruptedException {
        InputSplit split = context.getInputSplit();
        Path path = ((FileSplit) split).getPath();
        filenameKey = new Text(path.toString());
    }

    @Override
    protected void map(NullWritable key, BytesWritable value, Context context)
        throws IOException, InterruptedException {
        context.write(filenameKey, value);
    }

}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setInputFormatClass(WholeFileInputFormat.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(BytesWritable.class);
    job.setMapperClass(SequenceFileMapper.class);
    return job.waitForCompletion(true) ? 0 : 1;
}

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

Так как используется входной формат `WholeFileInputFormat`, отображению достаточно только узнать имя файла для входного сплита. Для этого оно преобразует `InputSplit` из контекста в объект `FileSplit`, содержащий метод для получения пути к файлу. Путь хранится в объекте `Text` ключа. Функция свертки является тождественной (явно не задается), а в качестве выходного формата используется `SequenceFileOutputFormat`.

Ниже приведен пример запуска для нескольких малых файлов. Мы решили использовать две задачи свертки:

```
% hadoop jar hadoop-examples.jar SmallFilesToSequenceFileConverter \
 -conf conf/hadoop-localhost.xml -D mapred.reduce.tasks=2 input/smallfiles output
```

Создаются два частичных файла в формате `SequenceFile`. Для просмотра их содержимого следует передать параметр `-text` оболочке файловой системы:

```
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00000
hdfs://localhost/user/tom/input/smallfiles/a    61 61 61 61 61 61 61 61 61 61
hdfs://localhost/user/tom/input/smallfiles/c    63 63 63 63 63 63 63 63 63 63
hdfs://localhost/user/tom/input/smallfiles/e
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00001
hdfs://localhost/user/tom/input/smallfiles/b    62 62 62 62 62 62 62 62 62 62
hdfs://localhost/user/tom/input/smallfiles/d    64 64 64 64 64 64 64 64 64 64
hdfs://localhost/user/tom/input/smallfiles/f    66 66 66 66 66 66 66 66 66 66
```

Входные файлы назывались *a*, *b*, *c*, *d*, *e* и *f*, и каждый содержал 10 повторений соответствующей буквы (файл *a* содержал 10 символов «*a*», и т. д.), кроме файла *e*, который был пустым. Мы видим это в текстовом представлении последовательных файлов, в котором выводится имя файла и его шестнадцатеричное представление.

В программу можно внести как минимум одно улучшение. Как упоминалось ранее, выделение одного отображения на файл неэффективно, поэтому вместо `FileInputFormat` лучше субклассировать `CombineFileInputFormat`. Об упаковке файлов в формат Hadoop Archive (вместо `SequenceFile`) рассказано в разделе «HAR», с. 121.

Текстовые входные данные

Hadoop показывает выдающиеся результаты в обработке неструктурированного текста. В этом разделе рассматриваются разновидности `InputFormat`, предоставляемые Hadoop для обработки текста.

TextInputFormat

`TextInputFormat` — формат входных данных, используемый по умолчанию. Каждая запись представляет собой текстовую строку. Ключ (`LongWritable`) определяет смещение начала строки в файле (в байтах). Значение представляет собой содержимое строки, за исключением всех завершителей строк (например, символы новой строки или возврата курсора), упакованное в объект `Text`. Таким образом, файл, содержащий следующий текст:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
```

But his face you could not see,
On account of his Beaver Hat.

преобразуется в один сплит из четырех записей. Записи интерпретируются как следующие пары «ключ-значение»:

(0, On the top of the Crumpty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

Разумеется, ключи *не являются* номерами строк. Этот вариант в общем случае не реализуем, потому что файл разбивается на сплиты по байтам, а не по границам строк. Сплиты обрабатываются независимо друг от друга. Строки подсчитываются по мере обработки, так что определить номер строки можно в сплите, но не в файле.

Однако смещение каждой строки в файле известно каждому сплиту независимо от других сплитов, так как каждый сплит знает размер предшествующих сплитов и просто суммирует их со смещениями для определения глобального смещения в файле. Обычно такого смещения достаточно для приложений, в которых необходим уникальный идентификатор для каждой строки. Его комбинация с именем файла уникальна в пределах файловой системы. Конечно, если все строки имеют фиксированную длину, вычисление номера строки сводится к простому делению смещения на длину.

СВЯЗЬ МЕЖДУ ВХОДНЫМИ СПЛИТАМИ И БЛОКАМИ HDFS

Логические записи, определяемые `FileInputFormat`, редко аккуратно укладываются в блоки HDFS. Например, для `TextInputFormat` логические записи представляют собой строки, которые сплошь и рядом пересекают границы HDFS. На функционировании вашей программы это обстоятельство не отразится — строки не пропускаются, не распадаются и т. д. Однако это обстоятельство следует учитывать, потому что из него следует, что отображения, локальные по данным (то есть отображения, выполняемые на хосте, на котором хранятся их входные данные), будут выполнять удаленное чтение. Небольшие затраты ресурсов, связанные с этим, обычно несущественны.

Пример приведен на рис. 7.3. Один файл делится на строки, границы которых не соответствуют границам блоков HDFS. Сплиты учитывают границы логических записей, поэтому мы видим, что строка 5 входит в первый сплит, хотя он и распространяется на первый и второй блок. Второй сплит начинается со строки 6.



Рис. 7.3. Логические записи и блоки HDFS для TextInputFormat

KeyValueTextInputFormat

Ключи `TextInputFormat`, являющиеся простыми смещениями в файле, обычно не слишком полезны. Часто каждая строка в файле содержит пару «ключ-значение», между которыми находится разделитель (например, символ табуляции). Именно такой вывод производит `TextOutputFormat`, используемая в Hadoop по умолчанию версия `OutputFormat`. Для правильной интерпретации таких файлов `KeyValueTextInputFormat` подойдет.

Разделитель задается при помощи свойства `mapreduce.input.keyvaluelinerecordreader.key.value.separator` (или `key.value.separator.in.input.line` в старом API). По умолчанию используется символ табуляции. Для примера рассмотрим следующий входной файл (знак → представляет символ табуляции):

```
line1→On the top of the Crumpty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat.
```

Как и в примере `TextInputFormat`, входные данные представляют собой один сплит из четырех записей, хотя на этот раз ключами являются последовательности `Text`, предшествующие символу табуляции в каждой строке:

```
(line1, On the top of the Crumpty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat.)
```

NLineInputFormat

С форматами `TextInputFormat` и `KeyValueTextInputFormat` каждому отображению выделяется переменное количество строк исходных данных. Количество зависит от размера сплита и длины строк. Если вы хотите, чтобы отображениям выделялось фиксированное количество строк, используйте формат `NLineInputFormat`. Как и с `TextInputFormat`, ключи содержат смещения внутри файла в байтах, а значения — сами строки.

Под «N» в имени подразумевается количество строк входных данных, выделяемых каждому отображению. При N=1 (по умолчанию) каждому отображению выделяется ровно одна строка входных данных. Значением N управляет свойство `mapreduce.input.lineinputformat.linespermap` (`mapred.line.input.format.linespermap` в старом API). Для примера снова рассмотрим те же четыре строки:

```
On the top of the Crumpty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.
```

Если, например, N=2, то каждый сплит будет содержать две строки. Первое отображение получит две начальные пары «ключ-значение»:

```
(0, On the top of the Crumpty Tree)  
(33, The Quangle Wangle sat,)
```

А второму отображению достанутся две другие пары:

```
(57, But his face you could not see,)  
(89, On account of his Beaver Hat.)
```

Ключи и значения совпадают с теми, которые создает `TextInputFormat`. Различается способ построения сплитов.

Обычно выделение задачи отображения для малого количества строк неэффективно (из-за затрат на создание задачи), но некоторые приложения получают малые объемы данных, проводят с ними интенсивные вычисления, после чего выдают результат. Хорошим примером служат компьютерные модели. Создавая файл входных параметров (по одному на строку), вы можете провести несколько параллельных вычислений, чтобы узнать, как изменяется модель при изменении параметра.



Если вычисления занимают много времени, возможны неприятности с тайм-аутами задач. Если задача не сообщает о прогрессе более 10 минут, трекер задач считает ее сбойной и уничтожает процесс (см. «Сбой задачи», с. 272).

Чтобы защититься от этого, проще всего организовать периодические сообщения о прогрессе — например, выводом статусного сообщения или увеличением счетчика. См. «Как оценивается прогресс в MapReduce?», с. 263.

Другой пример — использование Hadoop для организации загрузки данных из нескольких источников (например, баз данных). Вы создаете входной файл со списком источников, по одному на строку. Затем каждому отображению выделяется один источник, данные из которого загружаются в HDFS. Фаза свертки

этому заданию не нужна, поэтому количество сверток следует обнулить (вызовом `setNumReduceTasks()` для `Job`).

XML

Парсеры XML обычно работают с целыми документами, и, если большой документ XML состоит из нескольких входных сплитов, с разбором их по отдельности возникнут сложности. Конечно, можно обработать весь документ XML (если он не слишком велик) в одном отображении, использовав приемы из раздела «Обработка всего файла как записи», с. 318.

Большие документы XML, состоящие из серии «записей» (фрагментов XML), можно разбить на записи, используя простой поиск строк или регулярные выражения для определения начальных и конечных тегов. Это облегчает проблему при разбиении документа инфраструктурой, потому что следующий начальный тег записи легко находится простым сканированием от начала сплита — подобно тому, как `TextInputFormat` находит границы новых строк.

Для этой цели в Hadoop включен класс `StreamXmlRecordReader` (он находится в пакете `org.apache.hadoop.streaming`, хотя и может использоваться вне Streaming). Чтобы использовать его, выберите формат входных данных `StreamInputFormat` и задайте свойству `stream.recordreader.class` значение `org.apache.hadoop.streaming.StreamXmlRecordReader`. Далее установите свойства конфигурации задания с шаблонами начальных и конечных тегов (за подробностями обращайтесь к документации класса)¹.

Например, Википедия предоставляет возможность получения дампа своих статей в формате XML, хорошо подходящем для параллельной обработки MapReduce с использованием описанного метода. Данные собраны в один большой документ XML, содержащий серию элементов (например, элементы `page` содержат контент страницы и связанные с ним метаданные). Используя `StreamXmlRecordReader`, можно интерпретировать элементы `page` как записи для обработки задачей отображения.

Двоичные входные данные

Технология Hadoop MapReduce не ограничена обработкой текстовых данных. В ней также поддерживаются и двоичные форматы.

¹ Улучшенная реализация входного формата XML содержится в классе `XmllInputFormat` проекта Mahout (<http://mahout.apache.org/>).

SequenceFileInputFormat

В формате последовательных файлов Hadoop хранятся последовательности двоичных пар «ключ-значение». Последовательные файлы хорошо подходят для данных MapReduce — они поддерживают разбиение (наличие точек синхронизации позволяет синхронизироваться с границами записей из произвольной позиции файла — например, от начала сплита), они поддерживают сжатие как часть формата, а также позволяют сохранять произвольные типы с использованием различных сред сериализации (см. «SequenceFile», с. 186).

Чтобы использовать данные из последовательных файлов в качестве входных данных MapReduce, следует выбрать формат `SequenceFileInputFormat`. Ключи и значения определяются последовательным файлом, а вы должны позаботиться о соответствии входных типов отображений. Например, если последовательный файл использует ключи `IntWritable` и значения `Text` (как в примере из главы 4), отображение будет иметь сигнатуру `Mapper<IntWritable, Text, K, V>`, где `K` и `V` — типы выходных ключей и значений отображения.



Хотя это совершенно не следует из имени, `SequenceFileInputFormat` может читать не только последовательные файлы, но и объекты `MapFile`. Обнаружив каталог, в котором должен был находиться последовательный файл, `SequenceFileInputFormat` считает, что работает с объектом `MapFile`, и использует его файл данных. Именно по этой причине в Hadoop нет отдельного класса `MapFileInputFormat`.

SequenceFileAsTextInputFormat

`SequenceFileAsTextInputFormat` — разновидность `SequenceFileInputFormat`, преобразующая ключи и значения последовательного файла в объекты `Text`. Преобразование выполняется вызовом `toString()` для ключей и значений. В этом формате последовательные файлы могут использоваться в качестве входных данных для Streaming.

SequenceFileAsBinaryInputFormat

`SequenceFileAsBinaryInputFormat` — разновидность `SequenceFileInputFormat`, получающая ключи и значения последовательного файла в виде неструктурированных двоичных объектов. Они инкапсулируются в объектах `BytesWritable`, а приложение может интерпретировать нижележащий байтовый массив так, как считает нужным. В сочетании с процессом, создающим последовательные файлы при помощи метода `appendRaw()` класса `SequenceFile.Writer` или `SequenceFileAsBinaryOutputFormat`, это открывает возможность использования с MapReduce

двоичных типов данных (упакованных в последовательные файлы), хотя существует и более элегантная альтернатива — подключение к механизму сериализации Hadoop (см. «Программные среды сериализации», с. 189).

Множественные источники входных данных

Хотя входные данные задания MapReduce могут состоять из нескольких файлов (определенными комбинацией файловых шаблонов, фильтров и простых путей), вся совокупность входных данных интерпретируется одним объектом `InputFormat` и одной функцией отображения. Однако часто бывает так, что формат данных эволюционирует со временем, и функцию отображения приходится писать так, чтобы она работала со всеми унаследованными форматами. Также возможны источники данных, предоставляющие однотипные данные в разных форматах. Такое бывает при соединении наборов данных; см. «Соединения на стороне свертки», с. 370. Например, один источник может содержать простой текст, разделенный табуляциями, а другой — двоичный последовательный файл. И даже если источники имеют одинаковый формат, они могут иметь разные представления, а следовательно, должны разбираться по-разному.

Подобные проблемы элегантно решаются при помощи класса `MultipleInputs`, который позволяет задать используемые объекты `InputFormat` и `Mapper` для отдельных путей. Например, если мы хотим объединить метеорологические данные от UK Met Office¹ с данными NCDC для поиска максимальных температур, входные данные могут настраиваться следующим образом:

```
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

Этот код заменяет обычные вызовы `FileInputFormat.addInputPath()` и `job.setMapperClass()`. И данные Met Office, и данные NCDC хранятся в текстовом виде, поэтому в каждом случае используется `TextInputFormat`. Однако формат строк двух источников данных различается, поэтому мы используем две разные функции отображения. `MaxTemperatureMapper` читает входные данные NCDC и извлекает поля года и температуры. `MetOfficeMaxTemperatureMapper` читает входные данные Met

¹ Вообще говоря, данные Met Office доступны только для исследовательского и академического сообщества. Впрочем, небольшой объем ежемесячных погодных данных доступен по адресу <http://www.metoffice.gov.uk/climate/uk/stationdata/>.

Office и извлекает поля года и температуры. Здесь важно то, что выходные данные отображений имеют одинаковые типы, так как свертки (все из которых относятся к одному типу) работают с объединенными выходными данными отображений и ничего не знают о том, что они были сгенерированы разными отображениями.

Класс `MultipleInputs` содержит перегруженную версию `addInputPath()`, которая не получает отображение:

```
public static void addInputPath(Job job, Path path,
                                Class<? extends InputFormat> inputFormatClass)
```

Она удобна в ситуациях, когда у вас имеется только одно отображение (заданное методом `setMapperClass()` класса `Job`) с несколькими входными форматами.

Операции ввода (и вывода) с базами данных

`DBInputFormat` — входной формат для чтения информации из реляционных баз данных с использованием JDBC. Будьте внимательны, чтобы не перегрузить базу данных запуском слишком большого количества отображений. По этой причине `DBInputFormat` лучше использовать для загрузки относительно малых наборов данных — например, для объединения больших наборов данных из HDFS с использованием `MultipleInputs`. Соответствующий выходной формат `DBOutputFormat` удобен для направления вывода заданий (умеренного размера) в базу данных¹.

Для перемещения данных между реляционными базами данных и HDFS также можно воспользоваться Sqoop (см. главу 15).

Класс HBase `TableInputFormat` спроектирован для того, чтобы программы MapReduce могли работать с данными, хранящимися в таблицах HBase. Класс `TableOutputFormat` предназначен для записи выходных данных MapReduce в таблицу HBase.

Форматы выходных данных

Hadoop поддерживает форматы выходных данных, соответствующие входным форматам из предыдущего раздела. Иерархия классов `OutputFormat` изображена на рис. 7.4.

¹ О том, как использовать эти форматы, рассказано в статье «Database Access with Hadoop» в блоге Аарона Кимболла.

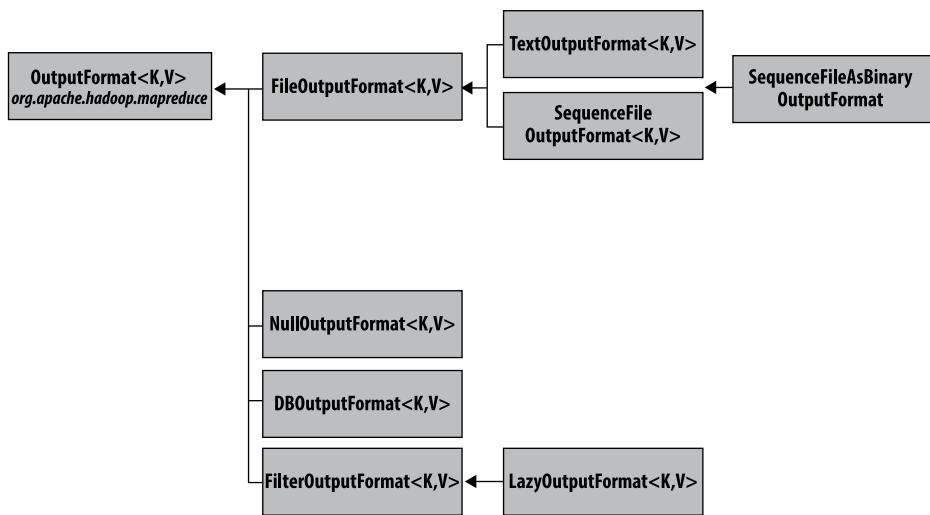


Рис. 7.4. Иерархия классов `OutputFormat`

Текстовые выходные данные

Выходной формат по умолчанию `TextOutputFormat` выводит записи в виде строк текста. Ключи и значения могут относиться к любому типу, так как `TextOutputFormat` преобразует их в строки вызовом `toString()`. Каждая пара «ключ-значение» разделяется символом табуляции, хотя разделитель можно сменить при помощи свойства `mapreduce.output.textoutputformat.separator` (`mapred.textoutputformat.separator` в старом API). Парным форматом чтения по отношению к `TextOutputFormat` в данном случае является `KeyValueTextInputFormat`, так как он разбивает строки на пары «ключ-значение» на основании настраиваемого разделителя (см. «`KeyValueTextInputFormat`», с. 324).

Вы можете исключить из вывода ключ или значение (или оба, что сделает этот формат эквивалентным `NullOutputFormat`, не выводящим ничего), используя тип `NullWritable`. Разделитель при этом выводиться не будет, что делает вывод пригодным для чтения с использованием формата `TextInputFormat`.

Двоичные выходные данные

SequenceFileOutputFormat

Как подсказывает название, `SequenceFileOutputFormat` записывает свой вывод в формате последовательных файлов. Такой вывод данных хорошо подойдет

в качестве ввода следующего задания MapReduce, потому что он компактен и поддерживает сжатие. Для управления сжатием используются статические методы `SequenceFileOutputFormat` (см. «Использование сжатия в MapReduce», с. 137). Пример использования `SequenceFileOutputFormat` приведен в разделе «Сортировка», с. 350.

SequenceFileAsBinaryOutputFormat

`SequenceFileAsBinaryOutputFormat` является «двойником» `SequenceFileAsBinaryInputFormat`; он записывает ключи и значения в низкоуровневом двоичном формате в контейнер `SequenceFile`.

MapFileOutputFormat

`MapFileOutputFormat` записывает выходные данные в формате `MapFile`. Ключи `MapFile` должны добавляться по порядку; следовательно, вы должны проследить за тем, чтобы свертки выдавали ключи в отсортированном порядке.



Входные ключи сверток заведомо отсортированы, но выходные ключи находятся под контролем функции свертки, и в общем контракте MapReduce нет никаких упоминаний о том, что выходные ключи свертки должны быть как-либо упорядочены. Дополнительные ограничения сортировки выходных ключей свертки необходимы только для `MapFileOutputFormat`.

Множественный вывод

`FileOutputFormat` и его субклассы создают набор файлов в выходном каталоге. На каждую задачу свертки приходится один файл, имена файлов присваиваются по номерам разделов: *part-r-00000*, *part-r-00001* и т. д. В некоторых ситуациях требуется более основательный контроль за именами файлов или возможность создания нескольких файлов в каждой задаче свертки. MapReduce включает в себя класс `MultipleOutputs`, который поможет вам в этом¹.

¹ В старой версии MapReduce API существует два класса для создания множественного вывода: `MultipleOutputFormat` и `MultipleOutputs`. По сути, класс `MultipleOutputs` обладает большей функциональностью, а `MultipleOutputFormat` — более широкими средствами контроля за структурой выходных каталогов и именами файлов. Класс `MultipleOutputs` в новом API сочетает лучшие особенности двух классов старого API. На сайте книги приведены аналоги примеров этого раздела для старого API с использованием как `MultipleOutputs`, так и `MultipleOutputFormat`.

Пример: разделение данных

Допустим, нам потребовалось разделить набор метеорологических данных по метеостанциям. Задание должно создавать для каждой станции один выходной файл, содержащий все записи для данной станции.

Одно из возможных решений — создать отдельную задачу свертки для каждой метеостанции. Для этого необходимы два условия. Во-первых, нужно написать разделитель, который помещает записи, относящиеся к одной метеостанции, в один раздел. Во-вторых, нужно задать количество задач свертки в задании, совпадающее с количеством метеостанций. Разделитель выглядит примерно так:

```
public class StationPartitioner extends Partitioner<LongWritable, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    public int getPartition(LongWritable key, Text value, int numPartitions) {
        parser.parse(value);
        return getPartition(parser.getStationId());
    }

    private int getPartition(String stationId) {
        ...
    }
}
```

Метод `getPartition(String)`, реализация которого не приводится, преобразует идентификатор станции в индекс раздела. Для работы методу необходим список всех идентификаторов станций; метод просто возвращает индекс идентификатора из списка.

У такого решения есть два недостатка. Во-первых, количество разделов должно быть известно до выполнения задания, как и количество метеостанций. Хотя NCDC предоставляет метаданные по своим станциям, нет никаких гарантий, что идентификаторы, встреченные в данных, будут совпадать с идентификаторами в метаданных. Станция, которая присутствует в метаданных, но не в данных, лишь попусту расходует слот свертки. Что еще хуже, станции, которая присутствует в данных, но не в метаданных, слот свертки вообще не достанется; она будет потеряна. Конечно, можно написать задание для извлечения уникальных идентификаторов станций, но это слишком примитивное решение.

Второй недостаток менее очевиден. Обычно жестко фиксировать количество разделов в приложении нежелательно, потому что разделы получаются слишком

мелкими или заметно различаются по размерам. Выполнение малого объема работы во многих свертках — неэффективный способ организации работы; гораздо лучше поручить больше работы меньшему количеству сверток, так как это сократит непроизводительные затраты ресурсов на выполнение задач. Кроме того, будет нелегко избежать разных размеров разделов. Разные метеостанции могут собирать разные объемы данных; представьте, что одна станция открылась год назад, а другая собирает данные уже 100 лет. Если некоторые задачи свертки выполняются значительно дольше других, они будут доминировать во времени выполнения задания, которое займет больше времени, чем реально необходимо.



Есть две особые ситуации, в которых можно разрешить приложению задавать количество разделов (или, что то же самое, количество задач свертки):

- 1. Нуль сверток.** Вырожденный случай: разделов нет, так как приложение выполняет только задачи отображения.
- 2. Одна свертка.** Иногда бывает удобно запускать маленькие задания для объединения вывода предшествующих заданий в один файл. Это следует делать только для небольших объемов данных, которые легко обрабатываются одной сверткой.

Гораздо эффективнее поручить определение количества разделов кластеру; чем больше в кластере будет доступно слотов свертки, тем быстрее завершится задание. Именно поэтому так хорошо работает используемая по умолчанию реализация `HashPartitioner`: она работает с любым количеством разделов и гарантирует равномерное распределение ключей, от которого создаются более равномерно распределенные разделы.

Если вернуться к использованию `HashPartitioner`, каждый раздел будет содержать данные нескольких метеостанций. Следовательно, чтобы создать отдельный файл для каждой метеостанции, необходимо организовать запись нескольких файлов для каждой свертки. В этом нам поможет `MultipleOutputs`.

MultipleOutputs

`MultipleOutputs` позволяет записывать данные в файлы, имена которых определяются выходными ключами и значениями (или произвольными строками). Это позволяет каждой свертке (или отображению в задании, состоящем только из отображений) создать более одного файла. Имена файлов строятся в формате `имя-t-nnnnn` для вывода отображений и `имя-r-nnnnn` для вывода сверток, где `имя` — произвольная строка, заданная в программе, а `nnnnn` — целый номер части (нумерация начинается с нуля). Номер части предотвращает возможные конфликты выходных данных разных разделов (отображения или свертки) при совпадении

имен. Программа в листинге 7.5 демонстрирует применение `MultipleOutputs` для разделения набора данных по метеостанциям.

Листинг 7.5. Разделение всего набора данных по файлам, имена которых соответствуют идентификаторам метеостанций

```
public class PartitionByStationUsingMultipleOutputs extends Configured
    implements Tool {

    static class StationMapper
        extends Mapper<LongWritable, Text, Text, Text> {

        private NcdcRecordParser parser = new NcdcRecordParser();

        @Override
        protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            parser.parse(value);
            context.write(new Text(parser.getStationId()), value);
        }
    }

    static class MultipleOutputsReducer
        extends Reducer<Text, Text, NullWritable, Text> {

        private MultipleOutputs<NullWritable, Text> multipleOutputs;
        @Override
        protected void setup(Context context)
            throws IOException, InterruptedException {
            multipleOutputs = new MultipleOutputs<NullWritable, Text>(context);
        }
        @Override
        protected void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
            for (Text value : values) {
                multipleOutputs.write(NullWritable.get(), value, key.toString());
            }
        }
        @Override
        protected void cleanup(Context context)
            throws IOException, InterruptedException {
            multipleOutputs.close();
        }
    }
}
```

```
@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setMapperClass(StationMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setReducerClass(MultipleOutputsReducer.class);
    job.setOutputKeyClass(NullWritable.class);
    return job.waitForCompletion(true) ? 0 : 1;
}

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

В классе свертки, где генерируется вывод, мы конструируем экземпляр `MultipleOutputs` в методе `setup()` и присваиваем его переменной экземпляра. Затем в методе `reduce()` для записи вывода используется экземпляр `MultipleOutputs`. Метод `write()` получает ключ и значение, а также имя. В качестве имени мы используем идентификатор метеостанции, так что конечным результатом является создание выходных файлов со схемой имен *идент_станции-r-nnnn*.

В одном из запусков нескольким выходным файлам были присвоены следующие имена:

```
output/010010-99999-r-00027
output/010050-99999-r-00013
output/010100-99999-r-00015
output/010280-99999-r-00014
output/010550-99999-r-00000
output/010980-99999-r-00011
output/011060-99999-r-00025
output/012030-99999-r-00029
output/012350-99999-r-00018
output/012620-99999-r-00004
```

Базовый путь, указанный в методе `write()` класса `MultipleOutputs`, интерпретируется относительно выходного каталога. Так как он может содержать разделители файловых путей (/), появляется возможность создания подкаталогов произвольной глубины. Например, следующая модификация делит данные по станциям

и по годам, чтобы подкаталоги с данными разных лет находились в каталоге, имя которого соответствует идентификатору станции (например, `029070-99999/1901/part-r-00000`):

```
@Override  
protected void reduce(Text key, Iterable<Text> values, Context context)  
    throws IOException, InterruptedException {  
    for (Text value : values) {  
        parser.parse(value);  
        String basePath = String.format("%s/%s/part",  
            parser.getStationId(), parser.getYear());  
        multipleOutputs.write(NullWritable.get(), value, basePath);  
    }  
}
```

`MultipleOutputs` делегирует выполнение операции классу `OutputFormat` отображения; в нашем примере это `TextOutputFormat`, но возможны и более сложные ситуации. За дополнительной информацией обращайтесь к документации Java.

Отложенный вывод

Субклассы `FileOutputFormat` создают выходные файлы (`part-r-nnnnn`), даже если они остаются пустыми. В некоторых приложениях создание пустых файлов нежелательно; в таких ситуациях на помощь приходит `LazyOutputFormat` — формат выходных данных, с которым выходной файл создается только при выдаче первой записи для заданного раздела. Чтобы использовать этот класс, вызовите его метод `setOutputFormatClass()` с объектом `JobConf` и базовым выходным форматом.

Streaming и Pipes поддерживают параметр `-lazyOutput` для включения `LazyOutputFormat`.

Вывод в базы данных

Выходные форматы для записи в реляционные базы данных и HBase описаны в разделе «Операции ввода (и вывода) с базами данных», с. 329.

8

Дополнительные возможности MapReduce

В этой главе рассматриваются некоторые нетривиальные возможности MapReduce, в числе которых счетчики, сортировка и соединение наборов данных.

Счетчики

Часто требуется собрать об анализируемых данных некоторую информацию, не имеющую прямого отношения к выполняемому анализу. Например, если при подсчете недействительных записей вдруг обнаруживается, что их доля в общем наборе данных аномально велика, стоит проверить, почему столько записей помечено как недействительные — может, в коде проверки записей допущена ошибка? Или исходные данные действительно содержат слишком много ошибок — в этом случае размер набора данных стоит увеличить, чтобы количество хороших записей было достаточно велико для содержательного анализа?

Счетчики являются полезным инструментом для сбора статистики о задании: как с целью контроля качества, так и для статистики уровня приложения. Данные счетчиков также пригодятся в процессе диагностики проблем. Вместо вывода сообщений в журнал из задач отображения или свертки часто бывает лучше воспользоваться счетчиком для регистрации определенных условий. Кроме того, что

для больших распределенных заданий данные счетчиков обрабатывать намного удобнее, чем журнальные сообщения, вы сразу получаете информацию о количестве возникновений того или иного условия. Собрать эту информацию из набора журнальных файлов намного сложнее.

Встроенные счетчики

Hadoop ведет для каждого задания набор встроенных счетчиков для вычисления различных метрик. Например, имеются счетчики количества обработанных байтов и записей; по ним можно убедиться в том, было ли обработано ожидаемое количество данных и сгенерирован ожидаемый объем выходных данных.

Счетчики делятсяся на группы. Для встроенных счетчиков определены группы, перечисленные в табл. 8.1.

Таблица 8.1. Группы встроенных счетчиков

Группа	Имя	Ссылка
Счетчики задач MapReduce	org.apache.hadoop.mapred.Task\$Counter (1.x) org.apache.hadoop.mapreduce.TaskCounter (после 1.x)	Таблица 8.2
Счетчики файловой системы	FileSystemCounters (1.x) org.apache.hadoop.mapreduce.FileSystemCounter (после 1.x)	Таблица 8.3
Счетчики FileInputFormat	org.apache.hadoop.mapred.FileInputFormat\$Counter (1.x) org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter (после 1.x)	Таблица 8.4
Счетчики FileOutputFormat	org.apache.hadoop.mapred.FileOutputFormat\$Counter (1.x) org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter (после 1.x)	Таблица 8.5
Счетчики заданий	org.apache.hadoop.mapred.JobInProgress\$Counter (1.x) org.apache.hadoop.mapreduce.JobCounter (после 1.x)	Таблица 8.6

Каждая группа содержит *счетчики задач* (обновляемые по мере выполнения заданий) и *счетчики заданий* (обновляемые по мере выполнения задания). Мы рассмотрим оба типа в следующих разделах.

Счетчики задач

Счетчики задач собирают информацию о задачах в ходе их выполнения; результаты объединяются по всем задачам, входящим в задание. Например, счетчик **MAP_INPUT_RECORDS** подсчитывает входные записи, прочитанные каждой задачей отображения, и объединяет информацию по всем задачам отображения, входящим в задание. Окончательное значение определяет общее количество входных записей для всего задания.

Счетчики задач отслеживаются для каждой попытки выполнения задачи, информация периодически отправляется трекеру задач, а затем трекеру заданий для глобального объединения. (см. «Обновления состояния», с. 261). Учтите, что в YARN используется другая схема передачи информации; см. «YARN (MapReduce 2)», с. 265. Счетчики задач каждый раз передаются полностью (вместо величины изменений по сравнению с последней передачи), так как это предотвращает ошибки, связанные с потерей сообщений. Следует учитывать, что во время выполнения задания счетчики могут уменьшаться в случае сбоя задачи.

Значения счетчиков актуальны только после успешного завершения задания. Впрочем, некоторые счетчики предоставляют полезную диагностическую информацию в ходе выполнения задачи; их значения полезно отслеживать через веб-интерфейс. Например, счетчики **PHYSICAL_MEMORY_BYTES**, **VIRTUAL_MEMORY_BYTES** и **COMMITTED_HEAP_BYTES** предоставляют информацию об изменениях в использовании памяти во время конкретной попытки выполнения задачи.

В категорию встроенных счетчиков задач входят счетчики из группы счетчиков задач MapReduce (табл. 8.2), а также счетчики из файловых групп (табл. 8.3, 8.4 и 8.5).

Таблица 8.2. Встроенные счетчики задач MapReduce

Счетчик	Описание
Входные записи отображений (MAP_INPUT_RECORDS)	Количество входных записей, прочитанных всеми отображениями в задании. Счетчик увеличивается при каждом чтении записи из RecordReader и передаче ее методу map() отображения
Пропущенные записи отображений (MAP_SKIPPED_RECORDS)	Количество входных записей, пропущенных всеми отображениями задания. См. «Пропуск некорректных записей», с. 293
Входные байты отображений (MAP_INPUT_BYTES)	Количество байт в несжатом вводе, прочитанном всеми отображениями задания. Счетчик увеличивается при каждом чтении записи из RecordReader и передаче ее методу map() отображения

продолжение ⇨

Таблица 8.2 (продолжение)

Счетчик	Описание
Низкоуровневые байты сплита (SPLIT_RAW_BYTES)	Количество байт в объектах входных сплитов, прочитанных отображениями. Эти объекты представляют метаданные сплитов (то есть смещение и длину в файле), а не собственно данные сплитов, поэтому их общий размер должен быть небольшим
Выходные записи отображений (MAP_OUTPUT_RECORDS)	Количество выходных записей, производимых всеми отображениями в задании. Счетчик увеличивается при каждом вызове метода collect() для объекта OutputCollector отображения
Выходные байты отображений (MAP_OUTPUT_BYTES)	Количество байт в несжатом выводе, произведенном всеми отображениями задания. Счетчик увеличивается при каждом вызове метода collect() для объекта OutputCollector отображения
Материализованные выходные байты отображения (MAP_OUTPUT_MATERIALIZED_BYTES)	Количество байт в выводе отображения, фактически записанном на диск. Если сжатие вывода отображений включено, это отразится на значении счетчика
Входные записи комбинаторов (COMBINE_INPUT_RECORDS)	Количество входных записей, прочитанных всеми комбинаторами в задании (если они определены). Счетчик увеличивается при каждом чтении в процессе перебора значений итератором комбинирующей функции. Следует учесть, что счетчик отражает количество значений, прочитанных комбинирующей функцией, а не количество разных групп ключей (эта метрика была бы бесполезной, потому что количество групп не обязательно соответствует количеству ключей в комбинирующей функции; см. «Комбинирующие функции», с. 68, и «Тасовка и сортировка», с. 279)
Выходные записи комбинирующей функции (COMBINE_OUTPUT_RECORDS)	Количество выходных записей, производимых всеми комбинирующими функциями в задании (если они определены). Счетчик увеличивается при каждом вызове метода collect() для объекта OutputCollector отображения
Входные группы сверток (REDUCE_INPUT_GROUPS)	Количество групп ключей, прочитанных всеми свертками задания. Счетчик увеличивается при каждом вызове метода reduce() свертки
Входные записи сверток (REDUCE_INPUT_RECORDS)	Количество входных записей, прочитанных всеми свертками в задании. Счетчик увеличивается при каждом чтении в процессе перебора значений итератором свертки. После того, как свертки прочитают весь ввод, значение счетчика должно совпадать с количеством выходных записей отображений

Счетчик	Описание
Выходные записи сверток (REDUCE_OUTPUT_RECORDS)	Количество выходных записей, производимых всеми свертками в задании. Счетчик увеличивается при каждом вызове метода collect() для объекта OutputCollector свертки
Пропущенные группы сверток (REDUCE_SKIPPED_GROUPS)	Количество групп ключей, пропущенных всеми свертками в задании. См. «Пропуск некорректных записей», с. 293
Пропущенные записи сверток (REDUCE_SKIPPED_RECORDS)	Количество входных записей, пропущенных всеми свертками в задании
Байты сверток после тасовки (REDUCE_SHUFFLE_BYTES)	Количество байт вывода отображений, скопированных в свертки в процессе тасовки
Выгруженные записи (SPILLED_RECORDS)	Количество записей, выгруженных на диск всеми задачами отображения и свертки в задании
Процессорное время в миллисекундах (CPU_MILLISECONDS)	Общее процессорное время выполнения задачи в миллисекундах — по данным /proc/cpuinfo
Байты физической памяти (PHYSICAL_MEMORY_BYTES)	Физическая память (в байтах), используемая задачей, — по данным /proc/meminfo
Байты виртуальной памяти (VIRTUAL_MEMORY_BYTES)	Виртуальная память (в байтах), используемая задачей, — по данным /proc/meminfo
Байты доступной памяти (COMMITTED_HEAP_BYTES)	Общий объем памяти (в байтах), доступной JVM, — по данным Runtime.getRuntime().totalMemory()
Время уборки мусора (GC_TIME_MILLIS)	Время, затраченное задачами на уборку мусора (в миллисекундах), — по данным GarbageCollectorMXBean.getCollectionTime(). (Недоступно в 1.x.)
Тасованные выходные файлы отображений (SHUFFLED_MAPS)	Количество выходных файлов отображений, переданных сверткам в процессе тасовки. См. «Тасовка и сортировка», с. 279. (Недоступно в 1.x)
Неудачные операции тасовки (FAILED_SHUFFLE)	Количество неудачных операций копирования вывода отображений в процессе тасовки. (Недоступно в 1.x)
Слияние вывода отображений (MERGED_MAP_OUTPUTS)	Количество выходных результатов отображений, слитых на стороне свертки в процессе тасовки. (Недоступно в 1.x)

Таблица 8.3. Встроенные счетчики файловой системы

Счетчик	Описание
Байты, прочитанные из файловой системы (BYTES_READ)	Количество байт, прочитанных из файловой системы задачами отображения и свертки. Отдельный счетчик создается для каждой файловой системы: Local, HDFS, S3, KFS и т. д.
Байты, записанные в файловую систему (BYTES_WRITTEN)	Количество байт, записанных в файловую систему задачами отображения и свертки

Таблица 8.4. Встроенные счетчики FileInputFormat

Счетчик	Описание
Прочитанные байты (BYTES_READ)	Количество байт, прочитанных задачами отображения через FileInputFormat

Таблица 8.5. Встроенные счетчики FileOutputFormat

Счетчик	Описание
Прочитанные байты (BYTES_WRITTEN)	Количество байт, записанных задачами отображения (для заданий, содержащих только отображения) или задачами свертки через FileOutputFormat

Счетчики заданий

Счетчики заданий (табл. 8.6) ведутся трекером заданий (или контроллером приложений в YARN), поэтому их не нужно передавать по сети (в отличие от других счетчиков, в том числе и определяемых пользователями). В них накапливается статистика уровня задания, а не значения, изменяющиеся во время выполнения задания. Например, счетчик TOTAL_LAUNCHED_MAPS подсчитывает количество заданий отображения, запущенных в ходе выполнения задания (включая сбойные задачи).

Таблица 8.6. Встроенные счетчики заданий

Счетчик	Описание
Запущенные задачи отображений (TOTAL_LAUNCHED_MAPS)	Количество запущенных задач отображений. Включает задачи, запущенные в результате спекулятивного выполнения

Счетчик	Описание
Запущенные задачи свертки (TOTAL_LAUNCHED_REDUCES)	Количество запущенных задач свертки. Включает задачи, запущенные в результате спекулятивного выполнения
Запущенные суперзадачи (TOTAL_LAUNCHED_UBERTASKS)	Количество запущенных суперзадач (см. «YARN (MapReduce 2)», с. 265). (Только для MapReduce на базе YARN)
Отображения в суперзадачах (NUM_UBER_SUBMAPS)	Количество отображений в суперзадачах. (Только для MapReduce на базе YARN)
Свертки в суперзадачах (NUM_UBER_SUBREDUCES)	Количество сверток в суперзадачах. (Только для MapReduce на базе YARN)
Сбойные задачи отображения (NUM_FAILED_MAPS)	Количество задач отображений, при выполнении которых произошел сбой. Возможные причины сбоев описаны в разделе «Сбой задачи», с. 272
Сбойные задачи свертки (NUM_FAILED_REDUCES)	Количество задач свертки, при выполнении которых произошел сбой
Сбойные суперзадачи (NUM_FAILED_UBERTASKS)	Количество суперзадач, при выполнении которых произошел сбой. (Только для MapReduce на базе YARN.)
Задачи отображения с локальностью данных (DATA_LOCAL_MAPS)	Количество задач отображения, выполняемых на том узле, на котором находятся их входные данные
Задачи отображения с сегментной локальностью (RACK_LOCAL_MAPS)	Количество задач отображения, выполняемых в том сегменте, в котором находятся их входные данные, но не обладающих локальностью данных
Задачи отображения с другой локальностью (OTHER_LOCAL_MAPS)	Количество задач отображения, выполняемых на узлах сегмента, отличного от сегмента хранения их входных данных. Межсегментные каналы связи — дефицитный ресурс; Hadoop старается размещать задачи отображения вблизи от их входных данных, поэтому значение счетчика должно быть низким. См. рис. 2.2
Общее время задач отображения (SLOTS_MILLIS_MAPS)	Общее время, затраченное на выполнение задач отображения (в миллисекундах). Включает задачи, запущенные в результате спекулятивного выполнения
Общее время задач свертки (SLOTS_MILLIS_REDUCES)	Общее время, затраченное на выполнение задач свертки (в миллисекундах). Включает задачи, запущенные в результате спекулятивного выполнения

Таблица 8.6 (продолжение)

Счетчик	Описание
Общее время ожидания задач отображения после резервирования слотов (FOLLOW_SLOTS_MILLIS_MAPS)	Общее время, затраченное на ожидание после резервирования слотов для задач отображения. Резервирование слотов — функция планировщика Capacity Scheduler для задач с высокими затратами памяти; см. «Ограничения памяти задач», с. 410. Не используется в MapReduce на базе YARN
Общее время ожидания задач свертки после резервирования слотов (FOLLOW_SLOTS_MILLIS_REDUCES)	Общее время, затраченное на ожидание после резервирования слотов для задач свертки. Резервирование слотов — функция планировщика Capacity Scheduler для задач с высокими затратами памяти; см. «Ограничения памяти задач», с. 410. Не используется в MapReduce на базе YARN

Счетчики Java, определяемые пользователем

MapReduce позволяет пользовательскому коду определять счетчики, которые затем инкрементируются так, как потребуется, в задаче отображения или свертки. Счетчики определяются перечислением Java, которое служит для группировки связанных счетчиков. Задание может определять произвольное количество перечислений, каждое из которых имеет произвольное количество полей. Имя перечисления определяет имя группы, а поля перечисления определяют имена счетчиков. Счетчики глобальны: инфраструктура MapReduce объединяет их по всем отображениям и сверткам, чтобы вычислить общий итог в конце задания.

Мы создали в главе 5 счетчики для подсчета неправильно сформированных записей в метеорологическом наборе данных. Программа в листинге 8.1 расширяет этот пример: в ней также подсчитываются отсутствующие записи и вычисляется распределение кодов качества температуры.

Листинг 8.1. Приложение для поиска максимальной температуры с подсчетом отсутствующих и некорректных записей, а также кодов качества

```
public class MaxTemperatureWithCounters extends Configured implements Tool {

    enum Temperature {
        MISSING,
        MALFORMED
    }

    static class MaxTemperatureMapperWithCounters
```

```
extends Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        parser.parse(value);
        if (parser.isValidTemperature()) {
            int airTemperature = parser.getAirTemperature();
            context.write(new Text(parser.getYear()),
                new IntWritable(airTemperature));
        } else if (parser.isMalformedTemperature()) {
            System.err.println("Ignoring possibly corrupt input: " + value);
            context.getCounter(Temperature.MALFORMED).increment(1);
        } else if (parser.isMissingTemperature()) {
            context.getCounter(Temperature.MISSING).increment(1);
        }

        // Динамический счетчик
        context.getCounter("TemperatureQuality",
            parser.getQuality()).increment(1);
    }
}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapperClass(MaxTemperatureMapperWithCounters.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    return job.waitForCompletion(true) ? 0 : 1;
}

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

Чтобы увидеть, что делает эта программа, лучше всего запустить ее для полного набора данных:

```
% hadoop jar hadoop-examples.jar MaxTemperatureWithCounters \
  input/ncdc/all output-counters
```

После успешного завершения задания в конце клиент задания выводит значения счетчиков. Вот те данные, которые нас интересуют:

```
12/02/04 19:46:38 INFO mapred.JobClient: TemperatureQuality
12/02/04 19:46:38 INFO mapred.JobClient: 2=1246032
12/02/04 19:46:38 INFO mapred.JobClient: 1=973422173
12/02/04 19:46:38 INFO mapred.JobClient: 0=1
12/02/04 19:46:38 INFO mapred.JobClient: 6=40066
12/02/04 19:46:38 INFO mapred.JobClient: 5=158291879
12/02/04 19:46:38 INFO mapred.JobClient: 4=10764500
12/02/04 19:46:38 INFO mapred.JobClient: 9=66136858
12/02/04 19:46:38 INFO mapred.JobClient: Air Temperature Records
12/02/04 19:46:38 INFO mapred.JobClient: Malformed=3
12/02/04 19:46:38 INFO mapred.JobClient: Missing=66136856
```

Динамические счетчики

В представленном коде используется динамический счетчик — то есть счетчик, не определяемый перечислением Java. Так как поля перечисления Java определяются во время компилирования, создавать новые счетчики «на ходу» с использованием перечислений не удастся. В данном случае нужно подсчитать распределение кодов качества температуры. И хотя возможные значения кода качества определены в спецификации формата, удобнее использовать динамический счетчик для выдачи фактических значений. Используемый метод объекта `Reporter` получает имена группы и счетчика в формате `String`:

```
public void incrCounter(String group, String counter, long amount).
```

Эти два способа создания счетчиков и обращения к ним — с использованием перечислений и строк — на самом деле эквивалентны, потому что Hadoop преобразует перечисления в строки для отправки счетчиков средствами RPC. С перечислениями чуть проще работать, они обеспечивают безопасность типов и подходят для большинства заданий. В тех редких случаях, когда счетчики приходится создавать динамически, используйте интерфейс `String`.

Содержательные имена счетчиков

По умолчанию имя счетчика совпадает с полным именем класса Java перечисления. В веб-интерфейсе и на консоли такие имена воспринимаются не лучшим

образом, поэтому Hadoop позволяет изменять отображаемые имена с использованием ресурсных пакетов. Мы воспользовались этой возможностью, так что вместо «Temperature\$MISSING» выводится «Air Temperature Records». Для динамических счетчиков в качестве отображаемых имен используются имена групп и счетчиков, так что с ними обычно проблем не бывает.

Рецепт определения содержательных имен несложен. Создайте файл свойств, имя которого совпадает с именем перечисления, а в качестве разделителя вложенных классов используется символ подчеркивания. Файл свойств должен находиться в одном каталоге с классом верхнего уровня, содержащим перечисление. Так, для счетчиков из листинга 8.1 файл называется *MaxTemperatureWithCounters_Temperature.properties*.

Файл свойств должен содержать одно свойство с именем *CounterGroupName*, значение которого определяет отображаемое имя всей группы. Для каждого поля в перечислении определяется соответствующее свойство, имя которого строится из имени поля и суффикса *.name*, а значение определяет отображаемое имя счетчика. Содержимое файла *MaxTemperatureWithCounters_Temperature.properties* выглядит так:

```
CounterGroupName=Air Temperature Records
MISSING.name=Missing
MALFORMED.name=Malformed
```

Hadoop использует стандартный механизм локализации Java для загрузки свойства того локального контекста, в котором выполняется ваша программа. Например, если вы определите китайскую версию свойств в файле с именем *MaxTemperature-WithCounters_Temperature_zh_CN.properties*, они будут использоваться при выполнении в локальном контексте *zh_CN*. За дополнительной информацией обращайтесь к документации *java.util.PropertyResourceBundle*.

Получение значений счетчиков

Кроме просмотра счетчиков в веб-интерфейсе и командной строке (команда *hadoop job -counter*), также можно получить их значения в Java API. Это можно делать даже во время выполнения задания, хотя чаще счетчики проверяются в конце выполнения задания, когда их значения стабилизируются. В листинге 8.2 приведена программа для вычисления процента записей с отсутствующими полями температуры.

Листинг 8.2. Приложение для вычисления процента записей с отсутствующими полями температуры

```
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.mapred.*;
```

продолжение ↗

Листинг 8.2 (продолжение)

```
import org.apache.hadoop.util.*;

public class MissingTemperatureFields extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 1) {
            JobBuilder.printUsage(this, "<job ID>");
            return -1;
        }
        String jobID = args[0];
        JobClient jobClient = new JobClient(new JobConf(getConf()));
        RunningJob job = jobClient.getJob(JobID.forName(jobID));
        if (job == null) {
            System.err.printf("No job with ID %s found.\n", jobID);
            return -1;
        }
        if (!job.isComplete()) {
            System.err.printf("Job %s is not complete.\n", jobID);
            return -1;
        }

        Counters counters = job.getCounters();
        long missing = counters.getCounter(
            MaxTemperatureWithCounters.Temperature.MISSING);

        long total = counters.getCounter(Task.Counter.MAP_INPUT_RECORDS);

        System.out.printf("Records with missing temperature fields: %.2f%%\n",
            100.0 * missing / total);
        return 0;
    }

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

Сначала мы получаем от `JobClient` объект `RunningJob`, вызывая метод `getJob()` с идентификатором задания. Мы проверяем, существует ли задание с указанным идентификатором. Его может и не быть — например, потому что идентификатор указан неправильно или на трекере заданий нет ссылки на задание (в памяти хранится только информация 100 последних заданий — количество определяется

свойством `mapred.jobtracker.completeuserjobs.maximum`, а при перезапуске трекера вся информация удаляется).

Убедившись в том, что задание завершено, мы вызываем метод `getCounters()` объекта `RunningJob`. Метод возвращает объект `Counters`, содержащий все счетчики задания. Класс `Counters` предоставляет различные методы для получения имен и значений счетчиков. Мы используем метод `getCounter()`, который получает перечисление для определения количества записей с отсутствующим полем температуры и общего количества обработанных записей (из встроенного счетчика).

Наконец, мы выводим процент записей с отсутствующим полем температуры. Для всего набора метеорологических данных результат будет выглядеть так:

```
% hadoop jar hadoop-examples.jar MissingTemperatureFields job_201202040938_0012
Records with missing temperature fields: 5.47%
```

Использование нового MapReduce API

В приведенном примере используется старый API, потому что новый эквивалент получения счетчиков после завершения задания в Hadoop 1.x недоступен. Главное отличие заключается в использовании объекта `Cluster` для получения объекта `Job` (вместо `RunningJob`) с последующим вызовом метода `getCounters()`:

```
Cluster cluster = new Cluster(getConf());
Job job = cluster.getJob(JobID.forName(jobID));
Counters counters = job.getCounters();
long missing = counters.findCounter(
    MaxTemperatureWithCounters.Temperature.MISSING).getValue();
long total = counters.findCounter(TaskCounter.MAP_INPUT_RECORDS).getValue();
```

Вместо перечисления `org.apache.hadoop.mapred.Task.Counter` из старого API в новом API используется перечисление `org.apache.hadoop.mapreduce.TaskCounter`.

Пользовательские счетчики в Streaming

Программа MapReduce на базе Streaming может увеличивать счетчики, отправляя специально отформатированную строку в стандартный поток ошибок, который в данном случае выполняет функции управляющего канала. Стока должна иметь следующий формат:

```
reporter:counter:группа,счетчик,величина
```

В следующем фрагменте на языке Python счетчик «Missing» из группы «Temperature» увеличивается на 1:

```
sys.stderr.write("reporter:counter:Temperature,Missing,1\n")
```

Аналогичным образом задается и сообщение состояния — в этом случае строка имеет формат

```
reporter:status:сообщение
```

Сортировка

Возможность сортировки данных занимает центральное место в MapReduce. Даже если для вашего приложения сортировка как таковая не актуальна, приложение может использовать стадию сортировки, предоставляемую MapReduce, для упорядочения своих данных. В этом разделе мы рассмотрим разные способы сортировки наборов данных и управления порядком сортировки в MapReduce. Сортировка данных Avro рассматривается в разделе «Сортировка с использованием Avro MapReduce», с. 183.

Подготовка

Мы отсортируем набор метеорологических данных по температуре. Хранение температур в объектах `Text` для сортировки не подходит, потому что целые числа со знаком не сортируются в лексикографическом порядке¹. Вместо этого мы будем хранить данные в последовательных файлах, в которых ключи `IntWritable` представляют температуру (и правильно сортируются), а значения `Text` соответствуют строкам данных.

Задание MapReduce в листинге 8.3 использует только задачи отображения и фильтрует входные данные, исключая из них записи, не содержащие действительные данные температуры. Каждое отображение создает один последовательный файл с блочным сжатием как выходные данные. Запуск производится следующей командой:

```
% hadoop jar hadoop-examples.jar SortDataPreprocessor input/ncdc/all \
  input/ncdc/all-seq
```

Листинг 8.3. Программа MapReduce для преобразования метеорологических данных в формат `SequenceFile`

```
public class SortDataPreprocessor extends Configured implements Tool {  
  
    static class CleanerMapper
```

¹ Распространенное обходное решение этой проблемы — особенно в текстовых приложениях `Streaming` — заключается в добавлении смещения для устранения отрицательных чисел и дополнения слева нулями, чтобы все числа имели одинаковое количество символов. Другое решение описано в разделе «`Streaming`», с. 366.

```
extends Mapper<LongWritable, Text, IntWritable, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        parser.parse(value);
        if (parser.isValidTemperature()) {
            context.write(new IntWritable(parser.getAirTemperature()), value);
        }
    }
}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setMapperClass(CleanerMapper.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(Text.class);
    job.setNumReduceTasks(0);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(job, true);
    SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(job,
        CompressionType.BLOCK);

    return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new SortDataPreprocessor(), args);
    System.exit(exitCode);
}
}
```

Частичная сортировка

В разделе «Задание MapReduce по умолчанию» на с. 299 мы видели, что по умолчанию MapReduce сортирует входные записи по ключам. В листинге 8.4

приведена версия программы для сортировки последовательных файлов с ключами `IntWritable`.

Листинг 8.4. Программа MapReduce для сортировки последовательного файла с ключами `IntWritable` и использованием стандартной реализации `HashPartitioner`

```
public class SortByTemperatureUsingHashPartitioner extends Configured
    implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
        if (job == null) {
            return -1;
        }

        job.setInputFormatClass(SequenceFileInputFormat.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        SequenceFileOutputFormat.setCompressOutput(job, true);
        SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
        SequenceFileOutputFormat.setOutputCompressionType(job,
            CompressionType.BLOCK);

        return job.waitForCompletion(true) ? 0 : 1;
    }

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

УПРАВЛЕНИЕ ПОРЯДКОМ СОРТИРОВКИ

Порядок сортировки ключей определяется объектом `RawComparator`, который находится следующим образом:

1. Если свойство `mapred.output.key.comparator.class` задано (явно либо вызовом `setSortComparatorClass()` для объекта `Job`), то используется экземпляр этого класса. (Эквивалентный метод старого API — `setOutputKeyComparatorClass()` для объекта `JobConf`.)

2. В противном случае ключи должны быть субклассом WritableComparable, и используется объект сравнения, зарегистрированный для класса ключа.

3. Если объект сравнения не зарегистрирован, используется реализация RawComparator, которая десериализует сравниваемые потоки байтов в объекты и делегирует выполнение операции методу compareTo() интерфейса WritableComparable.

Эти правила показывают, почему так важно зарегистрировать оптимизированные версии RawComparator для пользовательских реализаций Writable (см. «Реализация RawComparator», с. 155) и как легко изменить порядок сортировки, назначая собственный объект сравнения (см. «Вторичная сортировка», с. 361).

Предположим, программа будет запущена с использованием 30 сверток¹:

```
% hadoop jar hadoop-examples.jar SortByTemperatureUsingHashPartitioner \
-D mapred.reduce.tasks=30 input/ncdc/all-seq output-hashsort
```

Команда создает 30 выходных файлов, каждый из которых отсортирован. Тем не менее не существует простого способа объединения этих файлов (например, посредством конкатенации текстовых файлов) для получения глобально отсортированного файла. Во многих приложениях это не существенно; например, для выполнения поиска по ключу подойдет и частично отсортированный набор файлов.

Приложение: поиск в MapFile

Поиск по ключу достаточно хорошо работает в наборах файлов. Если изменить выходной формат на MapFileOutputFormat, как показано в листинге 8.5, выходные данные будут состоять из 30 объектов MapFile, с которыми можно выполнять операцию поиска².

Листинг 8.5. Программа MapReduce для сортировки SequenceFile и получения выходных данных в формате MapFile

```
public class SortByTemperatureToMapFile extends Configured implements Tool {  
  
    @Override  
    public void run(String[] args) throws Exception {  
        Configuration conf = getConf();  
        Job job = new Job(conf, "SortByTemperatureToMapFile");  
        job.setJarByClass(SortByTemperatureToMapFile.class);  
        job.setMapperClass(SortByTemperatureMapper.class);  
        job.setCombinerClass(SortByTemperatureCombiner.class);  
        job.setReducerClass(SortByTemperatureReducer.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
        job.setMapOutputKeyClass(Text.class);  
        job.setMapOutputValueClass(IntWritable.class);  
        job.setInputFormatClass(SequenceFileInputFormat.class);  
        job.setOutputFormatClass(MapFileOutputFormat.class);  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        job.waitForCompletion(true);  
    }  
}
```

продолжение ↗

¹ В разделе «Сортировка и слияние SequenceFile», с. 193, показано, как сделать то же самое с использованием примера программы сортировки, поставляемого вместе с Hadoop.

² На момент написания книги версия MapFileOutputFormat для нового API была недоступна в Hadoop версий 1.x. Эквивалентный код примеров для старого API доступен на веб-сайте книги.

Листинг 8.5 (продолжение)

```

public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setInputFormatClass(SequenceFileInputFormat.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputFormatClass(MapFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(job, true);
    SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(job,
        CompressionType.BLOCK);
    return job.waitForCompletion(true) ? 0 : 1;
}

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

```

MapFileOutputFormat предоставляет пару вспомогательных статических методов для выполнения поиска в выходных данных MapReduce; пример их использования приведен в листинге 8.6.

Листинг 8.6. Выборка первой записи с заданным ключом из набора MapFile

```

public class LookupRecordByTemperature extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            JobBuilder.printUsage(this, "<path> <key>");
            return -1;
        }
        Path path = new Path(args[0]);
        IntWritable key = new IntWritable(Integer.parseInt(args[1]));

        Reader[] readers = MapFileOutputFormat.getReaders(path, getConf());
        Partitioner<IntWritable, Text> partitioner =
            new HashPartitioner<IntWritable, Text>();

```

```

Text val = new Text();
Writable entry =
    MapFileOutputFormat.getEntry(readers, partitioner, key, val);
if (entry == null) {
    System.err.println("Key not found: " + key);
    return -1;
}
NcdcRecordParser parser = new NcdcRecordParser();
parser.parse(val.toString());
System.out.printf("%s\t%s\n", parser.getStationId(), parser.getYear());
return 0;
}

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

```

Метод `getReaders()` открывает `MapFile.Reader` для каждого из выходных файлов, созданных заданием MapReduce. Метод `getEntry()` использует `Partitioner` для выбора объекта `Reader` ключа и находит значение ключа вызовом метода `get()` объекта `Reader`. Если `getEntry()` возвращает `null`, это означает, что ключ не найден. В противном случае возвращается значение, которое преобразуется в идентификатор станции и год.

Чтобы увидеть, как работает программа, найдем первую запись с температурой -10°C (помните, что температуры хранятся в виде целых чисел, представляющих десятые доли градуса, поэтому запрашивается значение -100):

```
% hadoop jar hadoop-examples.jar LookupRecordByTemperature output-hashmapsort -100
357460-99999      1956
```

Также объекты `Reader` можно использовать напрямую для получения всех записей с заданным ключом. Возвращаемый массив объектов `Reader` упорядочен по разделам, поэтому объект `Reader` для заданного ключа может быть найден с использованием того же объекта `Partitioner`, который использовался в задании MapReduce:

```
Reader reader = readers[partitioner.getPartition(key, val, readers.length)];
```

После получения `Reader` мы получаем первый ключ методом `get()` объекта `MapFile`, после чего многократно вызываем `next()` для получения следующего ключа и значения, пока ключ не изменится. Программная реализация приведена в листинге 8.7.

Листинг 8.7. Получение всех записей с заданным ключом из набора MapFile

```

public class LookupRecordsByTemperature extends Configured implements Tool {
    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            JobBuilder.printUsage(this, "<path> <key>");
            return -1;
        }
        Path path = new Path(args[0]);
        IntWritable key = new IntWritable(Integer.parseInt(args[1]));

        Reader[] readers = MapFileOutputFormat.getReaders(path, getConf());
        Partitioner<IntWritable, Text> partitioner =
            new HashPartitioner<IntWritable, Text>();
        Text val = new Text();

        Reader reader = readers[partitioner.getPartition(key, val, readers.length)];
        Writable entry = reader.get(key, val);
        if (entry == null) {
            System.err.println("Key not found: " + key);
            return -1;
        }
        NcdcRecordParser parser = new NcdcRecordParser();
        IntWritable nextKey = new IntWritable();
        do {
            parser.parse(val.toString());
            System.out.printf("%s\t%s\n", parser.getStationId(), parser.getYear());
        } while(reader.next(nextKey, val) && key.equals(nextKey));
        return 0;
    }

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

```

Пример выполнения программы для получения и подсчета всех записей с температурой -10°C :

```
% hadoop jar hadoop-examples.jar LookupRecordsByTemperature output-hashmapsort
-100 \
2> /dev/null | wc -l
1489272
```

Полная сортировка

Как создать глобально отсортированный файл средствами Hadoop? Наивное решение — использовать один раздел¹. Однако для больших файлов такое решение крайне неэффективно, потому что одной машине придется обрабатывать все выходные данные, так что, по сути, вы лишаетесь всех преимуществ параллельной архитектуры, предоставляемых MapReduce.

Есть и другое решение — создать набор отсортированных файлов, которые в результате конкатенации образуют глобально отсортированный файл. Для этого следует использовать объект `Partitioner`, соблюдающий общий порядок сортировки. Например, при четырех разделах в первый раздел можно поместить ключи температур ниже -10°C , во второй — ключи температур от -10°C до 0°C , в третий — ключи температур от 0°C до 10°C , и в четвертый — свыше 10°C . И хотя этот метод работает, размеры разделов необходимо тщательно выбрать таким образом, чтобы они были примерно одинаковы, а одна свертка не доминировала во времени выполнения задания. Для только что описанной схемы относительные размеры разделов будут такими:

Диапазон температур	$<-10^{\circ}\text{C}$	$[-10^{\circ}\text{C}, 0^{\circ}\text{C})$	$[0^{\circ}\text{C}, 10^{\circ}\text{C})$	$\geq 10^{\circ}\text{C}$
Процент записей	11%	13%	17%	59%

Размеры разделов существенно различаются. А если потребуется создать еще больше разделов, необходимо лучше представлять распределение температур для всего набора данных. Можно без труда написать задание MapReduce для подсчета количества записей, попадающих в разные температурные диапазоны. Например, на рис. 8.1 показано распределение для диапазонов размером 1°C , каждая точка на графике соответствует одному диапазону.

Хотя эту информацию можно использовать для построения равномерного набора разделов, сама необходимость запуска задания, использующего весь набор данных для получения нужной информации, не идеальна. Также можно построить довольно равномерный набор на основании выборки из пространства ключей. Иначе говоря, вы анализируете небольшое подмножество ключей и аппроксимируете распределение, которое используется для определения разделов. К счастью, самостоятельно писать код не придется, так как средства анализа выборки включены в поставку Hadoop.

¹ Так же можно использовать Pig («Сортировка данных», с. 518) или Hive («Сортировка и агрегирование», с. 560) для выполнения сортировки одной командой.

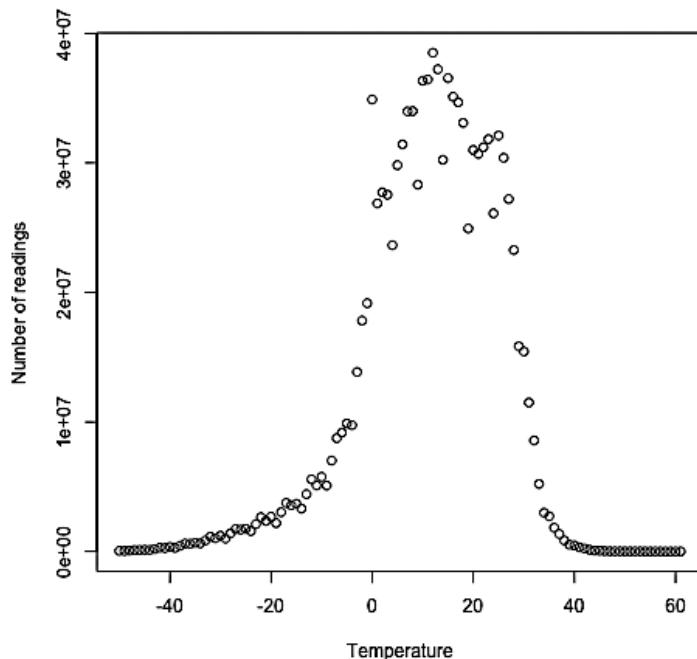


Рис. 8.1. Распределение температур для набора метеорологических данных

Класс `InputSampler` определяет вложенный интерфейс `Sampler`, реализации которого возвращают выборку ключей для заданных объектов `InputFormat` и `Job`:

```
public interface Sampler<K, V> {
    K[] getSample(InputFormat<K, V> inf, Job job)
        throws IOException, InterruptedException;
}
```

Обычно этот интерфейс не вызывается клиентами напрямую. Вместо него используется статический метод `writePartitionFile()` класса `InputSampler`, который создает последовательный файл для хранения ключей, определяющих разделы:

```
public static <K, V> void writePartitionFile(Job job, Sampler<K, V> sampler)
    throws IOException, ClassNotFoundException, InterruptedException
```

Последовательный файл используется `TotalOrderPartitioner` для создания разделов для задания сортировки. Листинг 8.8 показывает, как это делается.

Листинг 8.8. Программа MapReduce для сортировки SequenceFile с ключами IntWritable, использующая TotalOrderPartitioner для глобальной сортировки данных

```
public class SortByTemperatureUsingTotalOrderPartitioner extends Configured
    implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
        if (job == null) {
            return -1;
        }

        job.setInputFormatClass(SequenceFileInputFormat.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        SequenceFileOutputFormat.setCompressOutput(job, true);
        SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
        SequenceFileOutputFormat.setOutputCompressionType(job,
            CompressionType.BLOCK);

        job.setPartitionerClass(TotalOrderPartitioner.class);

        InputSampler.Sampler<IntWritable, Text> sampler =
            new InputSampler.RandomSampler<IntWritable, Text>(0.1, 10000, 10);

        InputSampler.writePartitionFile(job, sampler);

        // Добавление в DistributedCache
        Configuration conf = job.getConfiguration();
        String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
        URI partitionUri = new URI(partitionFile + "#" +
            TotalOrderPartitioner.DEFAULT_PATH);
        DistributedCache.addCacheFile(partitionUri, conf);
        DistributedCache.createSymlink(conf);

        return job.waitForCompletion(true) ? 0 : 1;
    }

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

Мы используем реализацию `RandomSampler`, которая выбирает ключи с постоянной вероятностью (0,1 в нашем примере). Другие параметры определяют максимальное количество проб и максимальное количество сплитов (10 000 и 10 соответственно; эти значения используются по умолчанию, когда `InputSampler` работает как приложение). Выборка прекращается при достижении первого из этих пределов. Так как выборка осуществляется на стороне клиента, важно ограничить количество загружаемых сплитов, чтобы процесс выполнялся быстро. На практике время выборки составляет малую часть общего времени задания.

`InputSampler` записывает файл, который должен быть доступным для задач, выполняемых в кластере; для этого мы добавляем его в распределенный кэш (См. «Распределенный кэш», с. 375.)

В одном из запусков в качестве границ разделов были выбраны значения $-5,6^{\circ}\text{C}$, $13,9^{\circ}\text{C}$ и $22,0^{\circ}\text{C}$ (для четырех разделов). С этими границами размеры разделов получаются более равномерными, чем в предыдущем случае:

Диапазон температур	$< -5,6^{\circ}\text{C}$	$[-5,6^{\circ}\text{C}, 13,9^{\circ}\text{C})$	$[13,9^{\circ}\text{C}, 22,0^{\circ}\text{C})$	$\geq 22,0^{\circ}\text{C}$
Процент записей	29%	24%	23%	24%

Выбор реализации `Sampler` зависит от входных данных. Например, реализация `Splitsampler`, проверяющая только первые n записей в сплите, не так хорошо подходит для отсортированных данных¹, потому что выборка ключей не распределяется по всему сплиту.

Вместе с тем `IntervalSampler` выбирает ключи с постоянными интервалами, и лучше работает с отсортированными данными. `RandomSampler` хорошо подходит для общих данных. Если ни один из вариантов не подходит для вашего приложения (не забудьте, что выборка выполняется для создания разделов, имеющих приблизительно равные размеры), напишите собственную реализацию интерфейса `Sampler`.

`InputSampler` и `TotalOrderPartitioner` хороши еще и возможностью свободного выбора количества разделов. Обычно этот выбор определяется количеством слотов сверток в вашем кластере (выбирайте число чуть меньше максимума, чтобы

¹ В некоторых приложениях часть входных данных отсортирована (или, по крайней мере, частично отсортирована). Например, метеорологические данные в нашем примере упорядочены по времени, что может породить систематическое смещение, поэтому безопаснее использовать `RandomSampler`.

учесть возможные сбои). Однако `TotalOrderPartitioner` работает только при четко определенных границах разделов. Одна из проблем с выбором большого количества разделов — возможность коллизий при небольшом пространстве ключей.

Пример запуска:

```
% hadoop jar hadoop-examples.jar SortByTemperatureUsingTotalOrderPartitioner \
-D mapred.reduce.tasks=30 input/ncdc/all-seq output-totalsort
```

Программа создает 30 выходных разделов, каждый из которых локально отсортирован; кроме того, все ключи раздела i меньше ключей раздела $i+1$.

Вторичная сортировка

Инфраструктура MapReduce сортирует записи по ключу, прежде чем они попадут к задачам свертки. Однако для каждого конкретного ключа данные не сортируются. Порядок следования значений даже не сохраняется между запусками, потому что данные поступают от разных задач отображения, время завершения которых изменяется между запусками. Вообще говоря, программы MapReduce обычно пишутся так, чтобы не зависеть от порядка передачи значений функции свертки. Тем не менее порядок значений может быть установлен посредством сортировки и группировки ключей.

Для примера возьмем программу MapReduce для вычисления максимальной температуры по годам. Если организовать сортировку значений (температур) по убыванию, данные не нужно будет перебирать для определения максимума; вместо этого будет достаточно взять первую запись за каждый год. (Такое решение не слишком эффективно для решения этой конкретной задачи, но оно демонстрирует работу вторичной сортировки).

Для достижения этой цели мы переходим к составным ключам: комбинации из года и температуры. Ключи должны сортироваться сначала по году (по возрастанию), а затем по температуре (по убыванию):

```
1900 35°C
1900 34°C
1900 34°C
...
1901 36°C
1901 35°C
```

Если бы мы ограничились одним изменением ключа, это ничему бы не помогло, потому что все записи одного года имели бы разные ключи и (в общем случае) не попадали бы в одну свертку. Например, записи $(1900, 35^\circ\text{C})$ и $(1900, 34^\circ\text{C})$ достались бы разным сверткам.

Если установить разделение по годам (как компонентам ключа), записи одного года заведомо попадут в одну свертку. Тем не менее и этого недостаточно для достижения нашей цели. Разделение гарантирует лишь то, что одна свертка получит все записи года; она не изменит факта группировки по ключам внутри раздела:

Partition	Group
1900 35°C	
1900 34°C	
1900 34°C	
...	
1901 36°C	
1901 35°C	

Осталось сделать последний шаг — включить управление группировкой. Если значения в свертке будут группироваться по году как части ключа, то все записи одного года будут находиться в одной группе свертки. А поскольку они сортируются по убыванию температур, максимальная температура будет определяться первой записью:

Partition	Group
1900 35°C	
1900 34°C	
1900 34°C	
...	
1901 36°C	
1901 35°C	

Итак, эффект сортировки по значению достигается по следующей схеме:

- В качестве ключа назначается комбинация из естественного ключа и естественного значения.
- Объект сравнения сортировки должен осуществлять упорядочение по составному ключу, то есть по естественному ключу с естественным значением.
- Разделение и сравнение при группировке для составного ключа должно учитывать только естественный ключ.

Код Java

Программная реализация этой схемы приведена в листинге 8.9. Как и прежде, программа использует входные данные в формате простого текста.

Листинг 8.9. Приложение для поиска максимальной температуры с сортировкой температур в ключе

```
public class MaxTemperatureUsingSecondarySort
    extends Configured implements Tool {

    static class MaxTemperatureMapper
        extends Mapper<LongWritable, Text, IntPair, NullWritable> {

        private NcdcRecordParser parser = new NcdcRecordParser();

        @Override
        protected void map(LongWritable key, Text value,
                           Context context) throws IOException, InterruptedException {

            parser.parse(value);
            if (parser.isValidTemperature()) {
                context.write(new IntPair(parser.getYearInt(),
                                          parser.getAirTemperature()), NullWritable.get());
            }
        }
    }

    static class MaxTemperatureReducer
        extends Reducer<IntPair, NullWritable, IntPair, NullWritable> {

        @Override
        protected void reduce(IntPair key, Iterable<NullWritable> values,
                             Context context) throws IOException, InterruptedException {

            context.write(key, NullWritable.get());
        }
    }

    public static class FirstPartitioner
        extends Partitioner<IntPair, NullWritable> {

        @Override
        public int getPartition(IntPair key, NullWritable value, int numPartitions)
        {
            // Умножаем на 127 для равномерности
            return Math.abs(key.getFirst() * 127) % numPartitions;
        }
    }

    public static class KeyComparator extends WritableComparator {
```

продолжение ↗

Листинг 8.9 (продолжение)

```
protected KeyComparator() {
    super(IntPair.class, true);
}
@Override
public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1;
    IntPair ip2 = (IntPair) w2;
    int cmp = IntPair.compare(ip1.getFirst(), ip2.getFirst());
    if (cmp != 0) {
        return cmp;
    }
    return -IntPair.compare(ip1.getSecond(), ip2.getSecond()); //reverse
}
}

public static class GroupComparator extends WritableComparator {
    protected GroupComparator() {
        super(IntPair.class, true);
    }
    @Override
    public int compare(WritableComparable w1, WritableComparable w2) {
        IntPair ip1 = (IntPair) w1;
        IntPair ip2 = (IntPair) w2;
        return IntPair.compare(ip1.getFirst(), ip2.getFirst());
    }
}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setPartitionerClass(FirstPartitioner.class);
    job.setSortComparatorClass(KeyComparator.class);
    job.setGroupingComparatorClass(GroupComparator.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    job.setOutputKeyClass(IntPair.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
}
```

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

В функции отображения мы создаем ключ, представляющий код и температуру, используя реализацию `IntPairWritable`. (`IntPair` — аналог класса `TextPair`, созданного нами в разделе «Пользовательские реализации `Writable`», с. 153.) Передавать какую-либо информацию в значении не нужно, потому что первую (максимальную) температуру можно получить в свертке из ключа, поэтому мы используем `NullWritable`. Свертка выводит первый ключ, который вследствие вторичной сортировки представляет собой объект `IntPair` для года и максимальной температуры. Метод `toString()` класса `IntPair` создает строку, разделенную символом табуляции, поэтому вывод представляет собой набор пар «год-температура», разделенных табуляцией.



Многим приложениям требуется доступ ко всем отсортированным значениям, а не только к первому значению, как в нашем случае. В этом случае поля значений должны заполняться, так как в свертке выбирается только первый ключ. Это приводит к неизбежному дублированию информации между ключом и значением.

Разделение по первому полю ключа (год) включается при помощи пользовательской реализации `FirstPartitioner`. Чтобы ключи сортировались по году (по возрастанию) и температуре (по убыванию), мы используем пользовательский класс сравнения (назначенный вызовом `setSortComparatorClass()`), который извлекает содержимое полей и выполняет необходимые сравнения. Аналогичным образом для группировки ключей по годам используется пользовательский объект сравнения (назначенный вызовом `setGroupingComparatorClass()`), который извлекает первое поле ключа для сравнения¹.

При запуске этой программы выводятся максимальные температуры для каждого года:

```
% hadoop jar hadoop-examples.jar MaxTemperatureUsingSecondarySort input/ncdc/all
\
> output-secondarysort
% hadoop fs -cat output-secondarysort/part-* | sort | head
```

продолжение ↗

¹ Для простоты пользовательские реализации сравнения приведены без оптимизации; о том, как ускорить их работу, рассказано в разделе «Реализация `RawComparator`», с. 155.

```
1901    317
1902    244
1903    289
1904    256
1905    283
1906    294
1907    283
1908    289
1909    278
1910    294
```

Streaming

Для выполнения вторичной сортировки в Streaming можно воспользоваться парой библиотечных классов, предоставляемых Hadoop. Ниже приведена управляющая программа, которая может использоваться для вторичной сортировки:

```
hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-* streaming.jar \
-D stream.num.map.output.key.fields=2 \
-D mapred.text.key.partition.options=-k1,1 \
-D mapred.output.key.comparator.class=\
org.apache.hadoop.mapred.lib.KeyFieldBasedComparator \
-D mapred.text.key.comparator.options="-k1n -k2nr" \
-input input/ncdc/all \
-output output_secondarysort_streaming \
-mapper ch08/src/main/python/secondary_sort_map.py \
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
-reducer ch08/src/main/python/secondary_sort_reduce.py \
-file ch08/src/main/python/secondary_sort_map.py \
-file ch08/src/main/python/secondary_sort_reduce.py
```

Наша функция отображения (листинг 8.10) выдает записи с полями года и температуры. Комбинация этих полей должна интерпретироваться как ключ, поэтому мы задаем `stream.num.map.output.key.fields` значение 2. Это означает, что значения будут пустыми (как и в коде на Java).

Листинг 8.10. Функция отображения для выполнения вторичной сортировки на языке Python

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

import re
import sys

for line in sys.stdin:
```

```
val = line.strip()
(year, temp, q) = (val[15:19], int(val[87:92]), val[92:93])
if temp == 9999:
    sys.stderr.write("reporter:counter:Temperature,Missing,1\n")
elif re.match("[01459]", q):
    print "%s\t%s" % (year, temp)
```

Однако разделы не должны определяться всем ключом, поэтому мы используем объект `KeyFieldBasedPartitioner`, позволяющий выполнять деление по части ключа. Его конфигурация определяется свойством `mapred.text.key.partition.options`. Значение `-k1,1` означает, что использоваться будет только первое поле ключа, а поля будут разделяться строкой, определяемой свойством `map.output.key.field.separator` (символ табуляции по умолчанию).

Затем нужен объект сравнения, который сортирует поле года по возрастанию, а поле температуры по убыванию, чтобы функция свертки могла просто вернуть первую запись в каждой группе. Hadoop предоставляет класс `KeyFieldBasedComparator`, идеально подходящий для этой цели. Порядок сравнения определяется спецификацией, сходной с той, которая используется программой GNU `sort`. Спецификация задается свойством `mapred.text.key.comparator.options`. Значение `-k1n -k2nr`, использованное в нашем примере, означает «сортировать по первому полю в числовом порядке, затем по второму полю в обратном числовом порядке». Класс `KeyFieldBasedComparator`, как и его «родственник» `KeyFieldBasedPartitioner`, для разбиения ключа на поля использует разделитель, заданный свойством `map.output.key.field.separator`.

В Java-версии нам пришлось задавать объект сравнения для группировки; в программах Streaming границы групп никак не обозначены, поэтому функция свертки должна обнаруживать их самостоятельно, отслеживая изменения года (листинг 8.11).

Листинг 8.11. Функция свертки для вторичной сортировки на языке Python

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

import sys

last_group = None
for line in sys.stdin:
    val = line.strip()
    (year, temp) = val.split("\t")
    group = year
    if last_group != group:
        print val
    last_group = group
```

При запуске программы Streaming выводит тот же результат, что и Java-версия.

Наконец, обратите внимание на то, что область применения `KeyFieldBasedPartitioner` и `KeyFieldBasedComparator` не ограничивается программами Streaming; они также могут применяться и в программах MapReduce на Java.

Соединения

MapReduce может выполнять операцию соединения (*join*) с большими наборами данных, но написание кода соединения «с нуля» является достаточно нетривиальной задачей. Вместо написания программ MapReduce стоит рассмотреть возможность использования средств более высокого уровня — например, Pig, Hive или Cascading, в которых операции соединения входят в число основных компонентов реализации.

Итак, какую же задачу мы пытаемся решить? Имеется два набора данных — например, база данных метеорологических станций и метеорологические данные. Требуется соединить эти два набора. Допустим, мы хотим просмотреть историю наблюдений для каждой станции, с подстановкой метаданных станций в каждую строку вывода (рис. 8.2).

Реализация соединения зависит от размера наборов данных и способа их разделения. Если один набор данных большой (метеорологические данные), а другой достаточно мал, чтобы его можно было скопировать в каждый узел кластера (как метаданные станций), соединение может быть выполнено заданием MapReduce, которое собирает вместе записи каждой станции (например, посредством частичной сортировки по идентификатору станции). Задача отображения или свертки использует меньший набор данных для поиска метаданных станции по идентификатору, чтобы эти данные могли выводиться с каждой записью. Этот подход рассматривается в разделе «Распространение побочных данных», с. 374, в котором речь пойдет о механике распределения данных по трекерам задач.

Если соединение выполняется задачей отображения, оно называется *соединением на стороне отображения*; соответственно, соединение, выполняемое задачей свертки, называется *соединением на стороне свертки*.

Если оба набора данных слишком велики для копирования на каждый узел в кластере, их все равно можно соединить средствами MapReduce на стороне отображения или свертки в зависимости от того, как структурированы данные. Типичный пример — пользовательская база данных и журнал некоторой пользовательской активности (например, журнал доступа). Для популярных служб вариант с распространением пользовательской базы данных (или журналов) по всем узлам MapReduce неприемлем.

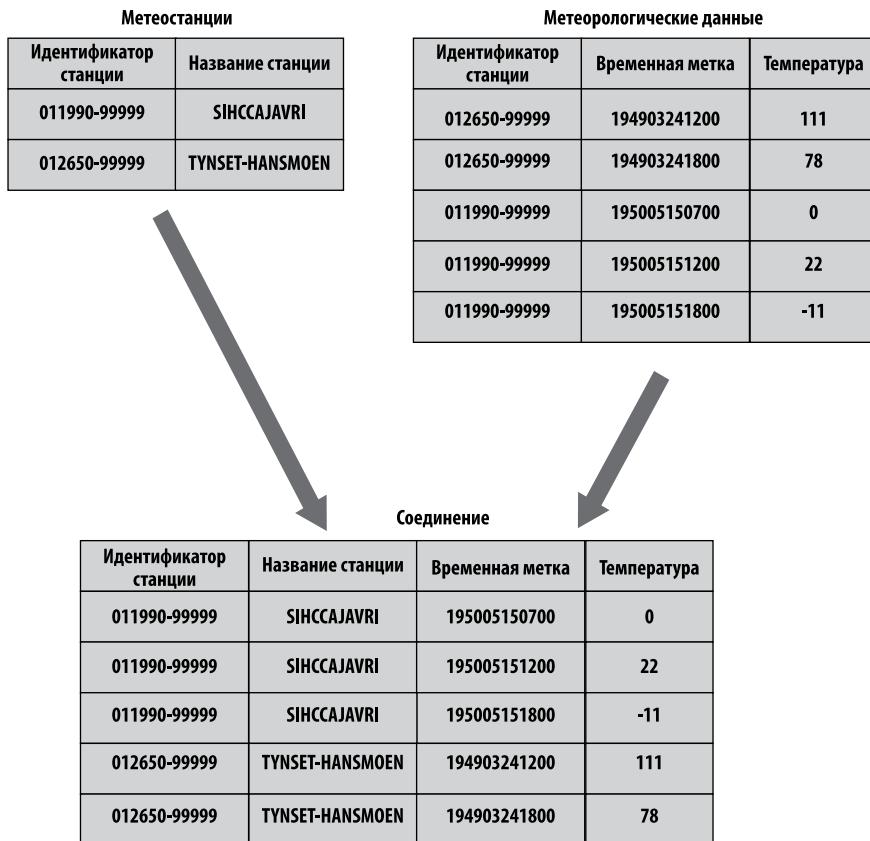


Рис. 8.2. Внутреннее соединение двух баз данных

Соединения на стороне отображения

Работа соединений на стороне отображения для больших наборов данных основана на выполнении операции до того, как данные попадут к функции отображения. Для этого входные данные каждого отображения должны быть разделены и отсортированы определенным образом. Каждый набор данных должен быть разделен на одинаковое количество разделов, и в каждом источнике данных сортировка должна быть выполнена по одному ключу (ключ соединения). Все записи с заданным ключом должны находиться в одном разделе. На первый взгляд такое требование выглядит достаточно жестко (и это действительно так), но оно соответствует описанию вывода задания MapReduce.

Соединение на стороне отображения может использоваться для сведения вывода нескольких заданий с одинаковым количеством сверток, одинаковыми ключами

и выходными файлами, которые не подвергались разбиению (например, из-за того, что они меньше блока HDFS, или из-за применения сжатия *gzip*). Если бы в нашем примере файл метеостанций был частично отсортирован по идентификатору станции, а затем идентичная сортировка была бы проведена с записями — снова по идентификатору станции и с тем же количеством сверток, то эти два вывода соответствовали бы условиям для выполнения соединения на стороне отображения.

Для выполнения соединения на стороне отображения используется класс `CompositeInputFormat` из пакета `org.apache.hadoop.mapreduce.join`. Источники ввода и тип соединения (внутреннее или внешнее) для `CompositeInputFormat` настраиваются через выражение соединения, написанное по простой схеме. Подробности и примеры приведены в документации пакета.

Пример `org.apache.hadoop.examples.Join` — программа общего назначения, работающая в режиме командной строки и предназначенная для выполнения соединений на стороне отображения. Она позволяет запустить задание MapReduce для любой заданной функции отображения и свертки для нескольких соединяемых источников данных.

Соединения на стороне свертки

Соединение на стороне свертки имеет более общий характер, чем соединение на стороне отображения, поскольку оно не требует особого структурирования входных наборов данных. Вместе с тем оно менее эффективно, потому что обоим наборам данных приходится пройти через тасовку MapReduce. Основной принцип заключается в том, что отображение включает в каждую запись данные о ее источнике и использует ключ соединения как выходной ключ отображения, так что записи с одинаковыми ключами сводятся вместе в свертке. Для практической реализации этой схемы необходимы следующие компоненты:

Множественные источники входных данных

Входные источники наборов данных обычно имеют разные форматы, поэтому для разделения логики разбора и пометки для каждого источника очень удобно использовать класс `MultipleInputs` (см. «Множественные источники входных данных», с. 328).

Вторичная сортировка

Как упоминалось ранее, свертка получит записи обоих источников с одинаковым ключом, но порядок следования этих записей не гарантирован. Однако для соединения данных важно, чтобы данные одного источника предшествовали данным другого. Для соединения метеорологических

данных запись метеостанции должна быть первым из значений, видимых для каждого ключа, чтобы функция свертки могла дополнить метеорологические данные названием станции и немедленно выдать их. Конечно, если бы записи буферизировались в памяти, их можно было бы получить в любом порядке, но такой буферизации следует избегать, потому что слишком большое количество записей в любой группе может привести к превышению объема памяти, доступной свертке¹.

В разделе «Вторичная сортировка» на с. 361 было показано, как задать порядок следования значений для каждого ключа в свертке; мы воспользуемся этим приемом.

Для пометки каждой записи мы воспользуемся классом `TextPair` из главы 4 для ключей (для хранения идентификатора станции) и метки. К значениям метки предъявляется единственное требование: они должны сортироваться так, чтобы записи станций предшествовали записям метеорологических данных. Для этого можно снабжать записи станций меткой 0, а записи метеорологических данных — меткой 1. Соответствующие классы отображений приведены в листингах 8.12 и 8.13.

Листинг 8.12. Отображение для пометки записей станций для соединения на стороне свертки

```
public class JoinStationMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
    private NcdcStationMetadataParser parser = new NcdcStationMetadataParser();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        if (parser.parse(value)) {
            context.write(new TextPair(parser.getStationId(), "0"),
                new Text(parser.getStationName()));
        }
    }
}
```

Листинг 8.13. Отображение для пометки записей метеорологических данных для соединения на стороне свертки

```
public class JoinRecordMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
```

продолжение ➔

¹ Пакет `data_join` из каталога `contrib` реализует объединение на стороне свертки посредством буферизации в памяти, и это ограничение распространяется на него.

Листинг 8.13 (продолжение)

```

private NcdcRecordParser parser = new NcdcRecordParser();

@Override
protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    parser.parse(value);
    context.write(new TextPair(parser.getStationId(), "1"), value);
}

}

```

Свертке известно, что первой будет получена запись станции, поэтому она извлекает из значения имя и выводит его как часть каждой выходной записи (листинг 8.14).

Листинг 8.14. Свертка для соединения помеченных записей метеостанций с помеченными записями метеорологических данных

```

public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {

    @Override
    protected void reduce(TextPair key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        Iterator<Text> iter = values.iterator();
        Text stationName = new Text(iter.next());
        while (iter.hasNext()) {
            Text record = iter.next();
            Text outValue = new Text(stationName.toString() + "\t" +
                record.toString());
            context.write(key.getFirst(), outValue);
        }
    }
}

```

Код предполагает, что для каждого идентификатора станции в метеорологических данных существует ровно одна подходящая запись в наборе данных станций. Если это условие не соблюдается, придется обобщить код для включения метки в объекты значений с использованием другого объекта `TextPair`. Тогда метод `reduce()` сможет обнаружить (и обработать) отсутствующие или дублирующиеся записи перед обработкой метеорологических данных.



Так как объекты в итераторе значений свертки используются повторно (по соображениям эффективности), очень важно, чтобы код создавал копию первого объекта `Text` из итератора `values`:

```
Text stationName = new Text(iter.next());
```

Без создания копии ссылка `stationName` при преобразовании в строку будет относиться к только что прочитанному значению, а это приведет к ошибке.

Все компоненты задания собраны воедино в классе управляющей программы, приведенном в листинге 8.15. Принципиально то, что разделение и группировка осуществляются по первой части ключа — идентификатору станции, для чего используется специальная реализация `Partitioner` (`KeyPartitioner`) и специальный класс сравнения группы `FirstComparator` (из `TextPair`).

Листинг 8.15. Приложение для соединения записей метеорологических данных с названиями станций

```
public class JoinRecordWithStationName extends Configured implements Tool {  
  
    public static class KeyPartitioner extends Partitioner<TextPair, Text> {  
        @Override  
        public int getPartition(TextPair key, Text value, int numPartitions) {  
            return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;  
        }  
    }  
  
    @Override  
    public int run(String[] args) throws Exception {  
        if (args.length != 3) {  
            JobBuilder.printUsage(this, "<ncdc input> <station input> <output>");  
            return -1;  
        }  
  
        Job job = new Job(getConf(), "Join weather records with station names");  
        job.setJarByClass(getClass());  
  
        Path ncdcInputPath = new Path(args[0]);  
        Path stationInputPath = new Path(args[1]);  
        Path outputPath = new Path(args[2]);  
  
        MultipleInputs.addInputPath(job, ncdcInputPath,  
            TextInputFormat.class, JoinRecordMapper.class);  
        MultipleInputs.addInputPath(job, stationInputPath,  
            TextOutputFormat.class, JoinRecordReducer.class);  
    }  
}
```

продолжение ↗

Листинг 8.15 (продолжение)

```
    TextInputFormat.class, JoinStationMapper.class);
FileOutputFormat.setOutputPath(job, outputPath);

job.setPartitionerClass(KeyPartitioner.class);
job.setGroupingComparatorClass(TextPair.FirstComparator.class);

job.setMapOutputKeyClass(TextPair.class);

job.setReducerClass(JoinReducer.class);
job.setOutputKeyClass(Text.class);

return job.waitForCompletion(true) ? 0 : 1;
}

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

При выполнении программы для примерного набора данных будет получен результат следующего вида:

011990-99999	SIHCCAJAVA	RI	0067011990999991950051507004+68750...
011990-99999	SIHCCAJAVA	RI	0043011990999991950051512004+68750...
011990-99999	SIHCCAJAVA	RI	0043011990999991950051518004+68750...
012650-99999	TYNSET-HANSMOEN		0043012650999991949032412004+62300...
012650-99999	TYNSET-HANSMOEN		0043012650999991949032418004+62300...

Распространение побочных данных

Побочные данные (side data) можно определить как дополнительные данные, доступные только для чтения, необходимые заданию для обработки основного набора данных. Проблема заключается в том, чтобы удобно и эффективно предоставить доступ к побочным данным всем задачам отображения или свертки (распределенным по кластеру).

Использование конфигурации задания

Вы можете определять произвольные пары «ключ-значение» в конфигурации задания при помощи `set`-методов объекта `Configuration` (или `JobConf` в старом MapReduce API). Этот механизм хорошо подходит для передачи небольших блоков метаданных вашим задачам.

В задаче для получения данных из конфигурации используется метод `getConfiguration()` класса `Context`. (В старом API процедура была более сложной: разработчик переопределял метод `configure()` в реализации `Mapper` или `Reducer`, а для получения данных использовал `get`-метод для переданного объекта `JobConf`. Данные очень часто сохранялись в поле экземпляра, чтобы они могли использоваться в методе `map()` или `reduce()`.)

Обычно для кодирования метаданных хватает примитивного типа, а для объектов произвольной сложности приходится либо выполнять сериализацию самостоятельно (при наличии готового механизма преобразования объектов в строки и обратно), либо использовать класс Hadoop `Stringifier`. Реализация `DefaultStringifier` использует для сериализации объектов инфраструктуру сериализации Hadoop (см. «Сериализация», с. 140).

Этот механизм не должен применяться для передачи более нескольких килобайт данных, потому что он может создать лишние затраты памяти для демонов Hadoop, особенно в системе с сотнями заданий. Конфигурация задания читается трекером заданий, трекером задач и дочерней JVM, причем при каждом чтении в память загружаются все данные, даже если они не используются. Пользовательские свойства не используются трекером заданий и трекером задач, так что их загрузка обрамивается лишь напрасными тратами времени и памяти.

Распределенный кэш

Вместо сериализации побочных данных в конфигурации задания для распространения наборов данных лучше воспользоваться механизмом распределенного кэширования Hadoop. Он предоставляет средства для копирования на узлы заданий файлов и архивов, чтобы они могли использоваться задачами во время выполнения. Для снижения нагрузки на каналы связи файлы обычно копируются на каждый узел всего один раз для каждого задания.

Использование

Для программ, использующих `GenericOptionsParser` (к этой категории относятся многие программы, представленные в книге; см. «`GenericOptionsParser`, `Tool`

и ToolRunner», с. 211), можно задать распределяемые файлы в виде списка URI, разделенного запятыми, в параметре *-files*. Файлы могут храниться в локальной файловой системе, в HDFS или в другой файловой системе, поддерживаемой Hadoop (например, в S3). Если схема не задана, предполагается, что файлы являются локальными (причем даже в тех случаях, когда файловая система по умолчанию не является локальной файловой системой).

Вы также можете копировать файлы архивов (файлы JAR, ZIP, tar и gzip) в свои задачи с параметром *-archives*; на узле задачи архивы распаковываются. Параметр *-libjars* добавляет JAR-файлы в путь к классам задач отображения и свертки. Это удобно, если библиотечные JAR-файлы не упакованы в JAR-файл задания.



Streaming не использует распределенный кэш для копирования своих сценариев в кластере. Копируемый файл задается параметром *-file* (обратите внимание на единственное число), который должен быть задан для каждого копируемого файла. Кроме того, с параметром *-file* могут задаваться только файловые пути, а не URI, поэтому они должны быть доступны в локальной файловой системе клиента, запускающего задание Streaming.

Streaming также поддерживает параметры *-files* и *-archives* для копирования в распределенный кэш тех файлов, которые будут использоваться сценариями Streaming.

Посмотрим, как использовать распределенный кэш для распространения файла метаданных с названиями метеостанций. Выполните следующую команду:

```
% hadoop jar hadoop-examples.jar  
    MaxTemperatureByStationNameUsingDistributedCacheFile \  
    -files input/ncdc/metadata/stations-fixed-width.txt input/ncdc/all output
```

Команда копирует локальный файл *stations-fixed-width.txt* (схема не указана, поэтому путь автоматически интерпретируется как локальный файл) на узлы задач, чтобы мы могли использовать его для поиска названий станций. В листинге 8.16 приведен код приложения для поиска максимальных температур с названиями станций с использованием распределенного кэша.

Листинг 8.16. Приложение для поиска максимальных температур с выводом названий станций из таблицы поиска, переданной в виде файла в распределенном кэше

```
public class MaxTemperatureByStationNameUsingDistributedCacheFile  
    extends Configured implements Tool {  
  
    static class StationTemperatureMapper
```

```
extends Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        parser.parse(value);
        if (parser.isValidTemperature()) {
            context.write(new Text(parser.getStationId()),
                new IntWritable(parser.getAirTemperature()));
        }
    }
}

static class MaxTemperatureReducerWithStationLookup
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    private NcdcStationMetadata metadata;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {
        metadata = new NcdcStationMetadata();
        metadata.initialize(new File("stations-fixed-width.txt"));
    }

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {

        String stationName = metadata.getStationName(key.toString());

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(new Text(stationName), new IntWritable(maxValue));
    }
}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    продолжение ↗
```

Листинг 8.16 (продолжение)

```
if (job == null) {
    return -1;
}

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

job.setMapperClass(StationTemperatureMapper.class);
job.setCombinerClass(MaxTemperatureReducer.class);
job.setReducerClass(MaxTemperatureReducerWithStationLookup.class);

return job.waitForCompletion(true) ? 0 : 1;
}

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

Программа ищет максимальную температуру по метеостанциям, поэтому отображение (`StationTemperatureMapper`) просто выдает пары (идентификатор станции, температура). В комбинирующей функции мы повторно используем код `MaxTemperatureReducer` (из глав 2 и 5) для выбора максимальной температуры в заданной группе выходных данных отображения. Свертка (`MaxTemperatureReducerWithStationLookup`), помимо поиска максимальной температуры, использует файл из кэша для определения названия станции.

Метод `setup()` свертки используется для получения файла из кэша по его исходному имени относительно рабочего каталога задачи.



Распределенный кэш может использоваться для копирования файлов, не помещающихся в памяти. Объекты `MapFile` очень полезны в этом отношении, поскольку их формат хорошо подходит для поиска по ключу на диске (см. «`MapFile`», с. 195). Так как объект `MapFile` представляет собой набор файлов с определенной структурой каталогов, вам стоит перевести их в архивный формат (JAR, ZIP, TAR или TAR, обработанный gzip) и добавить в кэш с параметром `-archives`.

Ниже приведен фрагмент вывода с максимальными температурами для нескольких метеостанций:

PEATS RIDGE WARATAH	372
STRATHALBYN RACECOU	410
SHEOAKS AWS	399
WANGARATTA AERO	409
MOOGARA	334
MACKAY AERO	331

Как работает распределенный кэш

При запуске задания Hadoop копирует файлы, заданные параметрами *-files*, *-archives* и *-libjars*, в распределенную файловую систему (обычно HDFS). Затем перед запуском задачи трекер задач копирует файлы из распределенной файловой системы на локальный диск (кэш), чтобы задача могла обращаться к файлам. С этого момента файлы называются *локализованными*. С точки зрения задачи файлы просто доступны (и при этом неважно, что они были скопированы из HDFS). Кроме того, файлы, заданные параметром *-libjars*, добавляются в путь к классам задачи перед запуском.

Трекер также подсчитывает задачи, использующие каждый файл в кэше. Перед запуском задачи счетчик ссылок файла увеличивается на 1; после завершения задачи счетчик уменьшается на 1. Только когда счетчик ссылок уменьшится до 0, файл становится доступным для удаления, потому что он не используется ни одной задачей. Файлы удаляются, чтобы освободить место для новых файлов, при превышении кэшем определенного размера — по умолчанию 10 Гбайт. Чтобы изменить максимальный размер кэша, следует задать свойство конфигурации *local.cache.size* (задается в байтах).

Хотя эта архитектура не гарантирует, что последующие задачи из того же задания, выполняемые на том же трекере задач, обнаружат файл в кэше, вероятность этого весьма высока, так как задачи из задания обычно планируются для выполнения примерно в одно время. Скорее всего, у других заданий не будет возможности завершить выполнение, что приведет к удалению из кэша файла исходной задачи.

Файлы локализуются в каталоге *\${mapred.local.dir} /taskTracker/archive* на трекерах задач. Впрочем, приложениям это знать не обязательно, потому что символические ссылки на файлы создаются в рабочем каталоге задачи.

API для работы с распределенным кэшем

В большинстве приложений использование API распределенного кэша не обязательно, поскольку приложения могут работать с ним через *GenericOptionsParser*, как показано в листинге 8.16. Однако те приложения, которым необходимы расширенные возможности распределенного кэша, могут использовать API напрямую. API делится на две части: методы помещения данных в кэш (в классе *Job*)

и методы извлечения данных из кэша (в классе `JobContext`)¹. Методы `Job` для помещения данных в кэш:

```
public void addCacheFile(URI uri)
public void addCacheArchive(URI uri)
public void setCacheFiles(URI[] files)
public void setCacheArchives(URI[] archives)
public void addFileToClassPath(Path file)
public void addArchiveToClassPath(Path archive)
public void createSymlink()
```

Вспомните, что в кэш могут помещаться объекты двух типов: файлы и архивы. Файлы на узлах задач остаются в неизменном виде, а архивы распаковываются. Для каждого типа объектов существует три метода: `addCacheXXXX()` для добавления файла или архива в распределенный кэш, `setCacheXXXXs()` для задания целого списка файлов или архивов, добавляемых в кэш за один вызов (с заменой файлов, заданных в предыдущих вызовах), и `addXXXXToClassPath()` для добавления файла или архива в путь к классам задачи MapReduce. В табл. 8.7 эти методы сравниваются с параметрами `GenericOptionsParser` из табл. 5.1.

Таблица 5.1. API распределенного кэша

Метод API Job	Эквивалент GenericOptionsParser	Описание
<code>addCacheFile(URI uri)</code>	<code>-files</code>	Добавление файлов в распределенный кэш для копирования на узлы задач
<code>setCacheFiles(URI[] files)</code>	файл1,файл2,...	
<code>addCacheArchive(URI uri)</code>	<code>-archives</code>	Добавление архивов в распределенный кэш для копирования и распаковки на узлах задач
<code>setCacheArchives(URI[] files)</code>	архив1,архив2,...	
<code>addFileToClassPath(Path file)</code>	<code>-libjars</code> <code>jar1,jar2,...</code>	Добавление в распределенный кэш файлов, которые должны быть включены в путь к классам задачи MapReduce. Так как файлы не распаковываются, это может быть удобным способом включения JAR-файлов в путь к классам

¹ Если вы используете старый MapReduce API, те же методы находятся в пакете `org.apache.hadoop.filecache.DistributedCache`.

Метод API Job	Эквивалент GenericOptionsParser	Описание
addArchiveToClassPath(Path archive)	Нет	Добавление архивов в распределенный кэш для распаковки и включения в путь к классам задач MapReduce. Например, эта возможность удобна для включения в путь к классам каталога, так как вы можете создать архив, содержащий нужные файлы. Альтернативное решение — создание файла JAR и использование addFileToClassPath()



URI, указываемые в методах `add()` и `set()`, должны соответствовать файлам в общей файловой системе, существующим в момент запуска задания. Вместе с тем файлы, задаваемые с параметром `GenericOptionsParser` (например, `-files`), могут относиться к локальным файлам; в этом случае они копируются в общую файловую систему по умолчанию (обычно HDFS).

В этом проявляется принципиальное различие между прямым использованием Java API и использованием `GenericOptionsParser`: в отличие от `GenericOptionsParser`, Java API не копирует файл, заданный в методе `add()` или `set()`, в общую файловую систему.

Последний метод API распределенного кэша — `createSymbolicLink()` — создает символические ссылки на все файлы текущего задания, локализованные на узле задач. Имя символической ссылки задается идентификатором фрагмента в URI файла. Например, для файла, заданного URI `hdfs://namenode/foo/bar#myfile`, в рабочем каталоге задачи создается символическая ссылка `myfile` (пример использования этого API приведен в листинге 8.8). Если идентификатор фрагмента отсутствует, символическая ссылка не создается. Для файлов, добавленных в распределенный кэш с использованием `GenericOptionsParser`, символические ссылки создаются автоматически.



Символические ссылки не создаются для файлов в распределенном кэше при использовании локального исполнителя заданий. Если вы хотите, чтобы ваши задания работали как локально, так и в кластере, используйте методы `getLocalCacheFiles()` и `getLocalCacheArchives()` (см. далее).

Вторая часть API распределенного кэша находится в `JobContext`. Она используется из кода задач отображения или свертки для обращения к файлам из распределенного кэша.

```
public Path[] getLocalCacheFiles() throws IOException;
public Path[] getLocalCacheArchives() throws IOException;
public Path[] getFileClassPaths();
public Path[] getArchiveClassPaths();
```

Если для файлов из распределенного кэша созданы символические ссылки в рабочем каталоге задачи, к локализованному файлу можно обратиться прямо по имени, как было сделано в листинге 8.16. Также можно получить ссылку на файлы и архивы из кэша с использованием методов `getLocalCacheFiles()` и `getLocalCacheArchives()`. Для архивов возвращаемые пути относятся к каталогу, содержащему распакованные файлы. (Для полноты стоит упомянуть о методах `getFileClassPaths()` и `getArchiveClassPaths()` для получения файлов и архивов, включенных в путь к классам задачи.)

Файлы возвращаются в виде локальных объектов `Path`. Чтобы прочитать их, можно воспользоваться локальным экземпляром `FileSystem`, возвращенным методом `getLocal()`. Можно использовать API `java.io.File`, как показано в следующей модификации метода `setup()` класса `MaxTemperatureReducerWithStationLookup`:

```
@Override
protected void setup(Context context)
    throws IOException, InterruptedException {
    metadata = new NcdcStationMetadata();
    Path[] localPaths = context.getLocalCacheFiles();
    if (localPaths.length == 0) {
        throw new FileNotFoundException("Distributed cache file not found.");
    }
    File localFile = new File(localPaths[0].toString());
    metadata.initialize(localFile);
}
```

При использовании старого MapReduce API вместо этого используется статический метод класса `DistributedCache`:

```
@Override
public void configure(JobConf conf) {
    metadata = new NcdcStationMetadata();
    try {
        Path[] localPaths = DistributedCache.getLocalCacheFiles(conf);
        if (localPaths.length == 0) {
            throw new FileNotFoundException("Distributed cache file not found.");
    }
}
```

```

        File localFile = new File(localPaths[0].toString());
        metadata.initialize(localFile);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
}

```

Библиотечные классы MapReduce

В комплект поставки Hadoop входит библиотека классов отображения и свертки для часто используемых функций. Эти классы с краткими описаниями перечислены в табл. 8.8. За дополнительной информацией об их использовании обращайтесь к документации Java.

Таблица 8.8. Библиотечные классы MapReduce

Классы	Описание
ChainMapper, ChainReducer	Выполняет цепочку отображений в одном отображении, и свертку, за которой следует цепочка отображений, в одной свертке. (В символическом представлении $M+RM^*$, где M — отображение, а R — свертка.) Это может существенно сократить объем необходимого дискового ввода/вывода по сравнению с выполнением нескольких заданий MapReduce
FieldSelectionMapReduce (старый API) FieldSelectionMapper и FieldSelectionReducer (новый API)	Отображение и свертка, которые могут выбирать поля (по аналогии с командой UNIX cut) из входных ключей и значений и выдавать их как выходные ключи и значения
IntSumReducer, LongSumReducer	Свертки, суммирующие целые значения для вычисления суммы по каждому ключу.
InverseMapper	Отображение, меняющее местами ключи и значения
MultithreadedMapRunner (старый API) MultithreadedMapper (новый API)	Отображение (или исполнитель в старом API), параллельно выполняющее отображения в разных программных потоках
TokenCounterMapper	Отображение, разбивающее входное значение на слова (с использованием класса Java StringTokenizer) и выводящее каждое слово с количеством вхождений
RegexMapper	Отображение, находящее во входных данных совпадения регулярного выражения и выводящее совпадения с количеством вхождений

9

Создание кластера Hadoop

В этой главе объясняется, как организовать выполнение Hadoop в кластере. Выполнение HDFS и MapReduce на одном компьютере хорошо подходит для изучения этих систем, но практическую пользу они приносят лишь при выполнении на нескольких узлах.

Существуют разные способы организации кластеров Hadoop, от построения собственного кластера до выполнения на арендованном оборудовании или использовании технологий, предоставляющих Hadoop в виде облачного сервиса. Эта и следующая главы дадут вам достаточно информации для создания собственного кластера и управления им, но даже если все рутинное сопровождение выполняется за вас сервисом Hadoop, эти главы содержат немало полезных сведений о принципах работы системы.

Оборудование кластера

Технология Hadoop предназначена для выполнения на среднестатистическом оборудовании. Таким образом, пользователь не привязывается к дорогостоящим фирменным предложениям от одного поставщика; для построения кластера можно выбирать стандартизированное, общедоступное оборудование, предлагаемое многими поставщиками.

«Среднестатистическое» не означает «низкопроизводительное». Низкопроизводительные машины часто содержат дешевые компоненты с более высокой

вероятностью сбоев, чем более дорогие машины (при этом все равно относящиеся к классу среднестатистических). Когда под вашим управлением находятся десятки, сотни и тысячи машин, экономия на дешевых компонентах оказывается ложной, потому что более высокая вероятность сбоев повышает затраты на сопровождение. С другой стороны, использовать мощные машины, рассчитанные для работы с базами данных, тоже не рекомендуется из-за невысокого соотношения «цена/качество». И хотя для построения кластера такой же производительности потребуется меньше машин, чем для кластера из машин среднего уровня, отказ одного высокопроизводительного компьютера сильнее повлияет на работу кластера, потому что большая часть ресурсов кластера станет недоступной.

Аппаратные спецификации быстро устаревают, но для наглядности в 2010 году типичная конфигурация машины для узла данных и трекера задач Hadoop выглядит так:

Процессор

Два четырехъядерных процессора с частотой 2–2,5 ГГц.

Память

16–24 Гбайт, коррекция ошибок¹.

Дисковое пространство

Четыре диска SATA емкостью 1 Тбайт.

Сеть

Гигабитный Ethernet.

В вашем кластере спецификация оборудования наверняка будет другой, но, так как технология Hadoop проектировалась в расчете на поддержку многих ядер и дисков, она сможет в полной мере использовать более мощное оборудование.

Основная реализация Hadoop написана на Java, а следовательно, может работать на любой платформе с JVM, однако в ней найдется немало частей, ориентированных на Unix (например, управляющих сценариев), которые слишком сильно затрудняют реальную эксплуатацию на других plataформах. В общем случае операционные системы Windows не относятся к числу поддерживаемых платформ (хотя и могут использоваться с Cygwin как платформой разработки).

Насколько большим должен быть кластер? На этот вопрос нет точного ответа, но технология Hadoop тем и хороша, что вы можете начать с малого кластера (скажем,

¹ Желательно использовать память с коррекцией ошибок, так как некоторые пользователи Hadoop сообщали о чрезмерном количестве ошибок контрольных сумм при использовании в кластерах Hadoop памяти без коррекции ошибок.

из 10 узлов) и увеличивать его по мере роста доступного пространства и потребностей в вычислительных ресурсах. Во многих отношениях важнее другой вопрос: с какой скоростью должен расти кластер? Хорошей метрикой скорости роста является емкость пространства хранения данных.

ПОЧЕМУ НЕ ИСПОЛЬЗОВАТЬ RAID?

Массивы RAID на узлах данных в кластерах Hadoop никакой пользы не принесут (хотя RAID рекомендуется использовать для дисков на узлах имен, чтобы предотвратить возможное повреждение метаданных). Избыточность, предоставляемая RAID, не нужна, так как HDFS обеспечивает ее посредством репликации данных между узлами.

Более того, распределение данных RAID (RAID 0), часто используемое для повышения производительности дисковых операций, по скорости уступает технологии JBOD (Just a Bunch Of Disks) с циклическим распределением блоков HDFS между всеми дисками. Это объясняется тем, что операции чтения и записи RAID 0 ограничены скоростью самого медленного диска массива RAID. В JBOD дисковые операции независимы, поэтому средняя скорость операций выше, чем у самого медленного диска. На практике быстродействие дисков часто заметно различается даже для дисков одной модели. В тестах, проведенных в кластере Yahoo! (<http://markmail.org/message/xmzc45zi25htr7ry>), технология JBOD работала на 10% быстрее RAID 0 в одном teste (Gridmix) и на 30% быстрее в другом teste (пропускная способность записи HDFS).

Наконец, при сбое диска в конфигурации JBOD система HDFS продолжит работать без сбойного диска, тогда как в RAID сбой даже одного диска приводит к недоступности всего массива (а следовательно, и узла).

Предположим, если объем ваших данных увеличивается на 1 Тбайт в неделю и у вас организована трехсторонняя репликация HDFS, каждую неделю вам потребуется 3 Тбайт дискового пространства. Выделите место для промежуточных файлов и журналов (около 30%, допустим) — в среднем получается около одной машины в неделю (уровня 2010 г.). На практике вряд ли вы будете покупать каждую неделю новую машину и добавлять ее в кластер. Такие вычисления «на скорую руку» полезны именно тем, что они дают представление о размерах кластера. В нашем примере кластеру, в котором хранятся данные за два года, потребуется около 100 машин.

В малых кластерах (около 10 узлов) узел имен и трекер заданий обычно могут работать на одной управляющей машине (при условии, что хотя бы одна копия метаданных узла имен хранится в удаленной файловой системе). С ростом кластера и количества файлов, хранящихся в HDFS, узлу имен требуется больше памяти, и узлы имен и трекер заданий должны быть разнесены на разные машины.

Вторичный узел имен может работать на той же машине, что и основной, но также по соображениям экономии памяти (вторичный узел предъявляет те же требования к памяти) лучше выполнять его на отдельном оборудовании, особенно в больших кластерах. (Эта тема более подробно рассматривается в разделе «Сценарии основных узлов», с. 395.) Машины, на которых работают узлы имен, обычно используют 64-разрядное оборудование, чтобы избежать 3-гигабайтного ограничения размера кучи Java в 32-разрядных архитектурах¹.

Сетевая топология

Типичная архитектура кластера Hadoop состоит из двухуровневой сетевой топологии, изображенной на рис. 9.1. Обычно сегмент содержит 30–40 серверов, оснащается гигабайтным коммутатором и восходящим каналом к центральному коммутатору или маршрутизатору (обычно 1 Гбайт и выше). Принципиально то, что суммарная пропускная способность каналов между узлами одного сегмента намного больше пропускной способности между узлами разных сегментов.

Топология сегментов

Чтобы добиться от Hadoop максимальной производительности, важно настроить систему так, чтобы она знала топологию вашей сети. Если кластер работает на одном сегменте, делать ничего не нужно — эта конфигурация используется по умолчанию. Однако в многосегментных кластерах необходимо определить соответствия между узлами и сегментами, чтобы при распределении задач MapReduce по узлам система Hadoop отдавала предпочтение внутрисегментным передачам (там, где доступна большая пропускная способность) перед внесегментными. HDFS сможет размещать реплики более эффективно для оптимизации производительности и гибкости.

¹ Традиционно считалось, что другие машины кластера (трекер заданий, узлы данных/трекеры задач) должны быть 32-разрядными, чтобы избежать лишних затрат памяти из-за увеличения разрядности указателей. В Java 6 (обновление 14) поддерживаются «сжатые указатели на объекты», с которыми большая часть этих затрат пропадает, так что теперь выполнение на 64-разрядном оборудовании обходится без проблем.

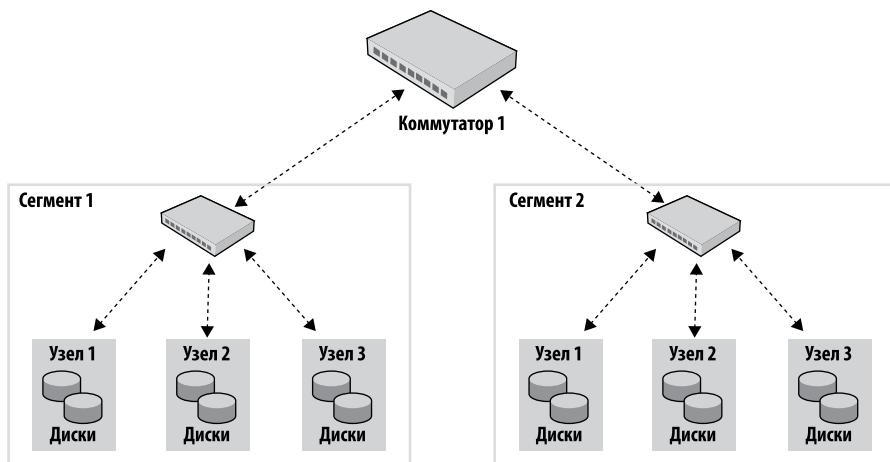


Рис. 9.1. Типичная двухуровневая сетевая архитектура кластера Hadoop

Сетевые локации (такие, как узлы и сегменты) представляются в виде дерева, отражающего сетевые «расстояния» между локациями. Узел имен использует сетевую информацию при определении места размещения реплик блоков (см. «Сетевая топология и Hadoop», с. 111); планировщик MapReduce использует топологическую информацию для поиска ближайшей реплики, содержащей входные данные задачи отображения.

Для сети на рис. 9.1 топология сегментов описывается двумя сетевыми адресами — допустим, `/switch1/rack1` и `/switch1/rack2`. Так как кластер содержит только один коммутатор верхнего уровня, адреса можно упростить до `/rack1` и `/rack2`.

Конфигурация Hadoop должна задать соответствие между адресами узлов и сетевыми локациями. Это соответствие описывается интерфейсом Java `DNSToSwitchMapping`, имеющим следующую сигнатуру:

```
public interface DNSToSwitchMapping {
    public List<String> resolve(List<String> names);
}
```

В параметре `names` передается список IP-адресов, а возвращаемое значение представляет собой список соответствующих строк сетевых адресов. Свойство конфигурации `topology.node.switch.mapping.impl` определяет реализацию интерфейса `DNSToSwitchMapping`, используемую узлом имен и трекером заданий для разрешения сетевых адресов рабочих узлов. В нашем примере `node1`, `node2` и `node3` связываются с `/rack1`, а `node4`, `node5` и `node6` — с `/rack2`.

В большинстве установок Hadoop заниматься самостоятельной реализацией интерфейса не требуется, так как по умолчанию используется реализация

`ScriptBasedMapping`, запускающая пользовательский сценарий для определения привязки. Местонахождение сценария определяется свойством `topology.script.file.name`. Сценарий получает переменное число аргументов с именами или IP-адресами хостов и направляет в стандартный вывод соответствующие сетевые адреса, разделенные пробелами. Пример приведен в вики Hadoop по адресу http://wiki.apache.org/hadoop/topology_rack_awareness_scripts.

Если сетевой адрес не задан, по умолчанию все узлы отображаются на один сетевой адрес с именем `/default-rack`.

Настройка и установка кластера

Итак, заказанное оборудование прибыло. Следующий шаг — монтаж устройств и установка программного обеспечения, необходимого для работы Hadoop.

Существуют разные способы установки и настройки Hadoop. В этой главе приведено описание процесса установки «с нуля» для дистрибутива Apache Hadoop, и общие сведения о том, что необходимо учитывать при создании кластера Hadoop. Также для управления установкой Hadoop можно использовать RPM или пакеты Debian; в этом случае стоит обратить внимание на дистрибутив Cloudera.

Для упрощения установки и сопровождения программ на каждом узле часто применяются средства автоматизации установки — такие, как KickStart (Red Hat Linux) или режим Fully Automatic Installation (Debian). Они позволяют автоматизировать установку операционной системы; для этого сохраняются ответы на вопросы, задаваемые в процессе установки (например, структура разделов диска), а также список устанавливаемых пакетов. Кроме того, поддерживается возможность выполнения пользовательских сценариев в конце процесса, очень важная для внесения окончательных настроек и изменений, не входящих в стандартный процесс установки.

Ниже описаны основные действия, необходимые для запуска Hadoop. Все они должны быть включены в сценарий установки.

Установка Java

Для работы Hadoop необходима установка Java 6 или более поздней версии. Рекомендуется использовать последнюю стабильную версию Sun JDK, хотя дистрибутивы Java от других фирм-поставщиков тоже могут подойти. Следующая команда проверяет наличие правильной установки Java:

```
% java -version
java version "1.6.0_12"
Java(TM) SE Runtime Environment (build 1.6.0_12-b04)
Java HotSpot(TM) 64-Bit Server VM (build 11.2-b01, mixed mode)
```

Создание пользователя Hadoop

Обычно на машине создается специальная учетная запись пользователя Hadoop, чтобы отделить установку Hadoop от других служб, работающих на той же машине.

В малых кластерах некоторые администраторы назначают домашним каталогом пользователя смонтированный диск NFS, чтобы упростить распределение ключей SSH (см. ниже). Сервер NFS обычно находится за пределами кластера Hadoop. Если вы используете NFS, *autofs* позволит смонтировать файловую систему NFS при необходимости тогда, когда системе понадобится к ней обратиться. *Autofs* предоставляет частичную защиту от сбоя сервера NFS и позволяет использовать реплицированные файловые системы для преодоления сбоев. Также необходимо учитывать ряд других аспектов NFS — например, синхронизацию UID и GID. За информацией об использовании в Linux обращайтесь по адресу <http://nfs.sourceforge.net/nfs-howto/index.html>.

Установка Hadoop

Загрузите Hadoop на странице Apache Hadoop (<http://hadoop.apache.org/core/releases.html>), распакуйте содержимое дистрибутива в удобном месте — например, в */usr/local* (другой стандартный вариант — */opt*). Hadoop не устанавливается в домашнем каталоге пользователя, так как это может быть смонтированный каталог NFS:

```
% cd /usr/local
% sudo tar xzf hadoop-x.y.z.tar.gz
```

Также необходимо сменить владельцев файлов Hadoop:

```
% sudo chown -R hadoop:hadoop hadoop-x.y.z
```



Некоторые администраторы устанавливают HDFS и MapReduce в разных местах одной системы. На момент написания книги только HDFS и MapReduce из одного выпуска Hadoop были совместимы, но в будущих версиях требования к совместимости будут смягчены. Когда это произойдет, создание независимых установок станет оправданным, потому что оно предоставит большую свободу действий при обновлении (см. «Обновления», с. 463). Например, удобно иметь возможность обновления MapReduce (например, для устранения ошибки) без нарушения работоспособности HDFS.

Раздельные установки HDFS и MapReduce могут совместно использовать конфигурацию конфигурации; для этого в параметре `-config` (при запуске демонов) передается ссылка на общий каталог конфигурации. Журналы также могут вестись в одном каталоге, так как имена создаваемых журнальных файлов выбираются таким образом, чтобы избежать конфликтов имен.

Тестирование установки

Когда установочный сценарий будет создан, протестируйте его, выполнив установку на машинах своего кластера. Вероятно, для обнаружения всех недочетов в процессе установки вам потребуется несколько итераций. Когда установка заработает, переходите к настройке конфигурации Hadoop и проведите тестовый запуск. Этот процесс описан в следующих разделах.

Конфигурация SSH

Управляющие сценарии (но не демоны) Hadoop используют SSH для выполнения операций уровня кластера. Например, существует сценарий для остановки и запуска всех демонов в кластере. Учтите, что без управляющих сценариев можно обойтись — операции уровня кластера также могут выполняться и другими средствами (например, через Distributed Shell).

Для беспроблемной работы SSH необходимо настроить на прием входа без пароля для пользователя `hadoop` с машин кластера. Проще всего для этого сгенерировать пару открытого/закрытого ключа и разместить их в локации NFS, общедоступной в пределах кластера.

Начните с генерирования пары ключей RSA — для этого введите в учетной записи пользователя `hadoop` следующую команду:

```
% ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

Хотя нам нужен вход без пароля, ключи без паролей создавать не рекомендуется (хотя для локального псевдораспределенного кластера пустой пароль допустим). Введите пароль, когда вам будет предложено. Мы используем *ssh-agent*, чтобы избежать ввода пароля для каждого подключения.

Закрытый ключ хранится в файле, заданном параметром *-f* (*~/.ssh/id_rsa*), а открытый ключ — в файле с тем же именем, но с суффиксом *.pub* (*~/.ssh/id_rsa.pub*).

Затем необходимо проследить за тем, чтобы открытый ключ хранился в файле *~/.ssh/authorized_keys* на всех компьютерах кластера, к которым мы собираемся подключаться. Если домашний каталог пользователя *hadoop* находится в файловой системе NFS, как описано ранее, совместное использование ключей в кластере обеспечивается следующей командой:

```
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Если домашний каталог не открыт для общего доступа средствами NFS, открытые ключи придется распространять другими средствами (например, командой *ssh-copy-id*).

Проверьте возможность подключения с главной машины на рабочую через SSH; убедитесь в том, что *ssh-agent* работает¹, а затем выполните *ssh-add* для сохранения пароля. После этого вы сможете подключаться к рабочей машине без повторного ввода пароля.

Конфигурация Hadoop

Для управления конфигурацией установки Hadoop используются различные файлы, самые важные из которых перечислены в табл. 9.1. В этом разделе рассматривается конфигурация MapReduce 1 с демонами трекеров заданий и задач. Настройка конфигурации MapReduce 2 выполняется совершенно иначе и рассматривается в разделе «Конфигурация YARN» на с. 412.

Таблица 9.1. Конфигурационные файлы Hadoop

Имя файла	Формат	Описание
<code>hadoop-env.sh</code>	Сценарий Bash	Переменные окружения, используемые сценариями выполнения Hadoop

¹ Инструкции по запуску *ssh-agent* приведены в документации.

Имя файла	Формат	Описание
core-site.xml	Конфигурация Hadoop в формате XML	Параметры конфигурации Hadoop Core — например, параметры ввода/вывода, общие для HDFS и MapReduce
hdfs-site.xml	Конфигурация Hadoop в формате XML	Параметры конфигурации демонов HDFS; узел имен, вторичный узел имен и узлы данных
mapred-site.xml	Конфигурация Hadoop в формате XML	Параметры конфигурации демонов MapReduce; трекер заданий и трекеры задач
masters	Простой текст	Список машин (по одной в строке), на каждой из которых работает вторичный узел имен
slaves	Простой текст	Список машин (по одной в строке), на каждой из которых работает узел данных и трекер задач
hadoop-metrics.properties	Файл свойств Java	Свойства, управляющие публикацией метрик в Hadoop (см. «Метрики», с. 451)
log4j.properties	Файл свойств Java	Свойства системных журналов, журнала аудита узла имен и журнала задач для дочернего процесса трекера задач (см. «Журналы Hadoop», с. 240)

Все эти файлы находятся в каталоге *conf* дистрибутива Hadoop. Каталог конфигурации может быть перемещен в другую часть файловой системы (за пределы установки Hadoop, что заметно упрощает обновления) — при условии, что демоны запускаются с параметром *-config*, задающим местонахождение этого каталога в локальной файловой системе.

Управление конфигурацией

В Hadoop нет единой, глобальной точки хранения конфигурационной информации. Вместо этого на каждом узле Hadoop в кластере хранится собственный набор конфигурационных файлов; администратор должен обеспечить их синхронизацию в системе. Hadoop предоставляет простейшие средства синхронизации конфигурации с использованием *rsync* (см. далее); также существуют другие инструменты, упрощающие выполнение этой операции — такие, как *dsh* или *pdsh*.

Система Hadoop спроектирована так, чтобы для всех главных и рабочих машин мог использоваться один набор конфигурационных файлов. Огромным преимуществом такого подхода является его простота — как концептуальная (так как приходится иметь дело только с одной конфигурацией), так и оперативная (сценарии Hadoop достаточно для управления одной конфигурацией).

В некоторых кластерах модель единой конфигурации не работает. Например, если кластер расширяется новыми машинами, отличающимся от существующих по составу оборудования, для новых машин придется определить другую конфигурацию, использующую их дополнительные ресурсы.

В таких ситуациях необходимо определить концепцию класса машины и поддерживать отдельную конфигурацию для каждого класса. Hadoop не предоставляет средств для решения этой задачи, но существует немало отличных программ для управления конфигурацией такого типа — например, Chef, Puppet, cfengine и bcfg2.

В кластерах любого размера синхронизация всех машин создает проблемы: что произойдет, если машина недоступна в момент отправки обновления? Кто гарантирует, что она получит обновление, когда станет доступной? Это большая проблема, которая может породить расхождения в конфигурации установки. И даже если вы используете управляющие сценарии для управления Hadoop, для управления кластером стоит использовать программы управления конфигурацией. Они также хорошо подходят для повседневного сопровождения — например, для устранения дефектов безопасности и обновления системных пакетов.

Управляющие сценарии

Hadoop поставляется со сценариями для выполнения команд, запуска и остановки демонов во всем кластере. Чтобы использовать эти сценарии (находящиеся в каталоге *bin*), необходимо передать Hadoop информацию о том, какие машины входят в кластер. Информация хранится в двух файлах: *masters* и *slaves*, каждый из которых содержит имена хостов или IP-адреса, по одному в строке. Имя файла *masters* выбрано неудачно, поскольку его содержимое определяет, на каких машинах (или машине) должен работать вторичный узел имен. В файле *slaves* перечислены машины, на которых должны работать узлы данных и трекеры задач. Оба файла, *masters* и *slaves*, обычно находятся в каталоге конфигурации, хотя файл *slaves* может находиться где угодно (и под другим именем) — для этого достаточно изменить значение переменной *HADOOP_SLAVES* в *hadoop-env.sh*. Кроме того, эти файлы не нужно распространять по рабочим узлам, потому что они используются только управляющими сценариями, выполняемыми на узле имен или трекере заданий.

В файле *masters* не нужно указывать, на каких машинах работает узел имен или трекер заданий, потому что это зависит от машины, на которой выполняются сценарии. (Более того, при их включении в файл *masters* на этих машинах будет

запущен вторичный узел имен, а это не всегда то, что требуется.) Например, сценарий *start-dfs.sh*, запускающий всех демонов в кластере, запускает узел имен на машине, на которой он выполняется. А если говорить конкретно, он

- 1) запускает узел имен на локальной машине (той, на которой выполняется сценарий);
- 2) запускает узел данных на каждой машине, указанной в файле *slaves*;
- 3) запускает вторичный узел имен на каждой машине, указанной в файле *masters*.

Похожий сценарий *start-mapred.sh* запускает всех демонов MapReduce в кластере. А если говорить конкретно, он:

- 1) запускает трекер заданий на локальной машине;
- 2) запускает трекер задач на каждой машине, указанной в файле *slaves*.

Файл *masters* не используется управляемыми сценариями MapReduce.

Сценарии *stop-dfs.sh* и *stop-mapred.sh* останавливают демонов, запущенных соответствующими сценариями запуска.

Для запуска и остановки демонов Hadoop используется сценарий *hadoop-daemon.sh*. При использовании упомянутых сценариев не следует вызывать *hadoop-daemon.sh* напрямую. Но если вам потребуется управлять демонами Hadoop из другой системы или ваших собственных сценариев, то сценарий *hadoop-daemon.sh* станет хорошей точкой интеграции. Аналогичным образом сценарий *hadoop-daemons.sh* (с буквой «*s*») удобен для запуска одного демона в группе хостов.

Сценарии основных узлов

В зависимости от размера кластера существуют разные конфигурации запуска основных демонов: узла имен, вторичного узла имен и трекера заданий. В малых кластерах (из нескольких десятков узлов) удобно разместить их на одном компьютере; тем не менее с увеличением кластера появляются веские причины для их разделения.

Узел имен требует значительных затрат памяти, так как он хранит в ней метаданные файлов и блоков для всего пространства имен. Вторичный узел имен, хотя и про-стаивает большую часть времени, требует сходных затрат памяти при создании контрольной точки (см. «Образ файловой системы и журнал изменений», с. 436). Для файловых систем с большим количеством файлов на одной машине может не хватить физической памяти для работы как основного, так и вторичного узла имен.

Вторичный узел имен хранит копию последней контрольной точки метаданных файловой системы, которую он создает. Хранение этой (устаревшей) копии на

узле, отличном от узла имен, позволяет восстановить работоспособность в случае потери (или повреждения) всех файлов метаданных узла имен (эта тема более подробно рассматривается в главе 10).

В занятом кластере с множеством заданий MapReduce работа трекера задач приводит к существенным затратам памяти и ресурсов процессора, поэтому трекер следует разместить на отдельном узле.

Независимо от того, работают основные демоны на одном или нескольких узлах, необходимо сделать следующее:

- запустите управляющие сценарии HDFS на машине узла имен. Файл *masters* должен содержать адрес вторичного узла имен;
- запустите управляющие сценарии MapReduce на машине трекера задач.

Когда узел имен и трекер задач работают на разных узлах, их файлы *slaves* должны быть синхронизированы, потому что на каждом узле кластера должен работать узел данных и трекер задач.

Настройки окружения

В этом разделе рассматривается настройка переменных в сценарии *hadoop-env.sh*.

Память

По умолчанию Hadoop выделяет 1000 Мбайт (1 Гбайт) памяти каждому запускаемому демону. Это значение задается переменной `HADOOP_HEAPSIZE` из файла *hadoop-env.sh*. Кроме того, трекер задач запускает отдельные дочерние JVM для выполнения задач отображения и свертки; они тоже должны учитываться при подсчете суммарных затрат памяти на рабочей машине.

Максимальное количество задач отображения, выполняемых на трекере задач одновременно, определяется свойством `mapred.tasktracker.map.tasks.maximum`; по умолчанию выполняются две задачи. Соответствующее свойство для задач свертки, `mapred.tasktracker.reduce.tasks.maximum`, по умолчанию также разрешает выполнение двух сверток. Говорят, что на трекере задач имеется два слота отображения и два слота свертки.

Объем памяти, выделяемой каждой дочерней JVM, изменяется заданием свойства `mapred.child.java.opts`. Значение по умолчанию равно `-Xmx200m`, то есть каждой задаче предоставляются 200 Мбайт памяти. (Кстати говоря, здесь также можно задавать другие параметры JVM — например, включить подробное протоколирование уборки мусора в целях отладки.) Таким образом, конфигурация по умолчанию использует 2800 Мбайт памяти на рабочей машине (см. табл. 9.2).

Таблица 9.2. Затраты памяти на рабочем узле

JVM	Использование памяти по умолчанию	Память, используемая для 8 процессоров (400 Мбайт на дочерний процесс)
Узел данных	1000	1000
Трекер задач	1000	1000
Дочерняя задача отображения трекера задач	2200	7400
Дочерняя задача свертки трекера задач	2200	7400
Итого	2800	7600

Количество задач, одновременно выполняемых на трекере задач, зависит от количества процессоров на машине. Так как задания MapReduce обычно ограничены по скорости ввода/вывода, для повышения эффективности количество задач должно превышать количество процессоров. Превышение зависит от того, какую нагрузку выполняемые задания создают для процессора, но, как правило, на один процессор должно приходиться от 1 до 2 задач (как отображения, так и свертки).

Например, если машина оснащена восемью процессорами и вы хотите выполнять на каждом процессоре два процесса, задайте свойствам `mapred.tasktracker.map.tasks.maximum` и `mapred.tasktracker.reduce.tasks.maximum` значение 7 (не 8, потому что узел данных и трекер задач займут по одному слоту). Если вы также увеличите память, доступную для каждого дочернего процесса до 400 Мбайт, суммарные затраты памяти составят 7600 Мбайт (см. табл. 9.2).

Поместится ли память, выделенная Java, в 8 Гбайт физической памяти? Это зависит от других процессов, работающих на машине. Если на ней выполняются программы Streaming и Pipes, вероятно, такое выделение памяти будет неподходящим (а память, выделенную дочерним процессом, следует сократить), потому что пользовательским процессам (Streaming или Pipes) остается недостаточно памяти. Важно предотвратить выгрузку процессов из памяти на диск, так как это приведет к серьезному снижению быстродействия. Точные настройки памяти сильно зависят от кластера и могут оптимизироваться со временем по результатам наблюдений за использованием памяти в кластере. Для сбора информации хорошо подходят такие инструменты, как Ganglia (см. «GangliaContext», с. 453). О том, как обеспечить соблюдение ограничений памяти задачи, рассказано в разделе «Ограничения памяти задач», с. 410.

В Hadoop также предусмотрены параметры, управляющие использованием памяти для операций MapReduce. Они могут задаваться на уровне заданий; более подробное описание приведено в разделе «Тасовка и сортировка», с. 279.

На основных узлах каждый из демонов узла имен, вторичного узла имен и трекера заданий по умолчанию использует 1000 Мбайт памяти, что в сумме дает 3000 Мбайт.

СКОЛЬКО ПАМЯТИ НУЖНО УЗЛУ ИМЕН?

Работа узла имен может занимать много памяти, так как ссылка на каждый блок каждого файла хранится в памяти. Точную формулу привести трудно, потому что использование памяти зависит от количества блоков на файл, длины имени файла и количества каталогов в файловой системе; кроме того, она может изменяться в зависимости от выпуска Hadoop.

Используемый по умолчанию объем памяти — 1000 Мбайт на узел имен — обычно достаточен для нескольких миллионов файлов, но, как правило, для оценки размеров стоит с запасом выделить 1000 Мбайт на миллион блоков.

Например, в 200-узловом кластере с 4 Тбайт дискового пространства на узел, размером блока 128 Мбайт и коэффициентом репликации 3 хватит места для 2 миллионов блоков (и более): $200 \times 4\ 000\ 000 \text{ Мбайт} / (128 \text{ Мбайт} \times 3)$. Таким образом, в данном случае задание памяти узла имен в 2000 Мбайт станет хорошей отправной точкой.

Память узла имен можно увеличить без изменения памяти, выделяемой другим демонам Hadoop; для этого в переменную `HADOOP_NAMENODE_OPTS` сценария `hadoop-env.sh` включается параметр JVM, задающий размер памяти. `HADOOP_NAMENODE_OPTS` позволяет передавать JVM узла имен дополнительные параметры. Таким образом, например, если вы используете Sun JVM, значение `-Xmx2000m` укажет, что узлу имен должны быть выделены 2000 Мбайт памяти. Если вы измените объем памяти, выделенной узлу имен, не забудьте сделать то же самое для вторичного узла имен (с использованием переменной `HADOOP_SECONDARYNAMENODE_OPTS`), так как его требования к памяти сравнимы с требованиями первичного узла имен. Как правило, вторичный узел имен в такой ситуации запускается на другой машине.

Аналогичные переменные окружения существуют и для других демонов Hadoop, так что при желании вы сможете настроить выделенную им память. За подробностями обращайтесь к `hadoop-env.sh`.

Java

Местонахождение используемой реализации Java определяется переменной `JAVA_HOME` в сценарии `hadoop-env.sh` или переменной окружения `JAVA_HOME`, если

значение не задано в *hadoop-env.sh*. Рекомендуется задать значение в *hadoop-env.sh*, чтобы оно было четко определено в одном месте и чтобы весь кластер гарантированно использовал одну версию Java.

Системные журналы

Системные журналы, создаваемые Hadoop, по умолчанию хранятся в каталоге *\$HADOOP_INSTALL/logs*. Местонахождение системных журналов можно изменить при помощи переменной *HADOOP_LOG_DIR* из *hadoop-env.sh*. Более того, рекомендуется это сделать, чтобы журналы хранились за пределами каталога, в котором установлена система Hadoop.

С этой настройкой файлы журналов хранятся в одном месте, даже в случае изменения установочного каталога из-за обновления. Обычно для этой цели используется каталог */var/log/hadoop*, который задается следующей строкой в сценарии *hadoop-env.sh*:

```
export HADOOP_LOG_DIR=/var/log/hadoop
```

Каждый демон Hadoop, выполняемый на машине, создает два файла журнала. Первый — журнальный вывод, записанный через *log4j*. Этот файл, имеющий суффикс *.log*, должен быть первым источником диагностической информации, потому что большинство журнальных сообщений записывается именно сюда. Стандартная конфигурация *log4j* в Hadoop использует механизм ротации файлов. Старые журналы никогда не удаляются, поэтому вы должны организовать их периодическое удаление или архивацию, чтобы на локальном узле не закончилось дисковое пространство.

Второй журнал объединяет стандартный вывод и стандартный журнал ошибок. Этот файл, имеющий суффикс *.out*, обычно содержит минимум вывода (или вообще не содержит его), так как Hadoop для ведения журналов использует *log4j*. Ротация производится только при перезапуске демона, и сохраняются только пять последних журналов. Старые журналы снабжаются дополнительным суффиксом от 1 до 5 (5 — самый старый файл).

Имена файлов журналов (обоих типов) складываются из имени пользователя, запустившего демон, имени демона и имени хоста. Например, после ротации файл журнала может называться *hadoop-tom-datanode-sturges.local.log.2008-07-04*.

Структура имен позволяет при необходимости заархивировать журналы со всех машин кластера в одном каталоге, так как имена файлов уникальны.

Имя пользователя в имени файла на самом деле используется по умолчанию в соответствии с настройкой *HADOOP_IDENT_STRING* в файле *hadoop-env.sh*. Если вы хотите присвоить экземпляру Hadoop другой идентификатор для имен журналов, задайте нужное значение *HADOOP_IDENT_STRING*.

Настройки SSH

Управляющие сценарии позволяют выполнять команды на (удаленных) рабочих узлах с основного узла через SSH. Настройка SSH может быть полезной по нескольким причинам. Например, можно сократить тайм-аут подключения (параметр `ConnectTimeout`), чтобы управляющие сценарии не «зависали», дожидаясь ответа от неработоспособных узлов. Разумеется, слишком далеко заходить не стоит — при слишком низком тайм-ауте занятые узлы будут проигнорированы, а это плохо.

Другой полезный параметр SSH — `StrictHostKeyChecking`. Если задать ему значение `no`, информация о новых хостах будет автоматически добавляться в известные файлы хостов. Со значением по умолчанию `ask` пользователь должен подтвердить, что ключ прошел проверку; этот режим не подходит для крупного кластера¹.

Чтобы передать SSH дополнительные параметры, определите переменную окружения `HADOOP_SSH_OPTS` в `hadoop-env.sh`. Другие параметры SSH описаны в документации `ssh` и `ssh_config`.

Управляющие сценарии Hadoop могут распространять конфигурационные файлы по всем узлам кластера с использованием `rsync`. По умолчанию эта возможность отключена, но, если определить параметр `HADOOP_MASTER` в сценарии `hadoop-env.sh`, при каждом запуске демона дерево с корне `HADOOP_MASTER` будет синхронизироваться с каталогом `HADOOP_INSTALL` локального узла.

А если в системе имеется два основных узла — узел имен и трекер заданий — на разных машинах? Один из них выбирается как источник, а другой узел синхронизируется по нему вместе со всеми рабочими узлами. Более того, для синхронизации можно использовать любую машину, даже находящуюся за пределами кластера Hadoop.

Так как переменная `HADOOP_MASTER` по умолчанию не задана, возникает проблема «обратной связи»: как убедиться в том, что `hadoop-env.sh` с установленным параметром `HADOOP_MASTER` присутствует на рабочих узлах? Для малых кластеров можно написать маленький сценарий, который копирует `hadoop-env.sh` с основного узла на все рабочие узлы. В больших кластерах такие программы, как `dsh`, могут осуществлять параллельное копирование. Также создание подходящего сценария `hadoop-env.sh` может стать частью сценария автоматизированной установки (например, с использованием Kickstart).

При запуске большого кластера с включенной синхронизацией рабочие узлы стартуют примерно одновременно; поток запросов `rsync` может вызвать перегрузку

¹ Дополнительная информация о безопасности и ключах хостов SSH приведена в статье Брайана Хэтча (Brian Hatch) «SSH Host Key Protection» по адресу <http://www.securityfocus.com/infosec/1806>.

основного узла. Чтобы этого не произошло, задайте переменной `HADOOP_SLAVE_SLEEP` небольшое значение в секундах — например, 0,1 соответствует одной десятой секунды. При выполнении команд на всех узлах кластера основной узел будет делать паузу заданной продолжительности между последовательными выполнениями команды для рабочих машин.

Важные свойства демонов Hadoop

Hadoop содержит огромное количество всевозможных конфигурационных свойств. В этом разделе мы рассмотрим те свойства, которые необходимо определить в любом реальном кластере (или, по крайней мере, понимать, почему значение по умолчанию подходит в данной ситуации). Эти свойства задаются в файлах Hadoop `core-site.xml`, `hdfs-site.xml` и `mapred-site.xml`. Типичное содержимое таких файлов приведено в листингах 9.1, 9.2 и 9.3. Обратите внимание: многие свойства имеют пометку `final`, предотвращающую их переопределение в конфигурациях заданий. О том, как создаются конфигурационные файлы Hadoop, рассказано в разделе «API конфигурации», с. 203.

Листинг 9.1. Типичный файл core-site.xml

```
<?xml version="1.0"?>
<!-- core-site.xml --&gt;
&lt;configuration&gt;
  &lt;property&gt;
    &lt;name&gt;fs.default.name&lt;/name&gt;
    &lt;value&gt;hdfs://namenode/&lt;/value&gt;
    &lt;final&gt;true&lt;/final&gt;
  &lt;/property&gt;
&lt;/configuration&gt;</pre>
```

Листинг 9.2. Типичный файл hdfs-site.xml

```
<?xml version="1.0"?>
<!-- hdfs-site.xml --&gt;
&lt;configuration&gt;
  &lt;property&gt;
    &lt;name&gt;dfs.name.dir&lt;/name&gt;
    &lt;value&gt;/disk1/hdfs/name,/remote/hdfs/name&lt;/value&gt;
    &lt;final&gt;true&lt;/final&gt;
  &lt;/property&gt;
  &lt;property&gt;
    &lt;name&gt;dfs.data.dir&lt;/name&gt;</pre>
```

продолжение ↗

Листинг 9.2 (продолжение)

```
<value>/disk1/hdfs/data,/disk2/hdfs/data</value>
<final>true</final>
</property>

<property>
<name>fs.checkpoint.dir</name>
<value>/disk1/hdfs/namesecondary,/disk2/hdfs/namesecondary</value>
<final>true</final>
</property>
</configuration>
```

Листинг 9.3. Типичный файл mapred-site.xml

```
<?xml version="1.0"?>
<!-- mapred-site.xml -->
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>jobtracker:8021</value>
<final>true</final>
</property>

<property>
<name>mapred.local.dir</name>
<value>/disk1/mapred/local,/disk2/mapred/local</value>
<final>true</final>
</property>

<property>
<name>mapred.system.dir</name>
<value>/tmp/hadoop/mapred/system</value>
<final>true</final>
</property>

<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>7</value>
<final>true</final>
</property>

<property>
<name>mapred.tasktracker.reduce.tasks.maximum</name>
<value>7</value>
```

```
<final>true</final>
</property>

<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx400m</value>
  <!-- Not marked as final so jobs can include JVM debugging options -->
</property>
</configuration>
```

HDFS

Для работы HDFS одну машину необходимо назначить узлом имен. В этом случае свойство `fs.default.name` задает URI файловой системы HDFS, в котором хост определяет имя хоста узла имен или IP-адрес и порт определяют порт, который будет прослушиваться узлом имен для взаимодействий RPC. Если порт не указан, по умолчанию используется порт 8020.



Файл `masters`, используемый управляющими сценариями, не используется демонами HDFS (или MapReduce) для определения имен хостов. Если вы не используете сценарии, на файл `masters` можно не обращать внимания.

Свойство `fs.default.name` также задает файловую систему по умолчанию, которая используется для разрешения относительных путей; это удобно, так как вам не приходится вводить лишние символы, и не нужно жестко программировать адрес конкретного узла имен. Например, с файловой системой по умолчанию, определенной в листинге 9.1, относительный URI `/a/b` преобразуется в `hdfs://namenode/a/b`.



Если вы используете HDFS, тот факт, что свойство `fs.default.name` определяет как узел имен HDFS, так и файловую систему по умолчанию, означает, что HDFS должна быть файловой системой по умолчанию в серверной конфигурации. Однако следует понять, что в клиентской конфигурации для удобства можно задать по умолчанию другую файловую систему.

Например, если вы используете файловые системы HDFS и S3, у вас появляется возможность указать любую из них по умолчанию в клиентской конфигурации. В этом случае вы сможете использовать для файловой системы по умолчанию относительные, а для другой — абсолютные URI.

Есть и другие свойства конфигурации, которые следует задать для HDFS: они определяют каталоги узла имен и узлов данных. Свойство `dfs.name.dir` определяет

список каталогов, в которых узел имен хранит метаданные файловой системы (журнал изменений и образ файловой системы). В целях избыточности в каждом каталоге хранится копия каждого файла метаданных. Свойство `dfs.name.dir` часто настраивается таки образом, чтобы метаданные узла имен записывались на один или два локальных диска, а также на удаленный диск (например, смонтированный каталог NFS). Такая конфигурация защищает от сбоев локального диска и сбоя всего узла имен, поскольку в обоих случаях файлы можно восстановить и использовать для запуска нового узла имен. (Вторичный узел имен получает только периодические контрольные точки узла имен, и не содержит его полной резервной копии).

Также стоит задать свойство `dfs.data.dir` со списком каталогов, в которых узел данных хранит свои блоки. В отличие от узла имен, использующего несколько каталогов для обеспечения избыточности, узел данных осуществляет циклическое чередование записи между своими каталогами, так что для достижения быстродействия следует указать каталог для каждого локального диска. Скорость чтения также повышается от хранения данных на нескольких дисках, потому что блоки распределяются между ними, а параллельные операции чтения разных блоков будут соответственно распределены по дискам.



Для достижения максимальной производительности диски следует монтировать с параметром `noatime`. С этим параметром при операциях чтения не записывается время последнего обращения к файлу, что существенно ускоряет выполнение операций.

Наконец, вы должны указать, в каких каталогах вторичный узел имен хранит контрольные точки файловой системы. Свойство `fs.checkpoint.dir` определяет список каталогов, в которых хранятся контрольные точки. Как и в случае с каталогами узла имен, в которых хранятся избыточные копии метаданных узла, контрольная точка образа файловой системы сохраняется в каждом каталоге для создания избыточности.

В табл. 9.3 приведена сводка важнейших свойств конфигурации HDFS.

Таблица 9.3. Важные свойства демона HDFS

Имя свойства	Тип	Значение по умолчанию	Описание
<code>fs.default.name</code>	URI	<code>file:///</code>	Файловая система по умолчанию. URI определяет имя хоста и порт, на котором работает сервер RPC узла имен. По умолчанию используется порт 8020. Свойство задается в файле <code>core-site.xml</code>

Имя свойства	Тип	Значение по умолчанию	Описание
dfs.name.dir	Имена каталогов, разделенные запятыми	<code> \${hadoop.tmp.dir}/dfs/name</code>	Список каталогов, в которых узел имен хранит свои копии метаданных. Узел имен сохраняет копию метаданных в каждом каталоге из списка
dfs.data.dir	Имена каталогов, разделенные запятыми	<code> \${hadoop.tmp.dir}/dfs/data</code>	Список каталогов, в которых узел данных хранит блоки. Каждый блок хранится только в одном из каталогов
fs.checkpoint.dir	Имена каталогов, разделенные запятыми	<code> \${hadoop.tmp.dir}/dfs/name-secondary</code>	Список каталогов, в которых вторичный узел имен хранит контрольные точки. Копия контрольной точки сохраняется в каждом каталоге из списка

Для HDFS каталоги хранения данных по умолчанию находятся во временном каталоге Hadoop (свойство `hadoop.tmp.dir` по умолчанию содержит `/tmp/hadoop-${user.name}`). Эти свойства следует задать так, чтобы очистка временных каталогов не приводила к потере данных.

MapReduce

Для работы MapReduce необходимо назначить одну машину трекером заданий; в малых кластерах это может быть та же машина, которая работает узлом имен. Для этого свойству `mapred.job.tracker` задается имя хоста или IP-адрес и порт, на котором будет вести прослушивание трекер заданий. Обратите внимание: свойство содержит не URI, а пару «хост/порт», разделенные двоеточием. Обычно используется порт 8021.

Во время выполнения задания MapReduce промежуточные данные и рабочие файлы записываются во временные локальные файлы. Так как данные могут содержать вывод задач отображения, который теоретически может быть очень большим, необходимо проследить за тем, чтобы дисковые разделы, указанные в свойстве `mapred.local.dir`, были достаточно большими. Свойство `mapred.local.dir` содержит список имен каталогов, разделенных запятыми; используйте все доступные локальные диски для распределения дисковых операций ввода/вывода. Как правило, для временных данных MapReduce используются те же диски и разделы (но другие каталоги), что и для блоков узлов данных (свойство `dfs.data.dir`, о котором говорилось ранее).

MapReduce использует распределенную файловую систему для совместного использования файлов (например, JAR-файла задания) трекерами задач, на которых

выполняются задачи MapReduce. Свойство `mapred.system.dir` задает каталог, в котором могут храниться эти файлы. Имя каталога разрешается относительно файловой системы по умолчанию (заданной `fs.default.name`) — обычно это HDFS.

Наконец, свойства `mapred.tasktracker.map.tasks.maximum` и `mapred.tasktracker.reduce.tasks.maximum` задают количество доступных ядер на машинах трекера задач, а свойство `mapred.child.java.opts` — объем памяти, доступной для дочерних JVM трекера задач (см. раздел «Память», с. 396).

В табл. 9.4 приведена сводка важнейших свойств конфигурации MapReduce.

Таблица 9.4. Важные свойства демона MapReduce

Имя свойства	Тип	Значение по умолчанию	Описание
<code>mapred.job.tracker</code>	Имя хоста и порт	local	Имя хоста и порт, на котором работает сервер RPC трекера задач. Со значением по умолчанию local трекер запускается во внутрипроцессном режиме при запуске задания MapReduce (запускать трекер заданий в этой ситуации не нужно; более того, при попытке запустить его вы получите ошибку)
<code>mapred.local.dir</code>	Имена каталогов, разделенные запятыми	<code> \${hadoop.tmp.dir} /mapred/local</code>	Список каталогов, в которых MapReduce хранит промежуточные данные заданий. Данные удаляются при завершении задания
<code>mapred.system.dir</code>	URI	<code> \${hadoop.tmp.dir} /mapred/system</code>	Каталог относительно <code>fs.default.name</code> , в котором во время выполнения задания хранятся общие файлы
<code>mapred.tasktracker.map.tasks.maximum</code>	int	2	Количество задач отображения, которые могут одновременно выполняться на трекере задач
<code>mapred.tasktracker.reduce.tasks.maximum</code>	int	2	Количество задач свертки, которые могут одновременно выполняться на трекере задач
<code>mapred.child.java.opts</code>	String	<code>-Xmx200m</code>	Параметры JVM, используемые при запуске дочернего процесса трекера задач, выполняющего задачи отображения и свертки. Свойство может задаваться на уровне заданий, что может быть полезно, например, для настройки свойств JVM при отладке

Имя свойства	Тип	Значение по умолчанию	Описание
mapreduce.map.java.opts	String	-Xmx200m	Параметры JVM, используемые для дочернего процесса, выполняющего задачи отображения. (Недоступно в 1.x)
mapreduce.reduce.java.opts	String	-Xmx200m	Параметры JVM, используемые для дочернего процесса, выполняющего задачи свертки. (Недоступно в 1.x)

Адреса и порты демонов Hadoop

Демоны Hadoop обычно запускают как сервер RPC (табл. 9.5) для взаимодействия между демонами, так и сервер HTTP, предоставляющий веб-страницы для пользователей (табл. 9.6). Для каждого сервера указывается сетевой адрес и номер порта, на котором должно вестись прослушивание. При указании сетевого адреса 0.0.0.0 Hadoop привязывается ко всем адресам на машине; также можно указать один конкретный адрес. Номер порта 0 приказывает серверу запуститься со свободным портом, но обычно так поступать не рекомендуется из-за нарушения политики управления трафиком на уровне кластера.

Таблица 9.5. Свойства сервера RPC

Имя свойства	Значение по умолчанию	Описание
fs.default.name	file:///	Если это свойство содержит HDFS URI, оно определяет адрес и порт сервера RPC узла имен. Если порт не задан, по умолчанию используется порт 8020
dfs.datanode.ipc.address	0.0.0.0:50020	Адрес и порт сервера RPC узла данных.
mapred.job.tracker	local	Если это свойство содержит имя хоста и порт, оно определяет адрес и порт сервера RPC трекера заданий. Обычно используется порт 8021
mapred.task.tracker.report.address	127.0.0.1:0	Адрес и порт сервера RPC трекера задач. Информация используется дочерней JVM трекера задач для взаимодействия с трекером. Использовать можно любой свободный порт, так как сервер связывается только с адресом обратной связи. Свойство следует изменять, только если машина не имеет адреса обратной связи

Кроме сервера RPC, узлы данных запускают сервер TCP/IP для передачи блоков. Адрес и порт сервера задаются свойством `dfs.datanode.address`; по умолчанию используется значение 0.0.0.0:50010.

Таблица 9.6. Свойства сервера HTTP

Имя свойства	Значение по умолчанию	Описание
<code>mapred.job.tracker.http.address</code>	0.0.0.0:50030	Адрес и порт сервера HTTP трекера задач
<code>mapred.task.tracker.http.address</code>	0.0.0.0:50060	Адрес и порт сервера HTTP трекера задач
<code>dfs.http.address</code>	0.0.0.0:50070	Адрес и порт сервера HTTP узла имен
<code>dfs.datanode.http.address</code>	0.0.0.0:50075	Адрес и порт сервера HTTP узла данных
<code>dfs.secondary.http.address</code>	0.0.0.0:50090	Адрес и порт сервера HTTP вторичного узла имен

Также предусмотрена возможность управления тем, какие сетевые интерфейсы узлы данных и трекеры задач будут сообщать как свои IP-адреса (для серверов HTTP и RPC). Оба свойства — `dfs.datanode.dns.interface` и `mapred.tasktracker.dns.interface` — имеют значения по умолчанию, при которых используется сетевой интерфейс по умолчанию. Также можно задать им адрес конкретного сетевого интерфейса (например, `eth0`).

Другие свойства Hadoop

В этом разделе рассматриваются другие свойства, которые тоже могут пригодиться при организации кластера Hadoop.

Принадлежность к кластеру

Для упрощения будущих операций добавления и удаления узлов можно создать файл со списком машин, которые могут присоединяться к кластеру в качестве узлов данных или трекеров задач. Этот файл задается свойствами `dfs.hosts` и `mapred.hosts` (для узлов данных и трекеров задач соответственно), а также соответствующими свойствами `dfs.hosts.exclude` и `mapred.hosts.exclude` для определения исключений. За подробностями обращайтесь к разделу «Включение и исключение узлов», с. 459.

Размер буфера

В операциях ввода/вывода Hadoop используется буфер размером 4 Кбайт (4096 байт). На современном оборудовании и операционных системах увеличение размера буфера обычно приводит к повышению производительности; 128 Кбайт (131072 байт) – стандартный выбор. Значение задается свойством `io.file.buffer.size` в файле `core-site.xml`.

Размер блока HDFS

Размер блока HDFS по умолчанию составляет 64 Мбайт, но во многих кластерах используются значения 128 Мбайт (134 217 728 байт) и даже 256 Мбайт (268 435 456 байт); это способствует снижению нагрузки на узел имен и дает отображениям больше данных для работы. Значение задается свойством `dfs.block.size` в файле `hdfs-site.xml`.

Зарезервированное дисковое пространство

По умолчанию узлы данных пытаются использовать все доступное пространство в своих каталогах хранения данных. Если вы хотите зарезервировать часть пространства томов для других целей, кроме HDFS, задайте свойству `dfs.datanode.du.reserved` величину резервируемого пространства в байтах.

Отложенное удаление

Файловые системы Hadoop поддерживают механизм отложенного удаления: удаленные файлы не уничтожаются, а перемещаются в специальный «мусорный» каталог, в котором остаются в течение некоторого времени до того, как будут безвозвратно уничтожены системой. Минимальный период хранения удаленных файлов задается свойством конфигурации `fs.trash.interval` в файле `core-site.xml`. По умолчанию интервал равен нулю, то есть отложенное удаление отключено.

Как и во многих операционных системах, механизм отложенного удаления Hadoop является функцией пользовательского уровня. Файлы, удаленные на программном уровне, уничтожаются немедленно. Впрочем, отложенное удаление можно использовать и на программном уровне – сконструируйте экземпляр `Trash`, а затем вызовите его метод `moveToTrash()` с объектом `Path` файла, предназначенного для удаления. Метод возвращает признак успешного выполнения; значение `false` означает, что либо отложенное удаление недоступно, либо файл уже находится в «мусорном» каталоге.

Если отложенное удаление включено, у каждого пользователя в домашнем каталоге имеется собственный «мусорный» каталог с именем `.Trash`. Восстановление

файлов проходит просто; достаточно найти файл в подкаталоге *.Trash* и переместить его из поддерева *trash*.

HDFS автоматически удаляет файлы в «мусорных» каталогах, но другие файловые системы этого не делают, поэтому вам придется периодически удалять их самостоятельно. Для удаления файлов, находящихся в «мусорном» каталоге дольше минимального периода, можно воспользоваться оболочкой файловой системы:

```
% hadoop fs -expunge.
```

Класс `Trash` предоставляет метод `expunge()`, который делает то же самое.

Планировщик заданий

Подумайте, не стоит ли заменить планировщик заданий по умолчанию одной из более функциональных альтернатив (см. «Планирование заданий», с. 277).

Ускорение запуска

По умолчанию планировщики ожидают завершения 5% задач отображения в задании, прежде чем планировать задачи свертки для того же задания. В больших заданиях задержка может вызвать проблемы с эффективностью использования кластера, так как задачи занимают слоты свертки в ожидании завершения задач отображения. Если задать свойству `mapred.reduce.slowstart.completed.maps` более высокое значение — например, 0,80 (80%), это будет способствовать повышению эффективности.

Ограничения памяти задач

В общих кластерах ошибки в программе MapReduce одного пользователя не должны выводить из строя узлы кластера. Подобные сбои возможны, например, при утечке памяти в задачах отображения и свертки, потому что нехватка свободной памяти на машине, на которой работает трекер задач, отразится на других выполняемых процессах.

Или рассмотрим другую ситуацию: пользователь увеличивает значение `mapred.child.java.opts`, лишая другие выполняемые задачи памяти и заставляя их выгружаться на диск. Пометка этого свойства ключевым словом `final` для кластера предотвратит его изменение пользователями в их заданиях, однако для использования некоторыми заданиями дополнительной памяти есть веские причины, поэтому такое решение не всегда приемлемо. Более того, даже блокировка `mapred.child.java.opts` не решит проблемы, потому что задачи могут порождать новые процессы, не ограничиваемые в использовании памяти. Например, задания Streaming и Pipes поступают именно так.

Для предотвращения подобных ситуаций необходимы средства контроля за использованием памяти задачами. Hadoop предоставляет два таких механизма. Более простой способ основан на использовании команды Linux *ulimit*; это может делаться на уровне операционной системы (в файле *limits.conf*, обычно находящемся в */etc/security*) или настройкой свойства *mapred.child.ulimit* в конфигурации Hadoop. Значение задается в килобайтах; оно должно «с запасом» превышать память JVM, заданную свойством *mapred.child.java.opts*; в противном случае дочерняя JVM может не запуститься.

Второй механизм — система мониторинга памяти задач Hadoop¹. Администратор устанавливает диапазон разрешенных лимитов виртуальной памяти для задач кластера, а пользователи указывают максимальные требования к памяти в конфигурации своих заданий. Если пользователь не указал требования к памяти, используются значения по умолчанию (*mapred.job.map.memory.mb* и *mapred.job.reduce.memory.mb*).

У этого механизма есть пара преимуществ перед механизмом *ulimit*. Во-первых, он контролирует использование памяти всем деревом процессов задачи, включая порожденные процессы. Во-вторых, он делает возможным планирование с учетом памяти: задачи планируются на трекерах, имеющих достаточно свободной памяти для их выполнения. Например, Capacity Scheduler использует настройки памяти при распределении слотов, и, если у задания *mapred.job.map.memory.mb* превышает *mapred.cluster.map.memory.mb*, планировщик выделяет на трекере задач более одного слота для выполнения задач отображения этого задания.

Для включения мониторинга памяти задач нужно задать все шесть свойств из табл. 9.7. Значения по умолчанию для всех свойств равны –1 (функция недоступна).

Таблица 9.7. Свойства мониторинга памяти задач MapReduce

Имя свойства	Тип	Значение по умолчанию	Описание
<i>mapred.cluster.map.memory.mb</i>	int	–1	Объем виртуальной памяти (в Мбайтах), определяющий слот отображения. Задачи отображения, требующие большего объема памяти, используют несколько слотов отображения
<i>mapred.cluster.reduce.memory.mb</i>	int	–1	Объем виртуальной памяти (в мегабайтах), определяющий слот свертки. Задачи свертки, требующие большего объема памяти, используют несколько слотов свертки

продолжение ↗

¹ YARN использует другую модель памяти с другими параметрами конфигурации. См. раздел «Память», с. 396.

Таблица 9.7 (продолжение)

Имя свойства	Тип	Значение по умолчанию	Описание
mapred.job.map.memory.mb	int	-1	Объем виртуальной памяти (в мегабайтах), необходимый для выполнения задачи отображения. Если задача превысит заданный лимит, она может быть аварийно завершена и помечена как сбойная
mapred.job.reduce.memory.mb	int	-1	Объем виртуальной памяти (в мегабайтах), необходимый для выполнения задачи свертки. Если задача превысит заданный лимит, она может быть аварийно завершена и помечена как сбойная
mapred.cluster.max.map.memory.mb	int	-1	Максимальное значение, которое пользователь может задать свойству mapred.job.map.memory.mb
mapred.cluster.max.reduce.memory.mb	int	-1	Максимальное значение, которое пользователь может задать свойству mapred.job.reduce.memory.mb

Создание учетных записей пользователей

Когда кластер Hadoop будет настроен и готов к работе, необходимо предоставить пользователям доступ к нему. Создайте домашний каталог для каждого пользователя и задайте права доступа к нему:

```
% hadoop fs -mkdir /user/username
% hadoop fs -chown username:username /user/username
```

Заодно установите для каталога ограничения по занимаемому пространству. Следующая команда устанавливает для пользовательского каталога ограничение в 1 Тбайт:

```
% hadoop dfsadmin -setSpaceQuota 1t /user/username
```

Конфигурация YARN

YARN — архитектура нового поколения для выполнения приложений MapReduce (см. «YARN (MapReduce 2)», с. 265). Она поддерживает другой набор демонов и параметров конфигурации, чем классическая архитектура MapReduce (также называемая MapReduce 1). В этом разделе мы рассмотрим различия между

двумя архитектурами и поговорим о том, как выполнять приложения MapReduce в YARN.

В YARN вместо трекеров заданий и трекеров задач имеется единый менеджер ресурсов, работающий на одной машине с узлом имен HDFS (в малых кластерах) или на выделенной машине, и менеджеры узлов, работающие на каждом рабочем узле в кластере.

Сценарий YARN *start-yarn.sh* (в каталоге *sbin*) запускает демоны YARN в кластере. Сценарий запускает менеджер ресурсов (на машине, на которой выполняется сценарий) и менеджеры узлов на всех машинах, перечисленных в файле *slaves*.

В YARN также имеется демон сервера истории заданий, который предоставляет пользователям подробную информацию о прошлых выполнениях заданий, и прокси-сервер веб-приложений, предоставляющий безопасный доступ к пользовательскому интерфейсу приложений YARN. В случае MapReduce веб-интерфейс, предоставляемый прокси-сервером, выдает информацию о текущем задании (см. «Веб-интерфейс MapReduce», с. 229). По умолчанию прокси-сервер веб-приложений работает в одном процессе с менеджером ресурсов, но его можно настроить на работу в режиме автономного демона.

YARN использует свой набор конфигурационных файлов, перечисленных в табл. 9.8; они используются в дополнение к файлам из табл. 9.1.

Таблица 9.8. Конфигурационные файлы YARN

Имя файла	Формат	Описание
yarn-env.sh	Сценарий Bash	Переменные окружения, используемые сценариями выполнения YARN
yarn-site.xml	Конфигурация Hadoop в формате XML	Параметры конфигурации демонов YARN: менеджер ресурсов, сервер истории заданий, прокси-сервер веб-приложений, менеджеры узлов

Важные свойства демонов YARN

При выполнении приложений MapReduce в архитектуре YARN файл *mapred-site.xml* также используется для общих свойств MapReduce, хотя свойства, относящиеся к трекеру заданий и трекерам задач, игнорируются. Ни одно из свойств в табл. 9.4 не относится к YARN, кроме *mapred.child.java.opts* (и сопутствующих свойств *mapreduce.map.java.opts* и *mapreduce.reduce.java.opts*, применяемых только к задачам отображения и свертки соответственно). Параметры JVM,

заданные таким образом, используются для запуска дочернего процесса YARN, выполняющего задачи отображения или свертки.

В конфигурационных файлах, приведенных в листинге 9.4, представлены некоторые важные свойства конфигурации, используемые для выполнения приложений MapReduce в YARN.

Листинг 9.4. Пример конфигурационных файлов для выполнения приложений MapReduce в архитектуре YARN

```
<?xml version="1.0"?>
<!-- mapred-site.xml -->
<configuration>
  <property>
    <name>mapred.child.java.opts</name>
    <value>-Xmx400m</value>
    <!-- Not marked as final so jobs can include JVM debugging options -->
  </property>
</configuration>
<?xml version="1.0"?>
<!-- yarn-site.xml -->
<configuration>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>resourcemanager:8032</value>
  </property>
  <property>
    <name>yarn.nodemanager.local-dirs</name>
    <value>/disk1/nm-local-dir,/disk2/nm-local-dir</value>
    <final>true</final>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce.shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>8192</value>
  </property>
</configuration>
```

Свойство `yarn.resourcemanager.address`, управляющее менеджером ресурсов YARN, получает пару «хост-порт». В клиентской конфигурации оно используется для подключения к менеджеру ресурсов (через RPC); кроме того, свойству `mapreduce.framework.name` должно быть присвоено значение `yarn`, чтобы клиент использовал YARN вместо локального исполнителя заданий.

Хотя YARN игнорирует значение `mapred.local.dir`, имеется эквивалентное свойство `yarn.nodemanager.local-dirs`, при помощи которого можно задать локальные диски для хранения промежуточных данных. Значение свойства представляет собой список локальных путей, разделенных запятыми, которые используются по схеме циклического чередования.

В YARN нет трекеров задач, предоставляющих выходные данные отображений задачам свертки, поэтому для выполнения этой функции YARN полагается на *обработчиков тасовки* (shuffle handlers) — вспомогательные службы, выполняемые на менеджерах узлов. Так как YARN является архитектурой общего назначения, обработчики тасовки MapReduce необходимо явно включить в файле `yarn-site.xml`, задав свойству `yarn.nodemanager.aux-services` значение `mapreduce.shuffle`.

В табл. 9.9 приведена сводка важных свойств конфигурации YARN.

Таблица 9.9. Важные свойства демонов YARN

Имя свойства	Тип	Значение по умолчанию	Описание
<code>yarn.resource-manager.address</code>	Имя хоста и порт	0.0.0.0:8032	Имя хоста и порт, на котором работает сервер RPC менеджера ресурсов
<code>yarn.nodemanager.local-dirs</code>	Имена каталогов, разделенные запятыми	<code>/tmp/nm-local-dir</code>	Список каталогов, в которых менеджеры узлов разрешают контейнерам хранить промежуточные данные. Данные удаляются при завершении задания
<code>yarn.nodemanager.aux-services</code>	Имена служб, разделенные запятыми		Список вспомогательных служб, выполняемых менеджером узла. Служба реализуется классом, определенным свойством <code>yarn.nodemanager.aux-services.service-name.class</code> . По умолчанию вспомогательные службы не заданы
<code>yarn.nodemanager.resource.memory-mb</code>	int	8192	Объем физической памяти (в мегабайтах), которая может быть выделена контейнерам, находящимся под управлением менеджера узла
<code>yarn.nodemanager.vmem-pmem-ratio</code>	float	2.1	Соотношение виртуальной и физической памяти для контейнеров. Использование виртуальной памяти может превышать использование физической памяти в заданной пропорции

Память

YARN работает с памятью на более детализированном уровне, чем в модели слотов, используемой классической реализацией MapReduce. Вместо указания фиксированного максимального количества слотов отображения и свертки, которые могут работать на трекере задач одновременно, YARN позволяет приложениям запросить для задачи произвольный объем памяти (в пределах установленных ограничений). В модели YARN менеджеры узлов выделяют память из пула, поэтому количество задач, выполняемых на конкретном узле, зависит от их суммарных требований к памяти, а не просто от фиксированного количества слотов.

Модель слотов может привести к неэффективному использованию кластера, так как отношение количеств слотов отображения и слотов свертки фиксируется в конфигурации уровня кластера. Однако количество необходимых слотов отображения и свертки изменяется со временем: в начале выполнения задания нужны только слоты отображения, тогда как в конце нужны только слоты свертки. В крупных кластерах с множеством параллельно выполняемых заданий потребность в слотах определенного типа менее ярко выражена, но без потерь все равно не обойтись. В YARN этой проблемы нет, так как типы слотов не различаются.

Факторы, влияющие на объем памяти, выделяемой менеджеру узлов для управления контейнерами, сходны с теми, которые обсуждались в разделе «Память», с. 396. Каждый демон Hadoop использует 1000 Мбайт; следовательно, для узла данных и менеджера узлов суммарные затраты составят 2000 Мбайт. Зарезервируйте необходимую память для других процессов, выполняемых на машине, а остаток можно будет выделить контейнерам менеджера узлов. Для этого свойству `yarn.nodemanager.resource.memory-mb` задается общий объем памяти в мегабайтах (по умолчанию 8192 Мбайт).

Следующий шаг — настройка памяти для отдельных заданий. Она определяется двумя свойствами: `mapred.child.java.opts` позволяет задать размер кучи JVM задачи отображения или свертки, а `mapreduce.map.memory.mb` (или `mapreduce.reduce.memory.mb`) определяет объем памяти, необходимой контейнерам задач отображения (или свертки). Последнее свойство используется контроллером приложений при конкурентном распределении ресурсов кластера, а также менеджером узла, который запускает контейнеры задач и наблюдает за их выполнением.

Допустим, свойству `mapred.child.java.opts` задано значение `-Xmx800m`, а свойству `mapreduce.map.memory.mb` оставлено значение по умолчанию 1024 Мбайт. При запуске задачи отображения менеджер узла выделяет контейнер размером 1024 Мбайт (уменьшая размер пула на указанную величину на время выполнения задачи) и запускает JVM задачи, настроенную с максимальным размером кучи 800 Мбайт. Обратите внимание: объем памяти процесса JVM превышает размер кучи, а превышение зависит от таких факторов, как используемые библиотеки,

размер пространства Permanent Generation Space и т. д. Здесь важно то, что физическая память, используемая процессом JVM, включая все порожденные процессы (например, процессы Streaming или Pipes), не может превышать выделенную (1024 Мбайт). Если контейнер использует больше памяти, чем положено, менеджер узла может завершить его и пометить как сбойный.

Планировщики могут устанавливать минимальные и максимальные ограничения для выделяемой памяти. Например, для Capacity Scheduler минимум по умолчанию составляет 1024 Мбайт (свойство `yarn.scheduler.capacity.minimum-allocation-mb`), а максимум по умолчанию составляет 10240 Мбайт (свойство `yarn.scheduler.capacity.maximum-allocation-mb`).

Также существуют ограничения по виртуальной памяти, которые должны соблюдаться контейнером. Если использование виртуальной памяти контейнером превысит заданную величину (выделенная физическая память, умноженная на некоторый коэффициент), менеджер узла может завершить процесс. Коэффициент определяется свойством `yarn.nodemanager.vmem-rmem-ratio`, по умолчанию равным 2.1. В приведенном выше примере порог виртуальной памяти, на котором может произойти завершение задачи, составляет 2150 Мбайт, то есть $2,1 \times 1024$ Мбайт.

Во время настройки параметров памяти очень полезно следить за фактическим использованием памяти процессом во время выполнения задания; в этом вам помогут счетчики задач MapReduce. Счетчики `PHYSICAL_MEMORY_BYTES`, `VIRTUAL_MEMORY_BYTES` и `COMMITTED_HEAP_BYTES` (см. табл. 8.2) дают оперативную информацию об использовании памяти, а следовательно, хорошо подходят для наблюдения во время попыток выполнения задачи.

Адреса и порты демонов YARN

Демоны YARN запускают один или несколько серверов RPC и HTTP; подробная информация приведена в табл. 9.10 и 9.11.

Таблица 9.10. Свойства серверов RPC YARN

Имя файла	Значение по умолчанию	Описание
<code>yarn.resourcemanager.address</code>	0.0.0.0:8032	Адрес и порт сервера RPC менеджера ресурсов. Используется клиентом (как правило, находящимся за пределами кластера) для взаимодействия с менеджером ресурсов

продолжение ➔

Таблица 9.10 (продолжение)

Имя файла	Значение по умолчанию	Описание
yarn.resourcemanager.admin.address	0.0.0.0:8033	Административный адрес и порт сервера RPC менеджера ресурсов. Используется клиентом-администратором (вызывается командой <code>yarn rmadmin</code> , обычно выполняемой за пределами кластера) для взаимодействия с менеджером ресурсов
yarn.resourcemanager.scheduler.address	0.0.0.0:8030	Адрес и порт сервера RPC планировщика менеджера ресурсов. Используется (внутри кластера) контроллерами приложений для взаимодействия с менеджером ресурсов
yarn.resourcemanager.resource-tracker.address	0.0.0.0:8031	Адрес и порт сервера RPC трекера ресурсов, принадлежащего менеджеру ресурсов. Используется (внутри кластера) менеджерами узлов для взаимодействия с менеджером ресурсов
yarn.nodemanager.address	0.0.0.0:0	Адрес и порт сервера RPC менеджера узла. Используется (внутри кластера) контроллерами приложений для взаимодействия с менеджерами узлов
yarn.nodemanager.localizer.address	0.0.0.0:8040	Адрес и порт сервера RPC локализатора менеджера узла
mapreduce.jobhistory.address	0.0.0.0:10020	Адрес и порт сервера RPC сервера истории заданий. Используется клиентом (как правило, находящимся за пределами кластера) для получения информации о истории заданий. Свойство задается в <code>mapred-site.xml</code>

Таблица 9.11. Свойства серверов HTTP YARN

Имя файла	Значение по умолчанию	Описание
yarn.resourcemanager.webapp.address	0.0.0.0:8088	Адрес и порт сервера HTTP менеджера ресурсов
yarn.nodemanager.webapp.address	0.0.0.0:8042	Адрес и порт сервера HTTP менеджера узла
yarn.web-proxy.address		Адрес и порт сервера HTTP прокси-сервера веб-приложений. Если значение не задано (по умолчанию), прокси-сервер веб-приложений будет выполняться в процессе менеджера ресурсов

Имя файла	Значение по умолчанию	Описание
mapreduce.jobhistory.webapp.address	0.0.0.0:19888	Адрес и порт сервера RPC сервера истории заданий. Свойство задается в mapred-site.xml
mapreduce.shuffle.port	8080	Номер порта HTTP обработчика тасовки. Используется для передачи выходных данных отображений, а не для обращения к веб-интерфейсу. Свойство задается в mapred-site.xml

Безопасность

Ранние версии Hadoop предполагали, что кластеры MapReduce и HDFS будут использоваться группами взаимодействующих пользователей в безопасной среде. Меры ограничения доступа проектировались для предотвращения случайной потери данных, а не несанкционированного доступа к ним. Например, система разрешений в HDFS предотвратит случайное удаление всей файловой системы из-за ошибки в программе или случайно введенной команды *hadoop fs -rmr /*, но не помешает злоумышленнику выдать себя за привилегированного пользователя (см. «Задание идентификатора пользователя», с. 210) для обращения или удаления любых данных в кластере.

В системе отсутствовал безопасный механизм аутентификации, который позволял бы Hadoop убедиться в том, что пользователь, желающий выполнить операцию в кластере, — тот, за кого он себя выдает, и ему можно доверять. Файловые разрешения HDFS предоставляют только механизм авторизации, то есть определяют операции, которые конкретный пользователь может выполнить с конкретным файлом. Например, чтение файла может быть разрешено только определенной группе пользователей, и ни один пользователь, не входящий в эту группу, прочитать файл не сможет. Однако одной авторизации недостаточно, потому что злоумышленник с сетевым доступом к кластеру может получить доступ к системе посредством фальсификации личных данных.

На практике доступ к данным, содержащим личную информацию (например, полное имя или IP-адрес пользователя), ограничивается небольшим подмножеством пользователей (кластера) из организации, которой разрешен доступ к этой информации. Менее секретные (или анонимные) данные могут быть доступны для большего круга пользователей. Часто бывает удобно разместить в одном кластере разные наборы данных с разными уровнями безопасности (от части и потому, что это позволит организовать совместное использование наборов данных с более низким уровнем безопасности). Однако для выполнения нормативных требований по защите данных в общих кластерах должна быть организована безопасная аутентификация.

Так выглядела ситуация в 2009 году, когда фирма Yahoo! поручила группе инженеров реализовать безопасный механизм аутентификации для Hadoop. Система Hadoop сама по себе не управляет учетными данными пользователей; вместо этого она пользуется сервисом Kerberos — проверенным протоколом сетевой аутентификации с открытым кодом. В свою очередь, Kerberos не управляет разрешениями. Kerberos только подтверждает, что пользователь является тем, за кого себя выдает; задача Hadoop — определить, разрешено ли пользователю выполнение заданного действия. Kerberos — достаточно обширная тема, поэтому здесь мы рассматриваем только использование Kerberos в контексте Hadoop. Читателям, которым нужна более подробная информация, рекомендуем обратиться к книге Джейсона Гармана (Jason Garman) «Kerberos: The Definitive Guide» (O'Reilly, 2003).

КАКИЕ ВЕРСИИ HADOOP ПОДДЕРЖИВАЮТ АУТЕНТИФИКАЦИЮ KERBEROS?

Поддержка Kerberos для аутентификации впервые появилась в Apache Hadoop серии 0.20.20x. В табл. 1.2 приведена информация о том, какие из последних версий поддерживают эту возможность.

Kerberos и Hadoop

Клиент, желающий получить доступ к сервису с использованием Kerberos, должен пройти три фазы, каждая из которых требует обмена сообщениями с сервером:

- *Аутентификация.* Клиент проходит проверку сервера аутентификации (AS, Authentication Server) и получает *билет для получения билета* TGT (Ticket-Granting Ticket) с временной меткой.
- *Авторизация.* Клиент использует TGT для запроса билета сервиса (service ticket) у сервера выдачи билетов (TGS, Ticket Granting Server).
- *Запрос на обслуживание.* Клиент использует билет сервиса для подтверждения своей подлинности на сервере, который предоставляет услугу, используемую клиентом. В случае Hadoop это может быть узел имен или трекер заданий.

Сервер AS вместе с сервером TGS образуют KDC (Key Distribution Center). Процесс в графическом виде представлен на рис. 9.2.

Фазы авторизации и запроса сервиса не являются действиями пользовательского уровня; клиент выполняет эти шаги от имени пользователя. Вместе с тем фаза

аутентификации обычно явно выполняется пользователем при помощи команды *kinit*, которая запрашивает пароль. Однако это не означает, что вам придется вводить пароль при каждом запуске задания или обращении к HDFS, потому что TFT по умолчанию действует 10 часов (а может быть продлен до недели). На практике аутентификация часто автоматизируется в процессе входа в операционную систему, благодаря чему начинает действовать механизм *единого входа* в Hadoop.

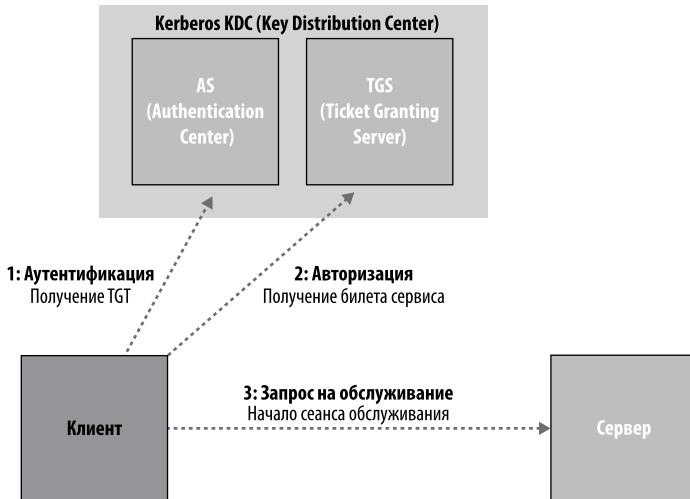


Рис. 9.2. Трехфазный протокол обмена билетами Kerberos

Если вы не хотите, чтобы система запрашивала у вас пароль (например, при запуске задания MapReduce в автоматическом режиме), создайте файл Kerberos *keytab* при помощи команды *ktutil*. Файл *keytab* содержит пароли и может передаваться *kinit* с параметром *-t*.

Пример

Рассмотрим пример процесса аутентификации. Прежде всего необходимо включить аутентификацию Kerberos, задав свойству *hadoop.security.authentication* в файле *core-site.xml* значение *kerberos*¹. По умолчанию используется значение

¹ Чтобы использовать аутентификацию Kerberos с Hadoop, необходимо установить, настроить и запустить KDC (в поставку Hadoop этот компонент не включен). Возможно, в вашей организации уже имеется сервер KDC (например, установка Active Directory); если же его нет, установите MIT Kerberos 5 KDC по инструкциям, приведенным в книге «Linux Security Cookbook» (O'Reilly, 2003).

`simple`, с которым для идентификации пользователя используется старый совместимый (но небезопасный) механизм проверки имени пользователя операционной системы.

Также необходимо включить авторизацию сервисного уровня, задав в том же файле свойству `hadoop.security.authorization` значение `true`. В файле `hadoop-policy.xml` можно настроить списки ACL (Access Control List), управляющие тем, каким пользователям и группам разрешено подключение к тем или иным сервисам Hadoop. Сервисы определяются на уровне протокола; есть службы для отправки заданий MapReduce, взаимодействий с узлами и т. д. По умолчанию все списки ACL содержат символ `*`, означающий, что всем пользователям разрешен доступ ко всем службам, но в реальном кластере доступ следует ограничить.

Формат списка ACL представляет собой перечень имен пользователей, разделенных запятыми, за которым через пробел следует перечень имен групп, разделенных запятыми. Например, список ACL `preston,howard directors,inventors` разрешит доступ пользователям `preston` и `howard`, а также участникам групп `directors` и `inventors`.

Посмотрим, что произойдет при попытке копирования локального файла в HDFS при включенной аутентификации Kerberos:

```
% hadoop fs -put quangle.txt .
10/07/03 15:44:58 WARN ipc.Client: Exception encountered while connecting to the
server: javax.security.sasl.SaslException: GSS initiate failed [Caused by GSSEx
ception: No valid credentials provided (Mechanism level: Failed to find any Ker
beros tgt)]
Bad connection to FS. command aborted. exception: Call to localhost/127.0.0.1:80
20 failed on local exception: java.io.IOException: javax.security.sasl.SaslExcep
tion: GSS initiate failed [Caused by GSSEception: No valid credentials provided
(Mechanism level: Failed to find any Kerberos tgt)]
```

Попытка выполнения операции завершилась неудачей, потому что у нас нет билета Kerberos. Чтобы получить билет, следует пройти аутентификацию в KDC с использованием команды `kinit`:

```
% kinit
Password for hadoop-user@LOCALDOMAIN: password
% hadoop fs -put quangle.txt .
% hadoop fs -stat %n quangle.txt
quangle.txt
```

На этот раз файл был успешно записан в HDFS. Обратите внимание: хотя мы выполнили две команды файловой системы, `kinit` пришлось вызвать только один раз, потому что билет Kerberos остается действительным 10 часов (команда `klist`

выводит срок действия билетов, а команда *kdestroy* делает билеты недействительными). После получения билета все работает так, как и должно.

Маркеры делегирования

В работе распределенных систем — таких, как HDFS или MapReduce — задействованы многочисленные взаимодействия «клиент-сервер», каждое из которых должно пройти аутентификацию. Например, операция чтения HDFS требует нескольких обращений к узлу имен и одному или нескольким узлам данных. Так как использование трехфазного протокола обмена билетами Kerberos для аутентификации каждого обращения в загруженном кластере создаст высокую нагрузку на KDC, Hadoop использует *маркеры делегирования* (delegation tokens) для аутентификации последующих обращений без повторной связи с сервером KDC. Hadoop создает и использует маркеры делегирования в прозрачном режиме от имени пользователей, поэтому от пользователя не потребуется ничего, кроме выполнения команды *kinit* для входа. Тем не менее желательно хотя бы в общих чертах представлять, как работает этот механизм.

Маркер делегирования генерируется сервером (узел имен в нашем случае). Их можно рассматривать как общий секретный ключ между клиентом и сервером. При первом обращении RPC к узлу имен у клиента нет маркера делегирования, поэтому клиент использует Kerberos для выполнения аутентификации. В составе ответа он получает маркер делегирования от узла имен. При последующих вызовах клиент представляет маркер делегирования, который узел имен может проверить (так как маркер был сгенерирован с использованием секретного ключа); таким образом клиент проходит аутентификацию на сервере.

Когда клиент хочет выполнить операцию с блоками HDFS, он использует особую разновидность маркера делегирования — так называемый *маркер блочного доступа*, который передается узлом имен клиенту в ответ на запрос метаданных. Клиент использует маркер блочного доступа при выполнении аутентификации себя на узлах данных. Это возможно только потому, что узел имен предоставляет доступ к своему секретному ключу, использованному для генерирования маркера блочного доступа, узлам данных. Ключ передается по каналу периодических сигналов, и позволяет узлам данных проверять маркеры блочного доступа. Значит, к блоку HDFS может обратиться только клиент, получивший действительный маркер блочного доступа от узла имен. Таким образом закрывается дефект безопасности в Hadoop, когда для получения доступа к блоку было достаточно одного идентификатора блока. Для включения этого режима следует задать `dfs.block.access.token.enable` значение `true`.

В MapReduce ресурсы заданий и метаданные (файлы JAR, входные сплиты и конфигурационные файлы) доступны в HDFS для трекера заданий, а пользовательский код выполняется на трекерах задач и обращается к файлам в HDFS (этот процесс объясняется в разделе «Выполнение заданий MapReduce», с. 256). Маркеры делегирования используются трекерами заданий и задач для обращения к HDFS в процессе выполнения задания. После завершения задания маркеры делегирования становятся недействительными.

Маркеры делегирования автоматически получаются для экземпляра HDFS по умолчанию. Если вашему заданию потребуется обращаться к другим кластерам HDFS, вы можете загрузить маркеры делегирования для этих кластеров, задав свойству `mapreduce.job.hdfs-servers` список HDFS URI, разделенных запятыми.

Другие улучшения в области безопасности

В HDFS и MapReduce реализован более высокий уровень безопасности, защищающий от несанкционированного доступа к ресурсам¹. Наиболее заметные изменения:

- Задачи могут выполняться с использованием учетной записи операционной системы пользователя, отправившего задания (вместо пользователя, запустившего трекер задач). Это означает, что операционная система изолирует выполняемые задачи, не позволяя им отправлять сигналы друг другу (например, чтобы уничтожить задачи другого пользователя), а доступ к локальной информации (например, данным задач) ограничивается благодаря разрешениям локальной файловой системы.

Для включения этой функции следует задать свойству `mapred.task.tracker.task-controller` значение `org.apache.hadoop.mapred.LinuxTaskController`². Кроме того, администраторы должны проследить за тем, чтобы каждый пользователь имел учетную запись на каждом узле в кластере (обычно это делается средствами LDAP).

- При выполнении задач от имени пользователя, отправившего задание, распределенный кэш (см. «Распределенный кэш», с. 375) безопасен. Файлы

¹ На момент написания книги другие проекты — такие, как HBase и Hive — не были интегрированы с этой моделью безопасности.

² LinuxTaskController использует расширение setuid с именем `task-controller`, находящееся в каталоге `bin`. Проследите за тем, чтобы этот двоичный файл принадлежал пользователю `root` и для него был установлен бит `setuid` (команда `chmod +s`).

с неограниченным доступом для чтения, помещаются в общий кэш (небезопасный); остальные файлы помещаются в приватный кэш, в котором их может прочитать только владелец.

- Пользователям разрешается просматривать и изменять только свои задания, но не задания других пользователей. Для включения этой функции следует задать свойству `mapred.acls.enabled` значение `true`. Свойствам конфигурации `mapreduce.job.acl-view-job` и `mapreduce.job.acl-modify-job` могут быть заданы списки пользователей, которым разрешен просмотр или изменение задания (элементы списка разделяются запятыми).
- Тасовка выполняется безопасно, а злоумышленник не может запросить выходные данные отображений другого пользователя. Вместе с тем данные тасовки не шифруются и могут быть перехвачены.
- При правильной настройке кластера злоумышленник не сможет запустить посторонний вторичный узел имен, узел данных или трекер задач, который присоединится к кластеру для возможного повреждения данных, хранимых в кластере. Для этого демоны должны пройти аутентификацию на основном узле, к которому они подключаются.
- Для включения этой функции необходимо сначала настроить Hadoop на использование файла `keytab`, сгенерированного ранее командой `kutil`. Например, для узла данных следует задать свойству `dfs.datanode.keytab.file` имя файла `keytab`, а свойству `dfs.datanode.kerberos.principal` — имя пользователя, использующего узел данных. Наконец, в файле `hadoop-policy.xml` должен быть задан список ACL для протокола `DataNodeProtocol` (используемого узлами данных для взаимодействия с узлом имен), с ограничением `security.datanode.protocol.ac1` именем пользователя узла данных.
- Узел данных может работать на привилегированном порте (с номером ниже 1024), чтобы клиент мог быть более или менее уверен в том, что запуск прошел безопасно.
- Задача может взаимодействовать только со своим родительским трекером задач, не позволяя атакующему перехватить данные MapReduce из задания другого пользователя.

Одной из проблем, еще не решенных в области безопасности Hadoop, является шифрование: ни взаимодействия RPC, ни блочные передачи не шифруются. Блоки HDFS также не хранятся в зашифрованном виде. Эти возможности запланированы в будущих версиях; кроме того, шифрование данных, хранимых в HDFS, может осуществляться в более ранних версиях Hadoop средствами самого приложения (например, написанием кодека сжатия с шифрованием).

Тестирование кластера Hadoop

Правильно ли настроен кластер? Лучший способ получить ответ на этот вопрос — запустить несколько заданий и убедиться в том, что вы получили ожидаемый результат. Также вам помогут контрольные тесты; сравнение полученных результатов с результатами других кластеров поможет вам убедиться в том, что производительность нового кластера примерно соответствует ожиданиям. Затем вы оптимизируете кластер по результатам контрольных тестов, чтобы «выжить» из него всю возможную производительность. Настройка часто производится под контролем установленных систем мониторинга (см. «Мониторинг», с. 450), при помощи которых вы наблюдаете за использованием ресурсов в кластере.

Для получения лучших результатов следует провести контрольные тесты в кластере, в котором другие пользователи не работают. На практике это означает, что тестирование проводится непосредственно перед запуском кластера в эксплуатацию и началом его активного использования. Когда пользователи начнут планировать периодическое выполнение заданий в кластере, вам вряд ли удастся найти свободное время (если только заранее не согласовать его с пользователем). Лучше выполнить контрольные тесты до того, как это произойдет.

Опыт показал, что большинство аппаратных сбоев в новых системах является сбоями жестких дисков. Выполняя контрольные тесты с интенсивным вводом/выводом (такие, как описанные ниже), вы можете «зафиксировать» состояние кластера, до того как произойдет сбой.

Контрольные тесты Hadoop

В поставку Hadoop включены контрольные тесты, которые запускаются очень легко и с минимальными затратами на настройку. Контрольные тесты упакованы в файл JAR. Чтобы получить их список с описаниями, запустите файл JAR без аргументов:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*~test.jar.
```

Как правило, при запуске без аргументов контрольные тесты выводят инструкции по использованию. Пример:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*~test.jar TestDFSIO
TestDFSIO 0.0.4
Usage: TestDFSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile
resultFileName] [-bufferSize Bytes]
```

TestDFSIO

Пакет TestDFSIO тестирует производительность ввода/вывода HDFS. Для организации параллельного чтения и записи файлов используется задание MapReduce. Каждый файл читается или записывается отдельной задачей отображения, а выходные данные отображения используются для сбора статистики, относящейся к только что обработанному файлу. На основании статистики, накапливаемой в свертках, строится сводный отчет.

Следующая команда записывает 10 файлов по 1000 Мбайт каждый:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-*-test.jar TestDFSIO -write -nrFiles 10  
-fileSize 1000
```

В конце выполнения результаты выводятся на консоль и записываются в локальный файл (данные присоединяются к существующему файлу, так что при повторном запуске контрольных тестов старые результаты не теряются):

```
% cat TestDFSIO_results.log  
----- TestDFSIO ----- : write  
Date & time: Sun Apr 12 07:14:09 EDT 2009  
Number of files: 10  
Total MBytes processed: 10000  
Throughput mb/sec: 7.796340865378244  
Average IO rate mb/sec: 7.8862199783325195  
IO rate std deviation: 0.9101254683525547  
Test exec time sec: 163.387
```

Файлы по умолчанию записываются в подкаталог *io_data* каталога */benchmarks/TestDFSIO* (местонахождение каталога можно изменить при помощи системного свойства *test.build.data*).

Для запуска контрольного теста чтения используется аргумент *-read*. Файлы должны уже существовать (то есть быть записаны TestDFSIO с параметром *-write*):

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-*-test.jar TestDFSIO -read -nrFiles 10  
-fileSize 1000
```

Результаты реального запуска:

```
----- TestDFSIO ----- : read  
Date & time: Sun Apr 12 07:24:28 EDT 2009  
Number of files: 10  
Total MBytes processed: 10000  
Throughput mb/sec: 80.25553361904304  
Average IO rate mb/sec: 98.6801528930664  
IO rate std deviation: 36.63507598174921  
Test exec time sec: 47.624
```

Чтобы после завершения контрольных тестов удалить все сгенерированные файлы из HDFS, используйте аргумент `-clean`:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*~test.jar TestDFSIO -clean.
```

Сортировка

В поставку Hadoop включена программа MapReduce, выполняющая частичную сортировку своих входных данных. Она очень удобна для контрольного тестирования всей системы MapReduce, так как в процессе тасовки происходит передача всего входного набора данных. Выполнение программы состоит из трех шагов: генерирование случайных данных, выполнение сортировки и проверка результатов.

Для генерирования случайных данных используется объект `RandomWriter`. Он запускает задание MapReduce с 10 отображениями на узел, и каждое отображение генерирует приблизительно 1 Гбайт случайных двоичных данных с ключами и значениями разных размеров. При желании эти параметры можно изменить, задав нужные значения свойствам `test.randomwriter.maps_per_host` и `test.randomwriter.bytes_per_map`. Также имеются настройки для диапазонов размеров ключей и значений; за подробностями обращайтесь к документации `RandomWriter`.

Пример вызова `RandomWriter` (из файла JAR примера, а не теста) для записи выходных данных в каталог с именем `random-data`:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*~examples.jar randomwriter random-data.
```

Затем можно запустить программу сортировки:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*~examples.jar sort random-data sorted-data.
```

Нас интересует общее время выполнения сортировки, однако будет поучительно понаблюдать за ходом выполнения задания через веб-интерфейс (<http://jobtracker-host:50030/>). Вы получите представление о том, сколько времени занимает каждая фаза задания. Также полезно поэкспериментировать с параметрами, упомянутыми в разделе «Оптимизация задания», с. 243.

И напоследок следует убедиться, что данные в `sorted-data` действительно правильно отсортированы:

```
% hadoop jar $HADOOP_INSTALL/hadoop-*~test.jar testmapredsort -sortInput random-data \
-sortOutput sorted-data
```

Команда запускает программу `SortValidator`, которая выполняет серию проверок с несортированными и отсортированными данными, и проверяет правильность сортировки. Результат выводится на консоль в конце выполнения:

```
SUCCESS! Validated the MapReduce framework's 'sort' successfully.
```

Другие контрольные тесты

Существует много других контрольных тестов Hadoop, но наиболее часто применяемыми являются следующие:

- MRBench (команда *mrbench*) выполняет небольшое задание заданное число раз. Этот тест хорошо дополняет тест сортировки, так как он проверяет скорость отклика малых заданий.
- NN Bench (команда *nnbench*) используется для нагрузочного тестирования оборудования узлов имен.
- Gridmix представляет собой набор контрольных тестов, моделирующих реальную нагрузку в кластере посредством имитации различных схем выборки данных, встречающихся на практике. За дополнительной информацией о запуске Gridmix обращайтесь к документации дистрибутива и к публикации в блоге по адресу [http://developer.yahoo.net/blogs/hadoop/2010/04/gridmix3_emulating_production.html¹](http://developer.yahoo.net/blogs/hadoop/2010/04/gridmix3_emulating_production.html).

Пользовательские задания

В процесс оптимизации желательно включить несколько заданий, типичных для ваших пользователей, чтобы ваш кластер был оптимизирован и для таких заданий, а не только для стандартных контрольных тестов. Если это ваш первый кластер Hadoop и у вас еще нет примеров пользовательских заданий, Gridmix станет хорошей заменой.

При выполнении ваших собственных заданий как контрольных тестов следует выбрать набор данных и использовать его каждый раз, когда вы выполняете контрольные тесты для сравнения результатов. При создании нового или обновлении существующего кластера вы сможете использовать тот же набор данных для сравнения производительности с предыдущими запусками.

Hadoop в облаке

Хотя многие организации строят кластеры Hadoop на собственном оборудовании, также часто встречается вариант с запуском приложений Hadoop в облаке на арендованном оборудовании или в виде сервиса. Например, Cloudera предоставляет инструменты для выполнения Hadoop в открытом или приватном облаке,

¹ PigMix – похожий набор контрольных тестов для Pig – доступен по адресу <https://cwiki.apache.org/confluence/display/PIG/PigMix>.

а Amazon предоставляет облачный сервис Hadoop, который называется Elastic MapReduce.

В этом разделе мы рассмотрим выполнение приложений Hadoop в Amazon EC2. Это отличный способ опробовать собственный кластер Hadoop с минимальными затратами.

Apache Whirr

Amazon Elastic Compute Cloud (EC2) — вычислительный сервис, позволяющий клиентам арендовать компьютеры (экземпляры — instances), на которых они могут запускать свои приложения. Клиент может запускать и завершать экземпляры по мере надобности, с почасовой оплатой за работу активных экземпляров.

Проект Apache Whirr (<http://whirr.apache.org/>) предоставляет Java API и набор сценариев, упрощающих запуск приложений Hadoop в EC2 и других платформах поставщиков облачного сервиса¹. Сценарии позволяют выполнять такие операции, как запуск или остановка кластера или получение списка работающих экземпляров в кластере.

Выполнение приложений Hadoop в EC2 особенно хорошо подходит для некоторых рабочих процессов. Например, если данные хранятся в Amazon S3, вы можете запустить кластер в EC2 и выполнять задания MapReduce, которые читают данные из S3 и записывают вывод обратно в S3 перед закрытием кластера. Если ваши кластеры имеют более долгий срок жизни, скопируйте данные S3 в систему HDFS, работающую на базе EC2, — это повысит эффективность их обработки, так как HDFS, в отличие от S3, может пользоваться локальностью данных (хранилище S3 не совмещено с узлами EC2).

Настройка

Начните с установки Whirr. Загрузите tar-файл с новейшей версией и распакуйте его на машину, с которой будет запускаться кластер:

```
% tar xzf whirr-x.y.z.tar.gz.
```

Для взаимодействия с машинами, работающими в облаке, Whirr использует SSH; соответственно, будет полезно сгенерировать пару ключей SSH, предназначенную исключительно для Whirr. Следующая команда создает пару ключей RSA

¹ В подкаталоге src/contrib/ec2 дистрибутива Hadoop находятся сценарии bash, но вместо них рекомендуется использовать Whirr.

с пустым паролем и сохраняет их в файле *id_rsa_whirr* в каталоге *.ssh* текущего пользователя:

```
% ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa_whirr.
```



Не путайте пару ключей Whirr SSH с сертификатами, закрытыми ключами или парами ключей SSH, связанными с вашей учетной записью Amazon Web Services. Система Whirr проектировалась для работы с разными поставщиками облачного сервера, и ей нужен доступ к открытому и закрытому ключу SSH беспарольной пары ключей, прочитанной из локальной файловой системы. На практике самым простым решением оказывается генерирование новой пары ключей для Whirr.

Теперь необходимо передать Whirr удостоверения нашего поставщика облачного сервиса. Их можно экспортовать в виде переменных окружения, хотя также можно указать их в командной строке или в конфигурационном файле:

```
% export AWS_ACCESS_KEY_ID='...'
% export AWS_SECRET_ACCESS_KEY='...'
```

Запуск кластера

Все готово к запуску кластера. В поставку Whirr входят файлы наборов параметров для запуска типовых конфигураций сервиса; мы используем такой файл для запуска Hadoop на платформе EC2:

```
% bin/whirr launch-cluster --config recipes/hadoop-ec2.properties \
--private-key-file ~/.ssh/id_rsa_whirr
```

Команда *launch-cluster* резервирует экземпляры в облаке и запускает на них сервисы, после чего возвращает управление пользователю.

Конфигурация

Прежде чем браться за использование кластера, рассмотрим конфигурацию Whirr более подробно. Параметры конфигурации передаются командам Whirr в конфигурационных файлах, задаваемых параметром *--config*, или по отдельности в аргументах командной строки (как, например, аргумент *--private-key-file*, который мы использовали для передачи местонахождения файла закрытого ключа SSH).

Файл набора параметров представляет собой файл свойств Java, определяющий ряд свойств Whirr. Рассмотрим важнейшие свойства из *hadoop-ec2.properties*, начиная с двух свойств, определяющих кластер и выполняемых в нем службы:

```
whirr.cluster-name=hadoop
whirr.instance-templates=1 hadoop-namenode+hadoop-jobtracker,5 hadoop-datanode+
hadoop-tasktracker
```

У каждого кластера есть имя, заданное свойством `whirr.cluster-name`; имя используется для идентификации кластера, чтобы вы могли выполнять с ним операции (например, вывести список всех работающих экземпляров или завершить работу кластера). Имя кластера должно быть уникальным в пределах учетной записи облачного сервиса, под которой работает кластер.

Свойство `whirr.instance-templates` определяет сервисы, работающие в кластере. Шаблон экземпляра задает количество и набор ролей, работающих в каждом экземпляре этого типа. Таким образом, в нашем примере имеется один экземпляр, работающий в роли `hadoop-namenode` и в роли `hadoop-jobtracker`, и 5 экземпляров, работающих в роли `hadoop-datanode` и `hadoop-tasktracker`. При помощи свойства `whirr.instance-templates` можно определить точную структуру кластера. Помимо Hadoop, существует много других сервисов; чтобы получить информацию о них, выполните `bin/whirr` без аргументов.

Следующая группа свойств задает облачные удостоверения:

```
whirr.provider=aws-ec2
whirr.identity=${env:AWS_ACCESS_KEY_ID}
whirr.credential=${env:AWS_SECRET_ACCESS_KEY}
```

Свойство `whirr.provider` определяет поставщика облачного сервиса, в нашем случае EC2 (другие поддерживаемые поставщики перечислены в документации Whirr). Свойства `whirr.identity` и `whirr.credential` содержат удостоверения, специфические для облака, — упрощенно говоря, имя пользователя и пароль, хотя терминология зависит от поставщика.

Три последних параметра предназначены для управления оборудованием кластера (возможности экземпляров — память, диск, процессор, скорость сети и т. д.), образом машины (операционная система) и географическим местонахождением (центр обработки данных). Все они зависят от поставщика сервиса, но, если их опустить, Whirr постарается подобрать хорошие значения по умолчанию.

```
whirr.hardware-id=c1.xlarge
whirr.image-id=us-east-1/ami-da0cf8b3
whirr.location-id=us-east-1
```

Свойства в файле имеют префикс `whirr`, но при передаче их в аргументах командной строки префикс не указывается. Таким образом, например, для задания имени кластера можно включить в командную строку параметр `--cluster-name hadoop`, и он заменит любое значение, указанное в файле свойств. А чтобы задать файл закрытого ключа в файле свойств, включите в него строку вида

```
whirr.private-key-file=/user/tom/.ssh/id_rsa_whirr
```

Также имеются свойства для указания версии Hadoop, которая должна работать в кластере, и назначения свойств конфигурации Hadoop в кластере (подробности содержатся в файле набора параметров).

Запуск прокси-сервера

В процессе использования кластера сетевой трафик от клиентов должен проходить через туннель SSH, который создается следующей командой:

```
% . ~/whirr/hadoop/hadoop-proxy.sh.
```

Прокси-сервер работает все время, пока работает кластер. Завершив работу кластера, остановите прокси-сервер клавишами Ctrl+C.

Выполнение задания MapReduce

Задания MapReduce можно запускать либо из кластера, либо с внешней машины. Здесь мы покажем, как запустить задание с машины, с которой был запущен кластер. Обратите внимание: версия Hadoop, установленная локально, должна совпадать с версией, работающей в кластере.

При запуске кластера файлы конфигурации Hadoop были созданы в каталоге `~/whirr/hadoop`. Мы можем использовать их для подключения к кластеру, задав переменной окружения `HADOOP_CONF_DIR` следующее значение:

```
% export HADOOP_CONF_DIR=~/whirr/hadoop.
```

Файловая система кластера пуста, и перед запуском задания необходимо заполнить ее данными. Параллельное копирование из S3 (об использовании файловых систем S3 в Hadoop рассказано в разделе «Файловые системы Hadoop», с. 90) программой Hadoop `distcp` является эффективным способом копирования данных в HDFS:

```
% hadoop distcp \
-Dfs.s3n.awsAccessKeyId='...' \
-Dfs.s3n.awsSecretAccessKey='...' \
s3n://hadoopbook/ncdc/all input/ncdc/all
```

После того, как данные будут скопированы, задание запускается обычным способом:

```
% hadoop jar hadoop-examples.jar MaxTemperatureWithCombiner \
/user/$USER/input/ncdc/all /user/$USER/output
```

Также можно указать, что выходные данные находятся в S3:

```
% hadoop jar hadoop-examples.jar MaxTemperatureWithCombiner \
/user/$USER/input/ncdc/all s3n://mybucket/output
```

За ходом выполнения задания можно проследить через веб-интерфейс трекера заданий по адресу *http://master_host:50030/*. Для обращения к веб-страницам на рабочих узлах необходимо настроить файл PAC (Proxy Auto-Config) в вашем браузере. О том, как это делается, рассказано в документации Whirr.

Завершение работы кластера

Чтобы завершить работу кластера, выполните команду *destroy-cluster*.

```
% bin/whirr destroy-cluster --config recipes/hadoop-ec2.properties.
```

Команда завершает все работающие экземпляры и удаляет все данные, хранящиеся в кластере.

10 Администрирование Hadoop

Предыдущая глава была посвящена настройке кластера Hadoop. В этой главе будут рассмотрены операции, обеспечивающие бесперебойную работу кластера.

HDFS

Дисковые структуры данных

Администратор должен иметь базовое представление о том, как организованы дисковые структуры данных основных компонентов HDFS — узла имен, вторичного узла имен, узлов данных. Если вы будете знать, для чего нужен тот или иной файл, вы быстрее проведете диагностику или заметите, что в системе возникли какие-то неполадки.

Структура каталогов узла имен

Отформатированный узел имен создает следующую структуру каталогов:

```
 ${dfs.name.dir}/  
 └── current/  
     ├── VERSION  
     ├── edits  
     ├── fsimage  
     └── fstime
```

Как говорилось в главе 9, свойство `dfs.name.dir` включает в себя список каталогов, содержащих зеркальные копии одного содержимого. Этот механизм обеспечивает необходимую гибкость, особенно в том случае, если один из каталогов смонтирован из NFS, как рекомендуется делать.

Файл `VERSION` представляет собой файл свойств Java с информацией о версии HDFS. Содержимое типичного файла выглядит так:

```
#Tue Mar 10 19:21:36 GMT 2009
namespaceID=134368441
cTime=0
storageType=NAME_NODE
layoutVersion=-18
```

`layoutVersion` — отрицательное целое число, определяющее версию дисковых структур данных HDFS. Номер версии не имеет отношения к версии дистрибутива Hadoop. При изменении версии дисковых структур данных номер версии уменьшается (например, после `-18` идет версия `-19`). Когда это происходит, HDFS необходимо обновить, потому что новый узел имен (или узел данных) не будет работать со старой версией дисковых структур. Обновление HDFS рассматривается в разделе «Обновления», с. 463.

Свойство `namespaceID` — уникальный идентификатор файловой системы, созданный при ее исходном форматировании. Узел имен использует его для идентификации новых узлов данных, поскольку узлам данных значение `namespaceID` остается неизвестным до регистрации на узле имен.

Свойство `cTime` содержит время создания дискового пространства узла имен. Для только что отформатированной области оно всегда равно нулю, но при обновлении файловой системы значение обновляется временной меткой.

Свойство `storageType` указывает, что каталог содержит структуры данных узла имен.

Также в каталоге узла имен находятся файлы `edits`, `fsimage` и `fstime`. Все они являются двоичными файлами, которые используют объекты Hadoop `Writable` в качестве формата сериализации (см. «Сериализация», с. 140). Чтобы понять, для чего нужны эти данные, необходимо получше разобраться в том, как работает узел имен.

Образ файловой системы и журнал изменений

Когда клиент файловой системы выполняет операцию записи (например, создает или перемещает файл), операция сначала фиксируется в журнале изменений. Журнал также содержит внутреннее представление метаданных файловой системы, обновляемое после модификации журнала изменений. Метаданные, хранящиеся в памяти, используются для обслуживания запросов чтения.

Журнал изменений синхронизируется после каждой записи, перед возвращением клиенту кода успешного выполнения. Для узлов имен, записывающих данные в несколько каталогов, запись должна синхронизироваться с каждой копией. Это предотвращает возможные потери операций из-за сбоев на машинах.

В файле *fsimage* хранится контрольная точка метаданных файловой системы. Тем не менее содержимое файла не обновляется при каждой операции записи в файловой системе, потому что запись файла *fsimage*, увеличивающегося до гигабайтных размеров, будет происходить очень медленно. Впрочем, это не вредит надежности, потому что в случае сбоя узла имен последнее состояние метаданных восстанавливается посредством загрузки *fsimage* с диска в память и последующим применением каждой операции из журнала изменений. Собственно, именно это делает узел имен в начале работы (см. «Безопасный режим», с. 441).



Файл *fsimage* содержит сериализованную форму всех индексных узлов файлов и каталогов в файловой системе. Индексный узел (*inode*) представляет собой внутреннее представление метаданных файла или каталога; в нем хранится такая информация, как уровень репликации файла, время изменения и обращения, разрешения доступа, размер блока и блоки, из которых строится файл. Для каталогов хранятся время изменения, разрешения и метаданные квоты.

Файл *fsimage* не содержит информацию о том, на каких узлах данных хранятся блоки. Эти данные хранятся в памяти; для их построения у узлов данных при присоединении к кластеру запрашиваются списки блоков, которые затем периодически обновляются для сохранения актуальности.

Как упоминалось ранее, файл *edits* может расти без ограничений. Хотя такое состояние дел не повлияет на систему во время работы узла имен, в случае перезапуска узла имен применение каждой операции из (очень длинного) журнала изменений займет много времени. В это время файловая система будет недоступна, что обычно нежелательно.

Проблема решается запуском вторичного узла имен, задача которого — создание контрольных точек метаданных файловой системы, хранящихся в памяти первичного (основного) узла¹. Процесс построения контрольных точек проходит по следующей схеме (рис. 10.1):

¹ Начиная с Hadoop версии 0.22.0, можно запускать узел имен с параметром *-checkpoint*, чтобы он запускал процесс построения контрольных точек для другого (первичного) узла имен. С функциональной точки зрения эта возможность эквивалентна запуску вторичного узла имен, но на момент написания книги она не имела никаких преимуществ перед ним. При запуске в условиях высокой доступности (см. «Высокая доступность HDFS», с. 85) контрольные точки могут создаваться резервным узлом.

1. Вторичный узел приказывает первичному начать новый файл изменений.
2. Вторичный узел получает от первичного файлы *fsimage* и *edits* (с использованием HTTP GET).

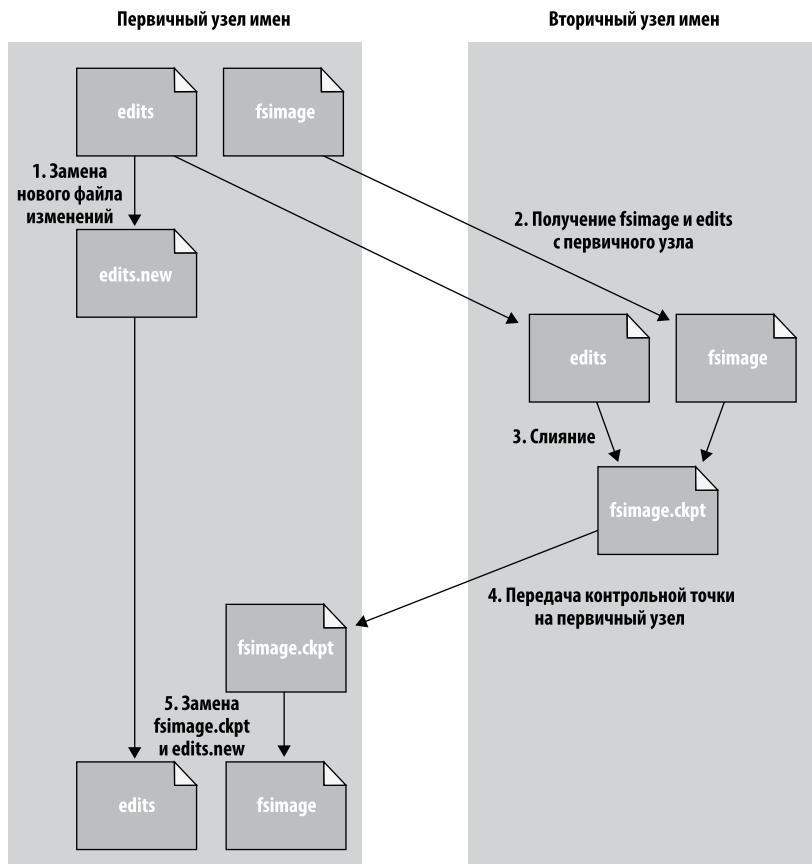


Рис. 10.1. Процесс создания контрольных точек

3. Вторичный узел загружает *fsimage* в память, применяет каждую операцию из *edits*, после чего создает новый консолидированный файл *fsimage*.
4. Вторичный узел отправляет первичному новую версию *fsimage* (с использованием HTTP POST).
5. Первичный узел заменяет старую версию *fsimage* новой, полученной от вторичного узла, а старый файл *edits* — новой, начатой на шаге 1. Также он обновляет файл *fstime* временем создания контрольной точки.

В конце процесса первичный узел имен содержит обновленную версию *fsimage* и более короткий файл *edits* (не обязательно пустой — какие-то изменения могли быть получены во время создания контрольной точки). Администратор может запустить этот процесс вручную командой *hadoop dfsadmin -saveNamespace*, если узел имен находится в безопасном режиме.

Эта процедура наглядно объясняет, почему вторичный узел предъявляет такие же требования к памяти, что и первичный узел (потому что он загружает *fsimage* в память); именно по этой причине в больших кластерах под вторичный узел выделяется специальная машина.

Расписание создания контрольных точек определяется двумя параметрами конфигурации. Вторичный узел имен создает контрольные точки каждый час (`fs.checkpoint.period` в секундах) или ранее, если объем журнала изменений достигает 64 Мбайт (`fs.checkpoint.size` в байтах), с проверкой каждые пять минут.

Структура каталогов вторичного узла имен

У процесса создания контрольных точек имеется полезный побочный эффект: в конце этого процесса на вторичном узле имеется контрольная точка, которая находится в подкаталоге *previous.checkpoint*. Она может использоваться для создания (запоздалых) резервных копий метаданных узла имен:

```
 ${fs.checkpoint.dir}/
   +-- current/
   |   +-- VERSION
   |   +-- edits
   |   +-- fsimage
   |   +-- fstime
   +-- previous.checkpoint/
       +-- VERSION
       +-- edits
       +-- fsimage
       +-- fstime
```

Структура этого каталога и каталога *current* вторичного узла идентична структуре каталогов узла имен. Это сделано сознательно, так как в случае общего отказа узла имен (при отсутствии резервных копий, пригодных для восстановления, даже из NFS) появляется возможность восстановления из вторичного узла имен. Это может быть сделано либо копированием соответствующего каталога на новый узел имен, либо (в том случае, если вторичный узел становится новым первичным узлом) включением параметра *-importCheckpoint* при запуске демона узла имен. Параметр *-importCheckpoint* загружает метаданные узла имен из последней контрольной точки в каталоге, определяемом свойством `fs.checkpoint.dir`, но

только при отсутствии метаданных в каталоге `dfs.name.dir`, чтобы предотвратить возможную перезапись драгоценных метаданных.

Структура каталогов узла данных

Узлы данных, в отличие от узлов имен, не нуждаются в форматировании, потому что они автоматически создают свои каталоги хранения данных при запуске. Основные файлы и каталоги узла данных:

```
 ${dfs.data.dir}/  
   └── current/  
       ├── VERSION  
       ├── blk_<id_1>  
       ├── blk_<id_1>.meta  
       ├── blk_<id_2>  
       ├── blk_<id_2>.meta  
       ├── ...  
       ├── blk_<id_64>  
       ├── blk_<id_64>.meta  
       ├── subdir0/  
       ├── subdir1/  
       ├── ...  
       └── subdir63/
```

Файл `VERSION` узла данных очень похож на одноименный файл узла имен:

```
#Tue Mar 10 21:32:31 GMT 2009  
namespaceID=134368441  
storageID=DS-547717739-172.16.85.1-50010-1236720751627  
cTime=0  
storageType=DATA_NODE  
layoutVersion=-18
```

Параметры `namespaceID`, `cTime` и `layoutVersion` содержат те же значения, что и в файле узла имен (собственно, значение `namespaceID` загружается с узла имен при первом подключении узла данных). Параметр `storageID` уникален для узла данных (его значение одинаково во всех каталогах хранения данных), и он используется узлом имен для однозначной идентификации узла данных. Параметр `storageType` идентифицирует каталог узла данных, предназначенный для хранения данных.

Также в каталоге `current` узла данных находятся файлы с префиксом `blk_`. Они делятся на два типа: собственно блоки HDFS (состоящие непосредственно из байтов файла) и метаданные блока (суффикс `.meta`). Файл блока состоит из непосредственных байтов хранимой части файла; файл метаданных состоит из заголовка

с версией и информацией типа, за которым следует серия контрольных сумм для секций блока.

Когда количество блоков в каталоге увеличивается до определенного порога, узел данных создает новый подкаталог, в котором размещаются новые блоки и сопровождающие метаданные. Новый подкаталог создается каждый раз, когда количество блоков в каталоге достигает 64 (свойство конфигурации `dfs.datanode.numblocks`). В итоге формируется дерево с высоким коэффициентом ветвления, так что даже для систем с очень большим количеством блоков глубина каталогов составит всего несколько уровней. Благодаря этой мере каждый каталог узла данных будет содержать разумное количество файлов, и это предотвратит проблемы, возникающие во многих операционных системах при нахождении в одном каталоге большого количества файлов (десятки или сотни тысяч).

Если свойство конфигурации `dfs.data.dir` задает несколько каталогов на разных дисках, блоки записываются в каталоги по принципу циклического чередования. Обратите внимание: блоки не реплицируются на разных дисках одного узла данных; репликация работает между узлами данных.

Безопасный режим

При запуске узел имен прежде всего загружает в память свой файл образа (*fsimage*) и применяет изменения из журнала изменений (*edits*). После того, как в памяти будет сконструирован целостный образ метаданных файловой системы, он создает новый файл *fsimage* (фактически создавая контрольную точку самостоятельно, без обращения к вторичному узлу имен) и пустой журнал изменений. Только на этой стадии узел имен начинает прослушивать запросы RPC и HTTP. Однако узел имен работает в безопасном режиме, то есть предоставляет клиентам доступ к файловой системе только для чтения.



Строго говоря, в безопасном режиме гарантирована только работоспособность операций, обращающихся к метаданным файловой системы (например, получения списка содержимого каталога). Чтение файла будет работать только в том случае, если блоки доступны в текущем наборе узлов данных в кластере, а модификации файлов (чтение, запись или удаление) всегда будут завершаться неудачей.

Вспомните, что узел имен не сохраняет информацию о местонахождении блоков в системе; эта информация хранится на узлах данных в форме списков блоков, которые на них хранятся. В ходе нормального функционирования системы у узла

имен имеется карта местонахождения блоков в памяти. Безопасный режим необходим для того, чтобы у узлов данных было время зарегистрироваться на узле имен со своими списками блоков, чтобы у узла имен было достаточно информации для эффективного управления файловой системой. Если узел имен не дождется регистрации достаточного количества узлов данных, он начнет процесс репликации блоков на новых узлах данных, что в большинстве случаев будет лишним (потому что ему достаточно дождаться регистрации дополнительных узлов данных) и создаст лишнюю нагрузку для ресурсов кластера. В безопасном режиме узел имен не выдает узлам данных инструкции репликации или удаления блоков.

Выход из безопасного режима происходит через 30 секунд после достижения условия минимальной репликации. Условие минимальной репликации означает, что 99,9% блоков во всей файловой системе соответствует минимальному уровню репликации (по умолчанию равному 1, но задаваемому свойством `dfs.replication.min`; см. табл. 10.1).

В только что отформатированном кластере HDFS узел имен не переходит в безопасный режим, так как в системе еще нет блоков.

Таблица 10.1. Свойства безопасного режима

Имя свойства	Тип	Значение по умолчанию	Описание
<code>dfs.replication.min</code>	int	1	Минимальное количество реплик, которое должно быть записано для того, чтобы операция записи считалась успешной
<code>dfs.safemode.threshold.pct</code>	float	0.999	Доля блоков системы, которая должна удовлетворять требованиям минимального уровня репликации, определяемым <code>dfs.replication.min</code> , при достижении которой узел имен выходит из безопасного режима. Если свойство равно 0, узел имен не стартует в безопасном режиме. Значение больше 1 означает, что узел имен никогда не выйдет из безопасного режима
<code>dfs.safemode.extension</code>	int	30 000	Продолжительность ожидания (в миллисекундах) после выполнения условия минимальной репликации, определяемого свойством <code>dfs.safemode.threshold.pct</code> . Для малых кластеров (десятки узлов) может быть равно 0

Вход и выход из безопасного режима

Чтобы проверить, находится ли узел имен в безопасном режиме, используйте команду `dfsadmin`:

```
% hadoop dfsadmin -safemode get  
Safe mode is ON
```

Информация о нахождении узла имен в безопасном режиме также выводится на первой странице веб-интерфейса HDFS.

Иногда перед выполнением команды (особенно в сценариях) необходимо дождаться, пока узел имен выйдет из безопасного режима. Задача решается параметром `wait`:

```
hadoop dfsadmin -safemode wait  
# Команда чтения или записи файла
```

Администратор может заставить узел имен войти в безопасный режим или выйти из него в любой момент. Иногда это необходимо сделать при выполнении операций сопровождения на кластере или после обновления кластера, чтобы проверить доступность данных. Для входа в безопасный режим используется следующая команда:

```
% hadoop dfsadmin -safemode enter  
Safe mode is ON
```

Команду можно использовать, пока узел все еще находится в безопасном режиме во время запуска; тем самым вы предотвратите его выход из безопасного режима. Другой способ обеспечить неограниченно долгое пребывание узла имен в безопасном режиме — задать свойству `dfs.safemode.threshold.pct` значение больше 1.

Узел имен выводится из безопасного режима командой

```
% hadoop dfsadmin -safemode leave  
Safe mode is OFF
```

Журналы аудита

HDFS может регистрировать в журнале все запросы на обращение к файловой системе — в некоторых организациях это является обязательным требованием в целях аудита. Ведение журнала аудита реализуется использованием средств `log4j` на уровне `INFO`. В конфигурации по умолчанию эта функция отключена, так как в `log4j.properties` установлен порог `WARN`:

```
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=WARNING
```

Ведение журнала аудита включается заменой `WARN` на `INFO`; в результате для каждого события HDFS в журнал узла имен будет добавляться новая строка. Пример для запроса на получение списка содержимого для `/user/tom`:

```
2009-03-13 07:11:22,982 INFO
    org.apache.hadoop.hdfs.server.namenode.FSNamesystem.
audit: ugi=tom,staff,admin ip=/127.0.0.1 cmd=listStatus
    src=/user/tom dst=null
perm=null
```

Желательно настроить `log4j` так, чтобы журнал аудита записывался в отдельный файл и не смешивался с другими журнальными записями узла имен. Пример приведен в Hadoop вики по адресу <http://wiki.apache.org/hadoop/HowToConfigure>.

Инструменты

dfsadmin

Программа `dfsadmin` представляет собой универсальный инструмент для получения информации о состоянии HDFS, а также для выполнения административных операций с HDFS. Она выполняется командой `hadoop dfsadmin` и требует привилегий суперпользователя.

Некоторые команды `dfsadmin` описаны в табл. 10.2. Для получения дополнительной информации используйте команду `-help`.

Таблица 10.2. Команды `dfsadmin`

Команда	Описание
<code>-help</code>	Выводит справку по заданной команде (или всем командам, если команда не указана)
<code>-report</code>	Выводит статистику файловой системы (аналогичную выводимой в веб-интерфейсе) и информацию о подключенных узлах данных
<code>-metasave</code>	Выводит в файл, находящийся в каталоге журналов Hadoop, информацию о реплицируемых или удаляемых блоках, а также список подключенных узлов данных
<code>-safemode</code>	Изменяет или выводит информацию о состоянии безопасного режима. См. «Безопасный режим», с. 441
<code>-saveNamespace</code>	Сохраняет текущий образ файловой системы, хранимый в памяти, в новом файле <code>fsimage</code> и очищает файл <code>edits</code> . Операция может выполняться только в безопасном режиме

Команда	Описание
-refreshNodes	Обновляет набор узлов данных, которым разрешено подключение к узлу имен. См. «Включение и исключение узлов», с. 459
-upgradeProgress	Получает информацию о прогрессе обновления HDFS или форсирует его продолжение. См. «Обновления», с. 463
-finalizeUpgrade	Удаляет предыдущую версию каталогов хранения данных узла имен и узлов данных. Используется после применения обновления и начала работы кластера с новой версией. См. «Обновления», с. 463
-setQuota	Задает квоты каталогов, устанавливающие ограничения на количество имен (файлов или каталогов) в дереве каталогов. Квоты каталогов предотвращают создание пользователями многочисленных мелких файлов — мера, направленная на экономию памяти узла имен (вспомните, что информация о каждом файле, каталоге и блоке файловой системы хранится в памяти)
-clrQuota	Сбрасывает заданные квоты каталогов
-setSpaceQuota	Задает квоты на выделяемое пространство для каталогов, устанавливающие ограничения на размер файлов, которые могут храниться в дереве каталогов
-clrSpaceQuota	Сбрасывает заданные квоты на выделяемое пространство
-refreshServiceAcl	Обновляет файл политики авторизации уровня сервиса на узле имен

Проверка файловой системы (*fsck*)

Для проверки состояния файлов HDFS в Hadoop существует программа *fsck*. Программа ищет блоки, отсутствующие на всех узлах данных, а также блоки с недостаточным или завышенным количеством реплик. Пример проверки всей файловой системы для небольшого кластера:

```
% hadoop fsck /
.....Status: HEALTHY
Total size: 511799225 B
Total dirs: 10
Total files: 22
Total blocks (validated): 22 (avg. block size 23263601 B)
Minimally replicated blocks: 22 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
```

продолжение ↗

```
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1
```

The filesystem under path '/' is HEALTHY

Программа *fsck* осуществляет рекурсивный обход пространства имен файловой системы, начиная с заданного пути (в нашем примере это корень файловой системы), и проверяет найденные файлы. Для каждого проверяемого файла выводится точка. Чтобы проверить файл, *fsck* получает метаданные блоков файла и ищет проблемы или расхождения. Обратите внимание: *fsck* получает всю информацию от узла имен; программа не связывается с узлами данных для фактического получения данных блоков.

Большая часть вывода *fsck* не требует разъяснений. Некоторые условия, которые ищет программа:

Избыточная репликация

Блоки, превышающие целевой коэффициент репликации для файла, к которому они принадлежат. Обычно избыточная репликация не является ошибкой, и HDFS автоматически удаляет лишние реплики.

Недостаточная репликация

Блоки, не достигающие целевого коэффициента репликации для файла, к которому они принадлежат. HDFS автоматически создает новые реплики таких блоков до тех пор, пока не будет достигнут нужный коэффициент репликации. Информацию о реплицируемых блоках (или ожидающих репликации) можно получить командой *hadoop dfsadmin -metasave*.

Неверное размещение реплик

Блоки, не удовлетворяющие политике размещения реплик (см. «Размещение реплик», с. 115). Например, если при коэффициенте репликации 3 в многосегментном кластере все три реплики блока находятся в одном сегменте, репликация считается некорректной, потому что реплики должны быть распределены минимум по двум сегментам. HDFS автоматически заново реплицирует такие блоки, чтобы они соответствовали требованиям политики размещения.

Поврежденные блоки

Блоки, у которых все реплики повреждены. Блоки, имеющие хотя бы одну неповрежденную реплику, не считаются поврежденными; узел имен

реплицирует неповрежденную реплику до достижения целевого коэффициента репликации.

Отсутствие реплик

Блоки, для которых в кластере не существует реплик.

Основные проблемы возникают с поврежденными или отсутствующими блоками, так как это означает потерю данных. По умолчанию *fsck* оставляет файлы с поврежденными или отсутствующими блоками, но вы можете приказать выполнить одну из следующих операций:

- Переместить файлы в каталог */lost+found* в HDFS (параметр *-move*). Файлы разбиваются на цепочки смежных блоков, чтобы упростить возможные попытки восстановления данных.
- Удалить файлы (параметр *-delete*). Удаленные файлы уже не удастся восстановить.

Поиск блоков файла

Программа *fsck* предоставляет простые средства поиска блоков, принадлежащих заданному файлу. Пример:

```
% hadoop fsck /user/tom/part-00007 -files -blocks -racks
/usr/tom/part-00007 25582428 bytes, 1 block(s): OK
0. blk_-3724870485760122836_1035 len=25582428
    repl=3 [/default-rack/10.251.43.2:50010,
/default-rack/10.251.27.178:50010, /default-rack/10.251.123.163:50010]
```

В выходных данных видно, что файл */user/tom/part-00007* состоит из одного блока, а также приводятся узлы данных, на которых находятся блоки. В команде использованы следующие параметры *fsck*:

- Параметр *-files* выводит строку с именем файла, размером, количеством блоков и их состоянием (наличие отсутствующих блоков).
- Параметр *-blocks* выводит информацию о каждом блоке в файле (каждый блок в отдельной строке).
- Параметр *-racks* выводит для каждого блока информацию о сегменте и адреса узлов данных.

Команда *hadoop fsck* без аргументов выводит полные инструкции по использованию.

Сканирование блоков на узлах данных

На каждом узле данных работает сканер блоков, который периодически проверяет все блоки, хранящиеся на узле данных. Это позволяет обнаруживать и исправлять

поврежденные блоки до того, как они будут прочитаны клиентами. Сканер `DataBlockScanner` ведет список блоков и последовательно проверяет их в поисках ошибок контрольных сумм.

Блоки периодически проверяются каждые три недели для защиты от ошибок дисков, возникающих со временем (периодичность определяется свойством `dfs.datanode.scan.period.hours`, значение которого по умолчанию составляет 504 часа). Информация о поврежденных блоках передается узлу имен для исправления.

Чтобы получить отчет о проверке блоков для узла данных, откройте веб-интерфейс узла данных по адресу `http://datanode:50075/blockScannerReport`. Пример отчета:

```
Total Blocks : 21131
Verified in last hour : 70
Verified in last day : 1767
Verified in last week : 7360
Verified in last four weeks : 20057
Verified in SCAN_PERIOD : 20057
Not yet verified : 1074
Verified since restart : 35912
Scans since restart : 6541
Scan errors since restart : 0
Transient scan errors : 0
Current scan rate limit KBps : 1024
Progress this period : 109%
Time left in cur period : 53.08%
```

С параметром `listblocks` (`http://datanode:50075/blockScannerReport?listblocks`) отчету предшествует список всех блоков узла данных с последними данными проверки. Приведем фрагмент списка блоков (строки переформатированы по ширине страницы):

```
blk_6035596358209321442 : status : ok      type : none
                           : scan time : 0      not yet verified
blk_3065580480714947643 : status : ok      type : remote
                           : scan time : 1215755306400  2008-07-11 05:48:26,400
blk_8729669677359108508 : status : ok      type : local
                           : scan time : 1215755727345  2008-07-11 05:55:27,345
```

В первом столбце выводится идентификатор блока, за ним следуют пары «ключ-значение». В поле `status` выводится результат проверки `failed` или `ok` (в зависимости от того, была ли обнаружена ошибка контрольной суммы при последнем сканировании блока). В поле `type` выводится тип сканирования: `local`, если

сканирование выполнялось фоновым программным потоком, `remote`, если оно выполнялось клиентом или удаленным узлом данных, или `none`, если сканирование еще не выполнялось. В последнем поле выводится время сканирования в миллисекундах с 1 января 1970 года, а также более удобочитаемое значение.

Balancer

Со временем распределение блоков по узлам данных может утратить баланс. Разбалансировка кластера влияет на локализацию данных MapReduce и повышает нагрузку на интенсивно используемые узлы данных, так что ее лучше избегать.

Демон Hadoop *balancer* перераспределяет блоки, перемещая их с перегруженных узлов данных на недогруженные. При этом соблюдается политика размещения реплик, снижающая вероятность потери данных за счет размещения реплик в разных сегментах (см. «Размещение реплик», с. 115). Блоки перемещаются до тех пор, пока кластер не будет сочен сбалансированным; это означает, что коэффициент использования каждого узла данных (отношение используемого пространства на узле к общей емкости узла) отличается от коэффициента использования кластера (отношение используемого пространства в кластере к общей емкости кластера) не более чем на заданный процент. Балансировка запускается командой

```
% start-balancer.sh
```

Параметр `-threshold` задает пороговый процент, по которому определяется сбалансированность кластера. Он не является обязательным; по умолчанию используется значение 10%. В любой момент времени в кластере может выполняться только один экземпляр *balancer*.

Балансировка выполняется до тех пор, пока кластер не станет сбалансированным. Программа *balancer* создает в стандартном каталоге журналов файл журнала, в который записывается строка для каждой выполненной итерации процесса перераспределения. Пример результата короткого запуска *balancer* для небольшого кластера:

Time Stamp	Iteration#	Bytes Already Moved	Bytes Left To Move	Bytes Being Moved
Mar 18, 2009 5:23:42 PM	0	0 KB	219.21 MB	
			150.29 MB	
Mar 18, 2009 5:27:14 PM	1	195.24 MB	22.45 MB	
			150.29 MB	

The cluster is balanced. Exiting...

Balancing took 6.07293333333333 minutes

Процесс распределения нагрузки выполняется в фоновом режиме, чтобы не создавать лишней нагрузки на кластер и не мешать работе других клиентов. Он ограничивает скорость копирования блоков с одного узла на другой. По умолчанию скорость копирования составляет всего 1 Мбайт/с, но ее можно изменить, задав нужное значение в байтах свойству `dfs.balance.bandwidthPerSec` в файле `hdfs-site.xml`.

Мониторинг

Мониторинг является важной частью администрирования систем. В этом разделе мы рассмотрим средства мониторинга Hadoop и возможность их интеграции с внешними системами мониторинга.

Целью мониторинга является выявление ситуаций, в которых кластер не обеспечивает желаемого уровня сервиса. Самыми важными объектами наблюдения являются основные демоны — узлы имен (первичный и вторичный) и трекер заданий. Сбои на узлах данных и трекерах задач неизбежны, особенно в больших кластерах; предоставьте дополнительную емкость, чтобы кластер мог выдержать наличие небольшого процента неработоспособных узлов в любой момент времени. Кроме средств, описанных далее, некоторые администраторы периодически запускают тестовые задания для проверки эффективности работы кластера.

Хотя эта тема здесь не рассматривается, в области средств мониторинга Hadoop ведется значительная работа. Например, Chukwa — система сбора данных и мониторинга на базе HDFS и MapReduce — особенно хорошо справляется с анализом данных журналов для выявления крупномасштабных тенденций.

Ведение журналов

Все демоны Hadoop создают файлы журналов. Хранящаяся в журналах информация чрезвычайно полезна для определения того, что же происходит в системе. О том, как настраивается процесс ведения журналов, рассказано в разделе «Системные журналы», с. 399.

Выбор уровня ведения журнала

В ходе диагностики бывает очень удобно временно сменить уровень ведения журнала некоторого компонента системы.

У демонов Hadoop имеется веб-страница для изменения уровня ведения любого журнала log4j; она находится в подкаталоге `/logLevel` веб-интерфейса демона. По

соглашению имена журналов Hadoop соответствуют именам классов, выполняющих операции с журналом. Впрочем, из этого правила существуют исключения, так что за именами журналов следует обращаться к исходному коду.

Например, чтобы включить отладочный уровень ведения журнала для класса `JobTracker`, следует открыть веб-интерфейс трекера заданий по адресу `http://jobtracker-host:50030/logLevel` и назначить имени журнала `org.apache.hadoop.mapred.JobTracker` уровень `DEBUG`.

То же самое можно сделать и в командной строке:

```
% hadoop daemonlog -setlevel jobtracker-host:50030 \
  org.apache.hadoop.mapred.JobTracker DEBUG
```

Уровни ведения журналов, заданные таким образом, сбрасываются при перезапуске демона. Если вам потребуется внести постоянное изменение, просто измените файл `log4j.properties` в конфигурационном каталоге. В нашем примере в него добавляется следующая строка:

```
log4j.logger.org.apache.hadoop.mapred.JobTracker=DEBUG
```

Получение данных трассировки стека

Демоны Hadoop предоставляют веб-страницу (`/stacks` в веб-интерфейсе), которая выдает дамп всех программных потоков, выполняемых в JVM демона. Например, дамп трекера заданий можно получить по адресу `http://jobtracker-host:50030/stacks`.

Метрики

Демоны HDFS и MapReduce собирают информацию о событиях и показателях, объединяемых общим термином «метрики». Например, узлы данных собирают следующие (и многие другие) метрики: количество записанных байт, количество реплицированных блоков и количество запросов чтения от клиентов (локальных и удаленных).

Метрики принадлежат некоторому *контексту*. В настоящее время в Hadoop используются контексты «dfs», «mapred», «grpc» и «jvm». Демоны Hadoop обычно собирают метрики в нескольких контекстах. Например, узлы данных собирают метрики для контекстов «dfs», «grpc» и «jvm».

ЧЕМ МЕТРИКИ ОТЛИЧАЮТСЯ ОТ СЧЕТЧИКОВ?

Главным различием является область действия: метрики собираются демонами Hadoop, а счетчики (см. «Счетчики», с. 337) собираются для задач MapReduce и обобщаются для всего задания. Кроме того, они различаются по целевой аудитории: в широком смысле метрики предназначены для администраторов, а счетчики — для пользователей MapReduce.

Также различаются способы сбора и обобщения информации. Счетчики относятся к функциональности MapReduce; система MapReduce следит за тем, чтобы значения счетчиков передавались с трекеров задач, на которых они собираются, трекеру заданий и клиенту, запустившему задание MapReduce. (Счетчики передаются по каналу периодических сигналов RPC; см. «Обновления состояния», с. 261.) Обобщение данных выполняется как трекерами задач, так и трекером заданий.

Механизм сбора метрик отделен от компонента, получающего обновления. Существуют разные подключаемые приемники, в число которых входят локальные файлы, Ganglia и JMX. Демон, собирающий метрики, агрегирует их, прежде чем передавать для вывода.

Контекст определяет единицу публикации; например, вы можете выбрать публикацию контекста «dfs», но не контекста «jvm». Метрики задаются в файле *conf/hadoop-metrics.properties*; по умолчанию все контексты настроены так, что они не публикуют свои метрики. Содержимое файла конфигурации по умолчанию (с опущенными комментариями):

```
dfs.class=org.apache.hadoop.metrics.spi.NullContext  
mapred.class=org.apache.hadoop.metrics.spi.NullContext  
jvm.class=org.apache.hadoop.metrics.spi.NullContext  
rpc.class=org.apache.hadoop.metrics.spi.NullContext
```

Каждая строка в файле определяет отдельный контекст и задает класс, обрабатывающий метрики этого контекста. Класс должен реализовать интерфейс *MetricsContext*; как подсказывает название, класс *NullContext* не публикует и не обновляет метрики¹.

Другие реализации *MetricsContext* рассматриваются в следующих разделах.

Чтобы просмотреть непосредственные метрики, собранные конкретным демоном Hadoop, откройте страницу */metrics*. Эта информация полезна для отладки.

¹ К сожалению, термин «контекст» в данном случае многозначен — им обозначается как набор метрик (контекст «dfs», например), так и класс, публикующий метрики (например, *NullContext*).

Например, метрики трекера заданий в виде простого текста можно просмотреть по адресу <http://jobtracker-host:50030/metrics>. Чтобы получить метрики в формате JSON, используйте адрес <http://jobtracker-host:50030/metrics?format=json>.

FileContext

Класс `FileContext` записывает метрики в локальный файл. Он предоставляет два свойства конфигурации: свойство `fileName` задает абсолютное имя файла, в который записываются данные, а свойство `period` — интервал времени (в секундах) между обновлениями файла. Оба свойства не являются обязательными; если они не заданы, метрики записываются в стандартный вывод каждые 5 секунд.

Свойства конфигурации применяются к имени контекста; при задании свойства его имя присоединяется к имени контекста (с разделением точкой). Например, чтобы вывести контекст «`jvm`» в файл, следует изменить его конфигурацию следующим образом:

```
jvm.class=org.apache.hadoop.metrics.file.FileContext  
jvm.fileName=/tmp/jvm_metrics.log
```

В первой строке контекст «`jvm`» переключается на использование `FileContext`, а во второй свойству `fileName` контекста «`jvm`» задается временный файл. Ниже приведены две выходные строки из файла журнала, разбитые на несколько строк по ширине страницы:

```
jvm.metrics: hostName=ip-10-250-59-159, processName=NameNode, sessionId=,  
gcCount=46, gcTimeMillis=394, logError=0, logFatal=0, logInfo=59, logWarn=1,  
memHeapCommittedM=4.9375, memHeapUsedM=2.5322647, memNonHeapCommittedM=18.25,  
memNonHeapUsedM=11.330269, threadsBlocked=0, threadsNew=0, threadsRunnable=6,  
threadsTerminated=0, threadsTimedWaiting=8, threadsWaiting=13  
jvm.metrics: hostName=ip-10-250-59-159, processName=SecondaryNameNode,   
sessionId=, gcCount=36, gcTimeMillis=261, logError=0, logFatal=0,  
logInfo=18, logWarn=4,  
memHeapCommittedM=5.4414062, memHeapUsedM=4.46756, memNonHeapCommittedM=18.25,  
memNonHeapUsedM=10.624519, threadsBlocked=0, threadsNew=0, threadsRunnable=5,  
threadsTerminated=0, threadsTimedWaiting=4, threadsWaiting=2
```

Класс `FileContext` полезен для отладки в локальной системе, но не подходит для больших кластеров, так как распределение выходных файлов по кластеру затрудняет их анализ.

GangliaContext

Ganglia (<http://ganglia.info/>) — система распределенного мониторинга для очень больших кластеров, распространяемая с открытым кодом. Она спроектирована

с таким расчетом, чтобы создавать минимальные непроизводительные затраты ресурсов в каждом узле кластера. Система Ganglia сама по себе собирает метрики (такие, как использование процессора и памяти), а использование класса `GangliaContext` позволяет внедрить в Ganglia метрики Hadoop.

`GangliaContext` имеет одно обязательное свойство `servers`, в котором передается список пар «хост-порт» для серверов Ganglia (разделенных запятыми и(или) пробелами). Дополнительная информация о настройке этого контекста приведена в вики Hadoop.

Чтобы иметь представление о том, какую информацию можно получить от Ganglia, взгляните на рис. 10.2 — он показывает, как количество задач в очереди трекера заданий изменяется со временем.

NullContextWithUpdateThread

И `FileContext`, и `GangliaContext` передают метрики во внешнюю систему. Однако некоторые системы мониторинга — прежде всего JMX — должны извлекать метрики из Hadoop. Для этой цели предназначен класс `ContextWithUpdateThread`. Как и `NullContext`, он не публикует никакие метрики, но запускает таймер, который периодически обновляет метрики, хранящиеся в памяти. Это гарантируется актуальность метрик на момент их выборки другой системой.

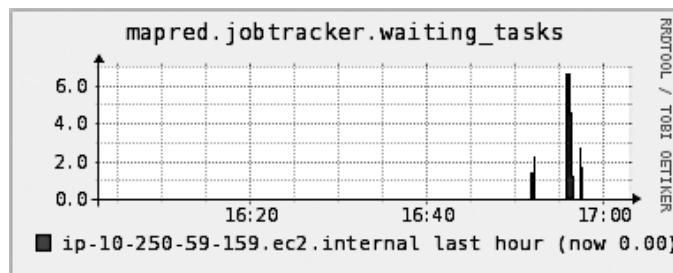


Рис. 10.2. График количества задач в очереди трекера заданий в Ganglia

Все реализации `MetricsContext`, за исключением `NullContext`, выполняют эту функцию обновления (и все предоставляют свойство `period`, по умолчанию равное 5 секундам), так что `NullContextWithUpdateThread` нужно использовать только в том случае, если вы не собираете метрики с использованием другого вывода. Например, при использовании `GangliaContext` этот класс позаботится об обновлении метрик, так что вы сможете использовать JMX без дополнительной конфигурации системы метрик. JMX более подробно рассматривается ниже.

CompositeContext

`CompositeContext` позволяет выводить один набор метрик в разные контексты — например, в `FileContext` и `GangliaContext`. Процесс конфигурации не столь очевиден, и его лучше всего продемонстрировать на примере:

```
jvm.class=org.apache.hadoop.metrics.spi.CompositeContext  
jvm.arity=2  
jvm.sub1.class=org.apache.hadoop.metrics.file.FileContext  
jvm.fileName=/tmp/jvm_metrics.log  
jvm.sub2.class=org.apache.hadoop.metrics.ganglia.GangliaContext  
jvm.servers=ip-10-250-59-159.ec2.internal:8649
```

Свойство `arity` задает количество субконтекстов; в нашем случае их два. Имена свойств каждого субконтекста изменяются так, чтобы в них присутствовал номер субконтекста, отсюда имена `jvm.sub1.class` и `jvm.sub2.class`.

JMX

Java Management Extensions (JMX) — стандартный Java API для мониторинга и управления приложениями. Hadoop включает несколько компонентов `MBean` (Managed Bean), предоставляющих метрики Hadoop приложениям с поддержкой JMX. Компоненты `MBean` предоставляют метрики в контекстах «`dfs`» и «`grpc`», но не в контексте «`mapred`» (на момент написания книги) или «`jvm`» (так как сама JVM предоставляет более широкий набор метрик JVM).

Эти компоненты `MBean` перечислены в табл. 10.3.

Таблица 10.3. Компоненты MBean в Hadoop

Класс MBean	Демоны	Метрики
NameNodeActivityMBean	Узел имен	Метрики активности узла имен (например, количество операций создания файла)
FSNamesystemMBean	Узел имен	Метрики состояния узла имен (например, количество подключенных узлов данных)
DataNodeActivityMBean	Узел данных	Метрики активности узла данных (например, количество прочитанных байт)
FSDatasetMBean	Узел данных	Метрики хранения узла данных (например, емкость и свободное пространство)
RpcActivityMBean	Все демоны, использующие RPC: узел имен, узел данных, трекер заданий и трекер задач	Статистика RPC (например, среднее время обработки)

В JDK входит программа JConsole для просмотра компонентов MBean работающей JVM. Ею удобно пользоваться для просмотра метрик Hadoop (рис. 10.3).

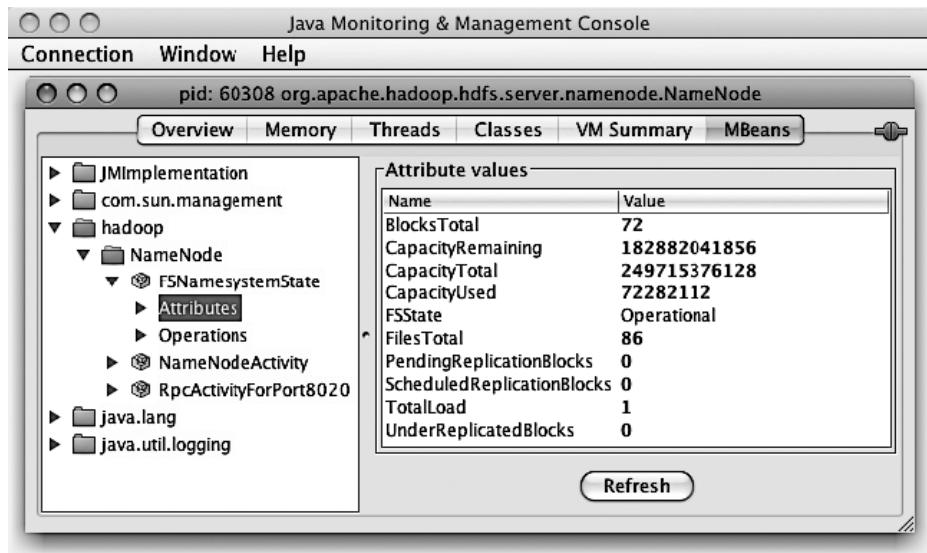


Рис. 10.3. Представление локального узла имен в JConsole, с выводом метрик состояния файловой системы



Хотя вы можете просматривать метрики Hadoop через JMX в конфигурации метрик по умолчанию, метрики не будут обновляться, если только вы не замените реализацию MetricsContext чем-то отличным от NullContext. Например, если метрики будут отслеживаться только через JMX, хорошо подойдет NullContextWithUpdateThread.

Многие сторонние системы мониторинга и оповещения (такие, как Nagios или Hyperic) могут обращаться с запросами к MBean, вследствие чего JMX становится естественным механизмом мониторинга кластера Hadoop из существующей системы мониторинга. Однако для этого вы должны включить удаленный доступ к JMX и выбрать уровень безопасности, подходящий для вашего кластера. Возможные варианты — парольная аутентификация, подключения SSL и клиентская аутентификация SSL. За подробным описанием настройки обращайтесь к официальной документации Java¹.

¹ <http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html>

Все варианты включения удаленного доступа к JMX требуют настройки системных свойств Java, что для Hadoop делается редактированием файла *conf/hadoop-env.sh*. Следующий пример показывает, как включить удаленный доступ к JMX с парольной аутентификацией на узле имен (без SSL). Процесс очень похож на процесс настройки других демонов Hadoop:

```
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.ssl=false  
-Dcom.sun.management.jmxremote.password.file=  
$HADOOP_CONF_DIR/jmxremote.password  
-Dcom.sun.management.jmxremote.port=8004 $HADOOP_NAMENODE_OPTS"
```

Файл *jmxremote.password* содержит список имен пользователей и паролей в виде простого текста; за дополнительной информацией о формате этого файла обращайтесь к документации JMX.

С этой конфигурацией мы можем использовать JConsole для просмотра MBean на удаленном узле имен. Также можно воспользоваться многочисленными инструментами JMX для получения значений атрибутов MBean. Пример использования программы командной строки *jmxquery* (и плагина Nagios, доступного по адресу <http://code.google.com/p/jmxquery/>) для получения количества блоков с недостаточным уровнем репликации:

```
% ./check_jmx -U service:jmx:rmi:///jndi/rmi://namenode-host:8004/jmxrmi -o \  
hadoop:service=NameNode,name=FSNamesystemState -A UnderReplicatedBlocks \  
-w 100 -c 1000 -username monitorRole -password secret  
JMX OK - UnderReplicatedBlocks is 0
```

Команда создает подключение JMX RMI к хосту *namenode-host* на порте 8004 и проводит аутентификацию с заданным именем пользователя и паролем. Она читает атрибут *UnderReplicatedBlocks* объекта с именем *hadoop:service=NameNode,name=FSNamesystemState* и выводит его значение на консоль. Параметры *-w* и *-c* задают уровень предупреждений и критический уровень для значения. Правильные значения обычно определяются после работы кластера в течение некоторого времени.

На практике для мониторинга кластеров Hadoop часто используется Ganglia в сочетании с такими системами оповещения, как Nagios. Ganglia хорошо подходит для эффективного сбора множества метрик и их графического представления, тогда как Nagios и другие аналогичные системы хорошо справляются с отправкой оповещений при достижении критического порога в каком-либо из меньших наборов метрик.

Сопровождение

Стандартные административные процедуры

Резервное копирование метаданных

Если метаданные узла имен будут потеряны или повреждены, вся файловая система станет неработоспособной, поэтому очень важно организовать резервное копирование этих файлов. Всегда храните несколько копий с разным возрастом (допустим, один час, один день, одна неделя и один месяц) для защиты от повреждения — либо собственно в копиях, либо в «живых» файлах, работающих на узле имен.

Простейший способ создания резервных копий — написание сценария, периодически архивирующего содержимое подкаталога *previous.checkpoint* вторичного узла имен (в каталоге, определенном свойством *fs.checkpoint.dir*) во внешнем хранилище. Сценарий должен также проверять целостность копии. Для этого он запускает демона локального узла имен и убеждается в том, что тот успешно загрузил файлы *fsimage* и *edits* в память (например, сканированием журнала узла имен в поисках сообщения об успешном выполнении)¹.

Резервное копирование данных

Хотя система HDFS спроектирована для надежного хранения информации, потери данных все же могут случаться, как и в любой системе хранения, поэтому стратегия резервного копирования абсолютно необходима. При огромных объемах данных, которые может хранить Hadoop, определиться с тем, какие данные следует копировать и где их хранить, не так просто. Прежде всего следует определиться с приоритетами. Наивысшим приоритетом обладают данные, которые невозможно восстановить и которые критичны для бизнеса; если данные относительно легко восстанавливаются или обладают невысокой коммерческой ценностью, им назначается низкий приоритет (и, возможно, их даже не стоит включать в резервное копирование).

В HDFS обычно устанавливается политика управления пользовательскими каталогами. Например, для каталогов устанавливаются пространственные квоты, и каждую ночь проводится их резервное копирование. Какой бы ни была ваша

¹ В Hadoop 2.0 появились инструменты Offline Image Viewer и Offline Edits Viewer, которые могут использоваться для проверки целостности файлов *fsimage* и *edits*. Обе программы поддерживают старые форматы этих файлов, так что их можно использовать для выявления проблем в файлах, сгенерированных предыдущими версиями Hadoop. Программы запускаются командами *hdfs oiv* и *hdfs oeiv*.

политика, проследите за тем, чтобы пользователи знали о ней, и понимали, чего можно ожидать.



Не стоит полагать, будто репликация HDFS заменяет резервное копирование. Ошибки в HDFS, как и сбои оборудования, могут привести к потере реплик. Хотя система Hadoop намеренно проектировалась так, чтобы свести к минимуму вероятность потери данных при сбоях оборудования, полностью исключить такую возможность все же нельзя, особенно в сочетании с программными ошибками или ошибочными действиями человека.

В области резервного копирования следует относиться к HDFS так же, как к RAID. Данные переживают потерю отдельного диска RAID, но не сбой контроллера RAID, программные ошибки (например, перезапись некоторых данных) или повреждение всего массива.

Программа *distcp* идеально подходит для резервного копирования в другие кластеры HDFS (желательно работающие на другой версии программного обеспечения для защиты от потерь, связанных с ошибками HDFS) или другие файловые системы Hadoop (например, S3 или KFS), потому что *distcp* может копировать файлы параллельно. Также можно выбрать для резервного копирования совершенно другую систему хранения информации с использованием методов экспортации данных из HDFS, описанных в разделе «Файловые системы Hadoop», с. 90.

Проверка файловой системы (*fsck*)

Желательно регулярно (например, ежедневно) запускать программу HDFS *fsck* для всей файловой системы с целью активного выявления отсутствующих или поврежденных блоков. См. «Проверка файловой системы (*fsck*)», с. 445.

Балансировка файловой системы

Регулярно запускайте программу *balancer* (см. «Balancer», с. 449) для поддержания равномерного распределения нагрузки на узлах данных.

Включение и исключение узлов

Администратору кластера Hadoop приходится время от времени добавлять и удалять узлы. Например, чтобы увеличить дисковое пространство, доступное для кластера, вы присоединяете к нему новые узлы. И наоборот, иногда кластер бывает нужно сократить, а для этого из него исключаются узлы. В каких-то ситуациях

требуется исключить узел, который ощутимо медленно работает или на котором сбои происходят чаще, чем следует.

На узлах обычно работает узел данных и трекер задач, которые включаются и исключаются из кластера попарно.

Включение новых узлов

Хотя включение нового узла может сводиться к простому добавлению ссылки на узел имен в файл *hdfs-site.xml*, добавлению ссылки на трекер заданий в файл *mapred-site.xml* и запуску демонов узла данных и трекера заданий, обычно желательно иметь список авторизованных узлов.

Разрешая любой машине подключаться к узлу имен и выполнять функции узла данных, вы создаете потенциальный риск для безопасности, потому что машина может получить доступ к данным, которые ей видеть не положено. Более того, поскольку такая машина не является настоящим узлом данных, вы ее не контролируете; она может остановиться в любой момент, что приведет к потере данных. Представьте, что произойдет, если к кластеру подключено несколько таких узлов, а реплики блока данных присутствуют только на «посторонних» узлах? В такой ситуации риск существует даже за брандмауэром (firewall) из-за возможной ошибки конфигурации, поэтому в любом реальном кластере администратор должен явно управлять узлами данных (и трекерами задач).

Узлы данных, которым разрешено подключение к узлу имен, перечисляются в файле, имя которого задается свойством *dfs.hosts*. Этот файл, находящийся в локальной файловой системе узла имен, содержит строку для каждого узла данных, заданного сетевым адресом (полученным от узла данных; чтобы увидеть его, откройте веб-интерфейс узла имен). Если для узла данных потребуется задать несколько сетевых адресов, перечислите их в одной строке, разделяя пробелами.

Аналогичным образом трекеры задач, которые могут подключаться к трекеру заданий, перечисляются в файле, имя которого задается свойством *mapred.hosts*. Как правило, существует один общий (*включаемый*) файл, на который ссылаются *dfs.hosts* и *mapred.hosts*, так как на узлах кластера работают демоны как узла данных, так и трекера задач.



Не путайте файл (или файлы), заданный свойствами *dfs.hosts* и *mapred.hosts*, с файлом *slaves*. Первый используется узлом имен и трекером заданий для определения того, какие рабочие узлы могут к ним подключаться.

Файл *slaves* используется управляющими сценариями Hadoop для выполнения операций в масштабе кластера — например, перезапуска кластера.

Демонами Hadoop он не используется.

Чтобы добавить новые узлы в кластер

- 1) добавьте сетевые адреса новых узлов во включаемый файл;
- 2) обновите узел имен новым набором разрешенных узлов данных, используя следующую команду:

```
% hadoop dfsadmin -refreshNodes;
```

- 3) обновите трекер заданий новым набором разрешенных трекеров задач, используя следующую команду:

```
% hadoop mradmin -refreshNodes;
```

- 4) обновите файл *slaves* новыми узлами данных, чтобы они включались в будущие операции, выполняемые управляющими сценариями Hadoop;
- 5) запустите новые узлы данных и трекеры задач;
- 6) убедитесь в том, что новые узлы данных и трекеры задач отображаются в веб-интерфейсе.

HDFS не перемещает блоки со старых узлов данных на новые для балансировки кластера. Для выполнения балансировки следует запустить программу *balancer* (см. «Balancer», с. 449).

Исключение старых узлов

Система HDFS проектировалась так, чтобы успешно выдерживать сбои узлов данных, но это не означает, что простое массовое завершение узлов данных не будет иметь нежелательных последствий. Например, с уровнем репликации 3 существует очень большая вероятность того, что одновременное отключение трех узлов данных, находящихся в разных сегментах, приведет к потере данных. Чтобы вывести из кластера узлы данных, следует сообщить о них узлу имен, чтобы он мог реплицировать блоки на другие узлы данных до отключения.

С трекерами задач Hadoop не выдвигает столь жестких требований. Если вы отключите трекер задач, в котором имеются работающие задачи, трекер заданий заметит сбой и перепланирует задачи на других трекерах.

Процессом исключения узлов управляет *файл исключения*, который для HDFS задается свойством `dfs.hosts.exclude`, а для MapReduce – свойством `mapred.hosts.exclude`. Эти свойства достаточно часто ссылаются на один и тот же файл. В файле исключения перечисляются узлы, которым запрещено подключение к кластеру.

Возможность подключения трекера задач к трекеру заданий проста: трекер задач может подключиться только в том случае, если он присутствует в файле

включения, но отсутствует в файле исключения. Если файл включения не указан или пуст, считается, что в нем находятся все узлы.

Для HDFS действуют несколько иные правила. Если узел данных присутствует и в файле включения, и в файле исключения, он может подключиться, но только для вывода из кластера. В табл. 10.4 перечислены различные комбинации условий для узлов данных. Как и в случае с трекерами задач, если файл включения не указан или пуст, считается, что в нем находятся все узлы.

Таблица 10.4. Приоритеты файлов включения и исключения HDFS

Узел присутствует в файле включения	Узел присутствует в файле исключения	Интерпретация
Нет	Нет	Узел не может подключаться
Нет	Да	Узел не может подключаться
Да	Нет	Узел может подключаться
Да	Да	Узел может подключаться, но медленно будет исключен

Чтобы исключить узлы из кластера:

- 1) добавьте сетевые адреса исключаемых узлов в файл исключения. Пока не обновляйте файл включения;
- 2) обновите узел имен новым набором разрешенных узлов данных, используя следующую команду:

```
% hadoop dfsadmin -refreshNodes;
```

- 3) обновите трекер заданий новым набором разрешенных трекеров задач, используя следующую команду:

```
% hadoop mradmin -refreshNodes;
```

- 4) откройте веб-интерфейс и проверьте, изменилось ли состояние исключаемых узлов на «Decommission In Progress». В этом состоянии узлы начинают копировать свои блоки на другие узлы данных в кластере;
- 5) когда у всех узлов данных появится состояние «Decommissioned», репликация всех блоков завершена. Завершите работу отключенных узлов;
- 6) удалите узлы из файла включения и выполните команду:

```
% hadoop dfsadmin -refreshNodes
```

```
% hadoop mradmin -refreshNodes
```

- 7) удалите узлы из файла *slaves*.

Обновления

Обновление кластера MapReduce и HDFS требует тщательного планирования. По важности на первом месте стоит обновление HDFS: если версия структуры данных файловой системы изменится, обновление автоматически преобразует данные и метаданные файловой системы в формат, совместимый с новой версией. Как и при любой процедуре, связанной с миграцией данных, существует некоторый риск потери данных — обязательно проведите резервное копирование данных и метаданных (см. «Стандартные административные процедуры», с. 458).

Процесс планирования должен включать в себя пробный запуск в малом тестовом кластере с копией данных, потеря которой не создаст проблем. Пробный запуск позволит вам познакомиться с процессом, настроить его под вашу конкретную конфигурацию кластера и инструментарий, а также устраниТЬ любые проблемы перед запуском процедуры обновления в рабочем кластере. Еще одним преимуществом тестового кластера является его доступность для тестирования клиентских обновлений. Общие проблемы совместимости для клиентов описаны в разделе «Совместимость», с. 45.

Обновление кластера при неизменной структуре файловой системы проходит достаточно прямолинейно: установите новые версии HDFS и MapReduce в кластере (и на клиентских машинах), завершите работу старых демонов, обновите конфигурационные файлы, запустите новых демонов и переключите клиентов на использование новых библиотек. Этот процесс обратим, так что отменить обновление тоже будет несложно.

После каждого успешного обновления следует выполнить пару завершающих операций:

- Удалите из кластера старую установку и конфигурационные файлы.
- Избавьтесь от предупреждений о несоответствии версий в коде и конфигурации.

Данные HDFS и обновления метаданных

Если вы используете только что описанную процедуру для перехода на новую версию HDFS с другой версией структур данных файловой системы, узел имен не запустится. В журнале появится сообщение, которое выглядит примерно так:

```
File system image contains an old layout version -16.  
An upgrade to version -18 is required.  
Please restart NameNode with -upgrade option.
```

Как определить, нужно ли обновлять файловую систему? Проще всего выполнить пробный запуск в тестовом кластере.

Обновление HDFS копирует метаданные и данные предыдущей версии. Обновление не удваивает затраты дискового пространства в кластере, так как узлы данных используют жесткие ссылки для хранения двух ссылок (на текущую и предыдущую версию) одного блока данных. Это позволит при необходимости легко вернуться к предыдущей версии файловой системы. Следует понимать, что все изменения, внесенные в данные обновленной системы, будут потеряны после отмены.

Вы можете хранить только предыдущую версию файловой системы, то есть возврат на несколько версий назад невозможен. Таким образом, для следующего обновления данных и метаданных HDFS необходимо удалить предыдущую версию — этот процесс называется *закреплением* обновления. После закрепления обновления вернуться к предыдущей версии уже не удастся.

Как правило, при обновлении можно пропускать версии (например, перейти с версии 0.18.3 на 0.20.0 без предварительной установки 0.19.x), но в некоторых случаях промежуточная установка все же обязательна. Такие случаи четко оговорены в замечаниях к выпуску.

Обновлять следует только «здравые» файловые системы. Прежде чем запускать обновление, проведите полную проверку (см. «Проверка файловой системы (*fsck*)», с. 445). В качестве дополнительной меры предосторожности можно сохранить копию вывода *fsck* с полным перечнем файлов и блоков в системе, чтобы сравнить ее с результатами выполнения *fsck* после обновления.

Также стоит удалить временные файлы после обновления — как файлы из системного каталога MapReduce в HDFS, так и локальные временные файлы.

Итак, после всей предварительной подготовки высокоуровневая процедура обновления кластера с миграцией структуры файловой системы выглядит так:

1. Прежде чем переходить к следующему обновлению, убедитесь в том, что предыдущее обновление было закреплено.
2. Завершите MapReduce и уничтожьте все «бесхозные» процессы задач на трекерах задач.
3. Завершите HDFS и создайте резервную копию узлов имен.
4. Установите новые версии Hadoop HDFS и MapReduce в кластере и клиентах.
5. Запустите HDFS с параметром *-upgrade*.
6. Дождитесь завершения обновления.
7. Проведите проверки работоспособности HDFS.
8. Запустите MapReduce.
9. Отмените или завершите обновление (не обязательно).

В процессе обновления рекомендуется удалить сценарии Hadoop из переменной окружения PATH. Это заставит вас явно указать, какую версию сценариев следует запустить. Для новых установочных каталогов удобно определить две переменные окружения; в следующих инструкциях определяются переменные OLD_HADOOP_INSTALL и NEW_HADOOP_INSTALL.

Запуск обновления

Чтобы провести обновление, выполните следующую команду (шаг 5 высокоуровневой процедуры обновления):

```
% $NEW_HADOOP_INSTALL/bin/start-dfs.sh -upgrade.
```

Узел имен обновляет свои метаданные, размещая предыдущую версию в новом каталоге с именем *previous*:

```
 ${dfs.name.dir}/current/VERSION
        /edits
        /fsimage
        /fstime
 /previous/VERSION
        /edits
        /fsimage
        /fstime
```

Узлы данных тоже обновляют свои каталоги хранения данных, сохраняя старую копию в каталоге с именем *previous*.

Ожидание завершения обновления

Процесс обновления проходит не мгновенно, но вы можете проверить прогресс обновления командой *dfsadmin* (шаг 6; события обновления также фиксируются в файлах журналов демонов):

```
% $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -upgradeProgress status
Upgrade for version -18 has been completed.
Upgrade is not finalized.
```

Проверка обновления

Из результатов команды видно, что обновление завершено. На этой стадии следует провести простейшие проверки работоспособности файловой системы (шаг 7) – например, проверить файлы и блоки программой *fsck*, протестировать основные операции с файлами. Возможно, на время выполнения проверок (выполняемых только с чтением данных) стоит перевести HDFS в безопасный режим, чтобы другие пользователи не могли вносить изменения.

Отмена обновления (не обязательно)

Если выяснится, что новая версия работает некорректно, вы можете вернуться к предыдущей версии (шаг 9). Это возможно только в том случае, если обновление еще не было завершено.



Процедура отмены возвращает файловую систему к состоянию, предшествующему выполнению обновления, так что любые последующие изменения будут потеряны. Иначе говоря, происходит возврат к предыдущему состоянию файловой системы, а не понижение текущего состояния файловой системы к предыдущей версии.

Начните с остановки новых демонов:

```
% $NEW_HADOOP_INSTALL/bin/stop-dfs.sh
```

Затем запустите старую версию HDFS с параметром *-rollback*:

```
% $OLD_HADOOP_INSTALL/bin/start-dfs.sh -rollback
```

Команда заставляет узел имен и узлы данных заменить свои текущие каталоги хранения данных их предыдущими копиями. Файловая система возвращается в прежнее состояние.

Закрепление обновления (не обязательно)

Когда новая версия HDFS вас устроит, закрепите обновление (шаг 9), чтобы удалить предыдущие каталоги.



После того как обновление будет закреплено, вернуться к предыдущей версии уже не удастся.

Этот шаг необходимо выполнить перед выполнением следующего обновления:

```
% $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -finalizeUpgrade  
% $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -upgradeProgress status  
There are no upgrades in progress.
```

Система HDFS полностью обновлена до следующей версии.

11 Pig

Pig повышает уровень абстракции при обработке больших наборов данных. MapReduce позволяет программисту задать функцию отображения, а затем функцию свертки, но для адаптации механизма обработки данных к этой схеме часто приходится использовать несколько стадий MapReduce, а это усложняет задачу. При использовании Pig структуры данных намного сложнее, с множеством значений и многоуровневой иерархией, а преобразования, применяемые к данным, намного мощнее (например, реализация соединений в MapReduce – задача не для слабо-нервных).

Pig состоит из двух основных частей:

- Язык для описания потоков данных, называемый *Pig Latin*.
- Исполнительная среда для запуска программ Pig Latin. В настоящее время доступны два варианта: локальное выполнение на одной JVM и распределенное выполнение в кластере Hadoop.

Программа Pig Latin состоит из серии операций (преобразований), которые применяются к входным данным для получения выходных данных. В целом эти операции описывают поток данных, который превращается исполнительной средой Pig в исполняемое представление, а затем запускается для выполнения. Во внутренней реализации Pig трансформирует преобразования в серию заданий MapReduce, однако эта трансформация в основном остается скрытой от программиста, что позволяет ему сосредоточиться на данных, а не на природе исполнения.

Язык сценариев Pig предназначен для анализа больших наборов данных. MapReduce часто критикуют за большую длину цикла разработки. Написание отображений и сверток, компилирование и упаковка кода, отправка заданий, получение результатов — все это занимает много времени. Даже с технологией Streaming, исключающей шаг компилирования и упаковки, процесс все равно остается сложным. Сильной стороной Pig является прежде всего возможность обработки терабайтов данных простым вводом полудюжины строк Pig Latin с консоли. В конце концов, система Pig была создана в Yahoo! для упрощения анализа гигантских наборов данных исследователями и инженерами. Pig предоставляет разностороннюю поддержку для написания запросов, так как в распоряжении программиста имеются дополнительные команды анализа структур данных в программах во время их написания. И что еще удобнее, Pig может провести тестовый запуск для представительного подмножества входных данных. Вы увидите, нет ли ошибок в процессе обработки, прежде чем распространять его на весь набор данных.

Одной из целей разработки Pig была хорошая расширяемость. Настраиваются практически все стадии обработки: загрузка, хранение, фильтрация, группировка, соединение — каждая операция может быть изменена при помощи пользовательских функций (UDF, User-Defined Function). Эти функции работают с вложенной моделью данных Pig, поэтому они могут очень глубоко интегрироваться с операторами Pig. Кроме того, пользовательские функции обычно лучше подходят для повторного использования, чем библиотеки, разработанные для написания программ MapReduce.

Впрочем, Pig подходит не для любых задач обработки данных. Как и MapReduce, эта технология проектировалась в расчете на пакетный режим обработки. Если вы хотите выполнить запрос, который задействует только небольшой объем данных большого набора, эффективность Pig оставит желать лучшего, потому что технология рассчитана на сканирование всего набора данных или, по крайней мере, его значительной части.

В некоторых случаях программы Pig по эффективности уступают программам, написанным для MapReduce. Однако с каждым выпуском разрыв сокращается, а группа Pig реализует все более хитроумные алгоритмы для применения реляционных операторов Pig. Можно без преувеличения сказать, что, если только вы не планируете потратить много сил на оптимизацию кода Java MapReduce, написание запросов на Pig Latin сэкономит вам время.

В этой главе приведено простейшее введение в Pig. За более подробным описанием обращайтесь к книге Алана Гейтса (Alan Gates) «Programming Pig» (O'Reilly, 2011).

Установка и запуск Pig

Pig работает как приложение на стороне клиента. Даже если вы захотите запустить Pig в кластере Hadoop, в самом кластере ничего дополнительно устанавливать не придется: Pig запускает задания и взаимодействует с HDFS (или другими файловыми системами Hadoop) с рабочей станции.

Процесс установки тривиален. Обязательным условием является предварительная установка Java 6 (а в Windows вам понадобится Cygwin). Загрузите стабильную версию по адресу <http://pig.apache.org/releases.html> и распакуйте архив в подходящее место на вашей рабочей станции:

```
% tar xzf pig-x.y.z.tar.gz.
```

Каталог двоичных файлов Pig удобно включить в путь командной строки, например:

```
% export PIG_INSTALL=/home/tom/pig-x.y.z  
% export PATH=$PATH:$PIG_INSTALL/bin
```

Переменная окружения JAVA_HOME также должна ссылаться на подходящую установку Java.

Чтобы просмотреть инструкции по использованию, введите команду *pig -help*.

Режимы исполнения

Pig поддерживает два режима исполнения: локальный режим и режим MapReduce.

Локальный режим

В локальном режиме Pig работает на одной JVM и обращается к локальной файловой системе. Этот режим подходит только для небольших наборов данных и только для проверки работы Pig.

Режим исполнения задается параметром *-x* или *-execType*. Чтобы запустить Pig в локальном режиме, задайте параметру значение *local*:

```
% pig -x local  
grunt>
```

Команда запускает Grunt – интерактивную оболочку Pig, которая ниже рассматривается более подробно.

Режим MapReduce

В режиме MapReduce Pig трансформирует запросы в задания MapReduce и запускает их в кластере Hadoop – псевдораспределенном или распределенном. Именно режим MapReduce (в распределенном кластере) следует использовать при запуске Pig с большими наборами данных.

Чтобы использовать режим MapReduce, необходимо сначала убедиться в том, что загруженная вами версия Pig совместима с используемой версией Hadoop. Версии Pig работают только с конкретными версиями Hadoop, перечисленными в замечаниях к выпуску.

Для поиска клиента Hadoop Pig использует переменную окружения `HADOOP_HOME`. Если значение переменной не задано, Pig использует собственную копию библиотек Hadoop. Учтите, что эта копия может не соответствовать версии Hadoop, используемой в кластере, поэтому переменную `HADOOP_HOME` лучше задать явно.

Затем необходимо передать Pig информацию об узле имен и трекере заданий кластера. Если установка Hadoop в `HADOOP_HOME` уже содержит соответствующие настройки, делать ничего не нужно. В противном случае следует задать в переменной `HADOOP_CONF_DIR` каталог, содержащий файл (или файлы) Hadoop с определениями `fs.default.name` и `mapred.job.tracker`.

Также эти два свойства можно задать в файле `pig.properties` каталога `conf` Pig directory (или каталога, заданного переменной `PIG_CONF_DIR`). Пример для псевдораспределенной конфигурации:

```
fs.default.name=hdfs://localhost/  
mapred.job.tracker=localhost:8021
```

Настройив Pig для подключения к кластеру Hadoop, можно запустить Pig с параметром `-x`, равным `mapreduce`, или вообще без параметра, потому что режим MapReduce используется по умолчанию:

```
% pig  
2012-01-18 20:23:05,764 [main] INFO org.apache.pig.Main -  
Logging error messages to: /private/tmp/pig_1326946985762.log  
2012-01-18 20:23:06,009 [main] INFO  
    org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -  
    Connecting to hadoop file system at:  
    hdfs://localhost/2012-01-18 20:23:06,274 [main] INFO  
    org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -  
    Connecting to map-reduce job tracker at: localhost:8021 grunt>
```

Pig включает в выходные данные информацию о файловой системе и трекере задач, к которым он подключился.

Запуск программ Pig

Существует три способа выполнения программ Pig. Все они работают как в локальном режиме, так и в режиме MapReduce.

Сценарий

Pig может выполнить файл сценария, содержащий команды Pig. Например, команда `pig script.pig` выполняет команды в локальном файле `script.pig`. Для очень коротких сценариев можно воспользоваться параметром `-e` для выполнения сценария, встроенного прямо в командную строку.

Grunt

Grunt — интерактивная оболочка для выполнения программ Pig. Grunt запускается в том случае, если для Pig не задан выполняемый файл, а параметр `-e` не используется. Сценарии Pig также могут запускаться прямо из Grunt командами `run` и `exec`.

Встроенные программы

Программы Pig можно запускать из кода Java при помощи класса `PigServer` — по аналогии с использованием JDBC для запуска программ SQL из Java. Для программного доступа к Grunt используется класс `PigRunner`.

Grunt

Средства редактирования строк в Grunt аналогичны тем, которые используются в GNU Readline (`bash`, многие другие приложения командной строки). Например, комбинация клавиш `Ctrl+E` переводит курсор в конец строки. Grunt также запоминает историю команд¹; для вызова строки из буфера истории используются комбинации клавиш `Ctrl+P` и `Ctrl+N` (предыдущая и следующая команда соответственно) или клавиши управления курсором.

Другая удобная функция Grunt — автозавершение ключевых слов и функций Pig Latin при нажатии клавиши `Tab`. Допустим, имеется следующая незавершенная строка:

```
grunt> a = foreach b ge
```

¹ История хранится в файле с именем `.pig_history` в домашнем каталоге.

При нажатии клавиши Tab ge расширяется до *generate*, ключевого слова Pig Latin:

```
grunt> a = foreach b generate
```

Чтобы настроить шаблоны автозавершения, создайте файл с именем *autocomplete* и разместите его в каталоге Pig (например, в подкаталоге *conf* установочного каталога Pig) или в каталоге, из которого была запущена оболочка Grunt. Каждая строка файла содержит одну лексему без пропусков (whitespace). При сопоставлении учитывается регистр символов. Очень удобно включить в этот файл часто используемые пути (особенно потому, что Pig не поддерживает автоматическое завершение имен файлов) или имена созданных вами пользовательских функций.

Команда *help* выводит список команд. Завершив сеанс работы с Grunt, выйдите из программы командой *quit*.

Редакторы Pig Latin

PigPen — плагин для Eclipse, предназначенный для разработки программ Pig. Он включает текстовый редактор сценариев Pig, генератор примеров (эквивалент команды **ILLUSTRATE**) и кнопку для выполнения сценария в кластере Hadoop. Также имеется окно графа операторов, в котором сценарий отображается в графической форме для визуализации потока данных. Полное описание установки и инструкции по использованию приведены в вики Pig по адресу <https://cwiki.apache.org/confluence/display/PIG/PigTools>.

Также существуют модули выделения синтаксиса Pig Latin для других редакторов, включая Vim и TextMate. За подробностями обращайтесь к вики Pig.

Пример

Начнем с простого примера: напишем программу для вычисления максимальной зарегистрированной температуры по годам для набора метеорологических данных на Pig Latin (по аналогии с программой для MapReduce из главы 2). Вся программа занимает всего несколько строк:

```
-- max_temp.pig: Определение максимальной температуры за год
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
(quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
```

```
max_temp = FOREACH grouped_records GENERATE group,
    MAX(filtered_records.temperature);
DUMP max_temp;
```

Чтобы понять, что происходит в программе, мы используем интерпретатор Grunt, который позволяет вводить строки и работать в интерактивном режиме. Запустите Grunt в локальном режиме и введите первую строку сценария Pig:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year:chararray, temperature:int, quality:int);
```

Для простоты программа предполагает, что входные данные представляют собой текст, разделенный табуляциями; каждая строка содержит только поля года, температуры и качества. (Как вы убедитесь позднее, Pig обладает большей гибкостью в отношении входных форматов.) Эта строка описывает входные данные, которые мы собираемся обрабатывать. Обозначение `year:chararray` описывает имя и тип поля; `chararray` – аналог типа Java `string`, а `int` – аналог типа Java `int`. Оператор `LOAD` получает URI; в нашем примере используется локальный файл, но его можно заменить HDFS URI. Секция `AS` (не обязательная) присваивает полям имена, чтобы к ним было удобнее обращаться в последующих инструкциях.

Результатом оператора `LOAD`, а на самом деле любого оператора Pig Latin, является *отношение*, которое представляет собой набор *кортежей* (*tuples*). Кортеж очень похож на строку в таблице базы данных, состоящую из нескольких полей в определенном порядке. В нашем примере функция `LOAD` создает набор кортежей (`year, temperature, quality`) по данным входного файла. Мы записываем отношение с одним кортежем на строку, при этом кортежи представляются списками элементов, разделенными запятыми и заключенными в круглые скобки:

```
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
```

Отношениям присваиваются *псевдонимы* (*aliases*) – имена, по которым к ним можно обращаться. В нашем примере отношению присваивается псевдоним `records`. Его содержимое выводится оператором `DUMP`:

```
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

Оператор `DESCRIBE` выводит структуру (*схему*) отношения по его псевдониму:

```
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
```

Из выходных данных мы видим, что отношение `records` состоит из трех полей с псевдонимами `year`, `temperature` и `quality` (которые были присвоены в секции `AS`). Типы полей также были заданы в секции `AS`. Типы Pig более подробно описаны ниже.

Вторая инструкция удаляет записи с отсутствующей температурой (признаком которой является значение 9999) или недопустимым значением качества. В нашем тестовом наборе данных ни одна запись не отфильтровывается:

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>>   (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grunt> DUMP filtered_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

Третья инструкция использует функцию `GROUP` для группировки отношения `records` по полю `year`. Для просмотра результата снова используется `DUMP`:

```
grunt> grouped_records = GROUP filtered_records BY year;
grunt> DUMP grouped_records;
(1949,{(1949,111,1),(1949,78,1)})
(1950,{(1950,0,1),(1950,22,1),(1950,-11,1)})
```

Теперь мы имеем два кортежа (строки) — по одному для каждого года во входных данных. Первое поле каждого кортежа (год) используется для группировки, а второе содержит неупорядоченное множество кортежей для этого кода. Для представления неупорядоченных множеств в Pig Latin используются фигурные скобки.

После такой группировки остается лишь найти максимальную температуру по кортежам каждого неупорядоченного множества. Но сначала попробуем понять структуру отношения `grouped_records`:

```
grunt> DESCRIBE grouped_records;
grouped_records: {group: chararray,filtered_records: {year: chararray,
temperature: int,quality: int}}
```

Из результатов видно, что Pig назначает полю группировки псевдоним `group`, а второе поле имеет такую же структуру, как и сгруппированное отношение `filtered_records`. Располагая такой информацией, можно опробовать четвертое преобразование:

```
grunt> max_temp = FOREACH grouped_records GENERATE group,  
>>   MAX(filtered_records.temperature);
```

Оператор `FOREACH` обрабатывает каждую строку и генерирует для нее производный набор строк, поля которых определяются секцией `GENERATE`. В нашем примере первое поле `group` содержит просто год. Со вторым полем дело обстоит сложнее.

Ссылка `filtered_records.temperature` обозначает поле `temperature` неупорядоченного множества `filtered_records` отношения `grouped_records`. `MAX` — встроенная функция для вычисления максимального значения полей неупорядоченного множества. В нашем случае она вычисляет максимальную температуру полей в каждом неупорядоченном множестве `filtered_records`. Проверим результат:

```
grunt> DUMP max_temp;  
(1949,111)  
(1950,22)
```

Итак, мы успешно вычислили максимальную температуру за каждый год.

Генерирование примеров

В приведенном примере использовался маленький набор данных из нескольких записей, чтобы было проще следить за преобразованиями данных и выполнять отладку. Создание сокращенного набора данных — настоящее искусство. В идеале такой набор должен быть достаточно разнообразным, чтобы проверить все случаи, встречающиеся в запросах (свойство полноты), но при этом не слишком большим, чтобы программист в нем мог разобраться (свойство компактности). Случайная выборка обычно не подходит, потому что операции соединения и фильтрации обычно исключают все случайные данные и оставляют пустой результат, не дающий представления об общем потоке данных.

Pig предоставляет в распоряжение программиста оператор `ILLUSTRATE`, генерирующий сравнительно полный и компактный набор данных. Ниже приведен результат выполнения `ILLUSTRATE` (переформатированный по ширине страницы):

```
grunt> ILLUSTRATE max_temp;
-----
| records | year:chararray | temperature:int | quality:int |
|-----|
|       | 1949             | 78            | 1           |
|       | 1949             | 111           | 1           |
|       | 1949             | 9999          | 1           |
|-----|
| filtered_records | year:chararray | temperature:int | quality:int |
|-----|
|       | 1949             | 78            | 1           |
|       | 1949             | 111           | 1           |
|-----|
| grouped_records | group:chararray | filtered_records:bag{:tuple(year:chararray, | |
|-----|
temperature:int,quality:int)} | |
|       | 1949             | {(1949, 78, 1), (1949, 111, 1)} |
|-----|
| max_temp | group:chararray | :int        |
|-----|
|       | 1949             | 111           |
|-----|
```

Обратите внимание: Pig использует некоторые исходные данные (сгенерированный набор данных должен быть реалистичным), а также создает новые данные. Заметив в запросе специальное значение 9999, Pig создает кортеж с этим значением для проверки инструкции FILTER.

Выходные данные ILLUSTRATE понятны и доступны. С ними вы легко разберетесь, как работает ваш запрос.

Сравнение с базами данных

Увидев Pig в действии, читатель может решить, что язык Pig Latin похож на SQL. Присутствие таких операторов, как `GROUP BY` и `DESCRIBE`, только укрепляет это впечатление. Однако между двумя языками существуют различия — как, впрочем, и между Pig и реляционными системами управления базами данных (РСУБД) в целом.

Самое важное различие заключается в том, что Pig Latin — язык программирования преобразований описания потока данных, а SQL — декларативный язык программирования. Иначе говоря, программа, написанная на Pig Latin, представляет собой пошаговый набор операций над входным отношением, каждый шаг которого представляет собой одно преобразование. Вместе с тем инструкции SQL представляют собой набор ограничений, которые в совокупности определяют результат. Программирование на Pig Latin во многих отношениях напоминает работу на уровне планировщика запросов РСУБД, который определяет, как преобразовать декларативную инструкцию в последовательность действий.

РСУБД хранят данные в таблицах с жестко определенной схемой. Pig не предъявляет столь жестких требований к обрабатываемым данным: схему можно определить во время выполнения, но она не обязательна. Фактически схема будет работать с любым источником кортежей (хотя источник должен поддерживать параллельное чтение — например, храниться в нескольких файлах), для чтения которых из их физического представления используется пользовательская функция. Самое распространенное представление — текстовый файл с полями, разделенными табуляциями; Pig предоставляет встроенную функцию загрузки для этого формата. В отличие от традиционных баз данных, не существует процесса импортирования данных для их загрузки в РСУБД. Данные загружаются из файловой системы (обычно HDFS) на первом шаге обработки.

Поддержка составных, вложенных структур данных отличает Pig от языка SQL, работающего с «плоскими» структурами данных. Кроме того, возможность использования пользовательских функций и потоковых операторов, плотно интегрированных в язык, и вложенных структур данных открывает более широкие возможности настройки Pig Latin, чем в большинстве диалектов SQL.

РСУБД обладают средствами поддержки оперативных запросов с малой задержкой (транзакции, индексы), отсутствующими в Pig. Как упоминалось ранее, Pig не поддерживает произвольное чтение или запросы, выполняемые за десятки миллисекунд. Также не поддерживается произвольная запись для обновления небольших частей данных; все операции записи осуществляются массово в потоковом режиме, как и в MapReduce.

Hive (см. главу 12) занимает промежуточное место между Pig и традиционными РСУБД. Как и Pig, система Hive использует HDFS для хранения информации, но

в остальном между ними существует ряд важных различий. Язык запросов Hive – HiveQL – основан на SQL, и каждый, кто знаком с SQL, очень быстро начинает писать запросы на HiveQL. Как и РСУБД, Hive работает с данными, хранящимися в таблицах; однако Hive может связать схему с данными, уже существующими с HDFS, так что шаг загрузки не является обязательным. Hive не поддерживает запросы с малой задержкой – особенность, общая с Pig.

Pig Latin

В этом разделе приведено неформальное описание синтаксиса и семантики языка программирования Pig Latin. Раздел не является полным справочником по языку¹, но он дает неплохое представление об основных конструкциях Pig Latin.

Структура

Программа Pig Latin состоит из последовательности инструкций, каждая из которых представляет некоторую команду или операцию. Например, операция GROUP является разновидностью инструкции:

```
grouped_records = GROUP records BY year;
```

Другой пример инструкции – команда вывода списка файлов в файловой системе Hadoop:

```
ls /
```

Инструкции обычно завершаются символом «;», как видно из примера с GROUP. Собственно, эта инструкция всегда завершается символом «;»; его отсутствие является синтаксической ошибкой. Вместе с тем завершать команду ls символом «;» не обязательно. В общем случае символ «;» не обязателен для команд интерактивного взаимодействия с Grunt. В эту группу входят интерактивные команды Hadoop, а также диагностические операторы (например, DESCRIBE). Впрочем, завершающий символ «;» никогда не является ошибкой, так что если у вас возникнут сомнения – проще добавить его.

Инструкции, которые должны завершаться символом «;», могут быть разбиты на несколько строк для удобочитаемости:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'  
AS (year:chararray, temperature:int, quality:int);
```

¹ Pig Latin не имеет формального определения как такового, но достаточно подробное руководство доступно по ссылке на веб-сайте Pig по адресу <http://pig.apache.org/>.

В Pig Latin существует две разновидности комментариев. Двойной дефис (--) определяет однострочные комментарии. Интерпретатор Pig Latin игнорирует все символы от первого дефиса до конца строки:

```
-- Моя программа  
DUMP A; -- Что хранится в А?
```

Комментарии в стиле С обладают большей гибкостью; начало и конец блока комментария отмечаются маркерами /* и */. Такие комментарии могут быть встроены в одну строку, а могут распространяться на несколько строк:

```
/*  
 * Описание моей программы  
 * в нескольких строках.  
 */  
A = LOAD 'input/pig/join/A';  
B = LOAD 'input/pig/join/B';  
C = JOIN A BY $0, /* игнорируем*/ B BY $1;  
DUMP C;
```

В Pig Latin существуют ключевые слова, которые имеют особый смысл в языке и не могут использоваться в качестве идентификаторов. В эту группу входят операторы (`LOAD`, `ILLUSTRATE`), команды (`cat`, `ls`), выражения (`matches`, `FLATTEN`) и функции (`DIFF`, `MAX`) — все эти категории описаны в следующих разделах.

В Pig Latin установлены смешанные правила учета регистра символов. В операторах и командах регистр символов не учитывается (чтобы упростить их интерактивное использование), однако в псевдонимах и именах функций он должен соблюдаться.

Инструкции

В процессе выполнения программы на Pig Latin происходит последовательный разбор каждой инструкции. Если в инструкции обнаруживаются синтаксические ошибки или другие (семантические) проблемы — например, неопределенные псевдонимы, интерпретатор завершает работу и выводит сообщение об ошибке. Интерпретатор строит логический план для каждой операции с отношениями, формируя основу программы на Pig Latin. Логический план инструкции добавляется в логический план программы, после чего интерпретатор переходит к следующей инструкции.

Важно, что в процессе построения логического плана программы обработка данных не происходит. Вспомните программу на Pig Latin из первого примера:

```
-- max_temp.pig: Определение максимальной температуры за год
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
(quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

Когда интерпретатор Pig Latin обнаруживает первую строку с инструкцией `LOAD`, он проверяет ее синтаксическую и семантическую правильность и включает ее в логический план, но не загружает данные из файла (и даже не проверяет, что этот файл существует). В самом деле, а куда он должен загружать их? В память? Даже если бы данные поместились в памяти, что с ними делать? Возможно, не все входные данные будут использованы (скажем, из-за того, что они будут отфильтрованы следующими командами), поэтому и загружать их будет бесполезно. Суть в том, что начинать какую-либо обработку до определения всего потока данных бессмысленно. Аналогичным образом Pig проверяет инструкции `GROUP` и `FOREACH...GENERATE` и добавляет их в логический план без исполнения. Исполнение начинается при обнаружении инструкции `DUMP`. В этой точке Pig преобразует логический план в физический и приступает к его исполнению.

МУЛЬТИЗАПРОСЫ

Так как команда `DUMP` является диагностическим инструментом, она всегда инициирует исполнение. С командой `STORE` дело обстоит иначе. В интерактивном режиме `STORE` работает как `DUMP` и всегда инициирует исполнение, а в пакетном режиме этого не происходит. Такое поведение объясняется соображениями эффективности. В пакетном режиме Pig разбирает весь сценарий и смотрит, нельзя ли применить какие-либо оптимизации для ограничения объема данных, читаемых или записываемых на диск. Например:

```
A = LOAD 'input/pig/multiquery/A';
B = FILTER A BY $1 == 'banana';
C = FILTER A BY $1 != 'banana';
STORE B INTO 'output/b';
STORE C INTO 'output/c';
```

Отношения `B` и `C` являются производными от `A`. Чтобы избежать повторного чтения `A`, Pig выполняет этот сценарий как одно задание MapReduce; `A` читается один раз, а задание записывает два выходных файла, для `B` и `C`. Эта особенность называется исполнением мультизапроса.

В предыдущих версиях Pig мультизапросы не поддерживались, и каждая инструкция STORE в сценарии, выполняемом в пакетном режиме, инициировала выполнение, в результате чего для каждой инструкции STORE генерировалось задание. Чтобы вернуться к старому поведению, следует отключить выполнение мультизапросов, выполнив команду `pig` с параметром `-M` или `-no_multiquery`.

Физический план, подготовленный Pig, состоит из серии заданий MapReduce, которые Pig в локальном режиме выполняет на локальной JVM, а в режиме MapReduce — в кластере Hadoop.



Чтобы просмотреть логический и физический планы, созданные Pig, выполните команду EXPLAIN для отношения (например, EXPLAIN max_temp;). EXPLAIN также выводит план MapReduce с физическими операторами, сгруппированными по заданиям MapReduce. Это хороший способ узнать, сколько заданий MapReduce выполнит Pig для вашего запроса.

Реляционные операторы (операторы отношений), которые могут быть частью логического плана в Pig, перечислены в табл. 11.1. Операторы более подробно рассматриваются в разделе «Операторы обработки данных», с. 508.

Таблица 11.1. Реляционные операторы Pig

Категория	Оператор	Описание
Загрузка и сохранение	LOAD	Загружает данные из файловой системы или другого хранилища в отношение
	STORE	Сохраняет отношение в файловой системе или другом хранилище
	DUMP	Выводит отношение на консоль
Фильтрация	FILTER	Удаляет лишние строки из отношения
	DISTINCT	Удаляет повторяющиеся строки из отношения
	FOREACH.. GENERATE	Добавляет или удаляет поля из отношения
	MAPREDUCE	Выполняет задание MapReduce, использующее отношение как входные данные

продолжение ↗

Таблица 11.1 (продолжение)

Категория	Оператор	Описание
	STREAM	Преобразует отношение, используя внешнюю программу
	SAMPLE	Формирует случайную выборку из отношения
Группировка и соединение	JOIN	Объединяет два и более отношения
	COGROUP	Группирует данные в двух и более отношениях
	GROUP	Группирует данные в одном отношении
	CROSS	Создает перекрестное соединение двух и более отношений
Сортировка	ORDER	Сортирует отношение по одному или нескольким полям
	LIMIT	Ограничивает размер отношения (максимальное количество кортежей)
Комбинирование и разбиение	UNION	Преобразует два и более отношения в одно
	SPLIT	Разбивает отношение на два и более отношения

Существуют другие виды инструкций, не включаемые в логический план. Например, диагностические операторы — `DESCRIBE`, `EXPLAIN` и `ILLUSTRATE` — позволяют пользователю взаимодействовать с логическим планом в целях отладки (табл. 11.2). Оператор `DUMP` тоже можно отнести к диагностическим, потому что он используется только для интерактивной отладки небольшого набора данных или в сочетании с `LIMIT` для выборки нескольких строк из большего отношения. Если размер отношения превышает несколько строк, следует использовать инструкцию `STORE+`, которая выводит данные в файл вместо направления их на консоль.

Таблица 11.2. Диагностические операторы Pig Latin

Оператор	Описание
DESCRIBE	Выводит схему отношения
EXPLAIN	Выводит логический и физический план
ILLUSTRATE	Выводит результат исполнения логического плана для сгенерированного подмножества входных данных

Pig Latin также предоставляет три инструкции — REGISTER, DEFINE и IMPORT, которые позволяют внедрять в сценарии Pig макросы и пользовательские функции (табл. 11.3).

Таблица 11.3. Инструкции Pig Latin для работы с макросами и пользовательскими функциями

Инструкция	Описание
REGISTER	Регистрирует файл JAR в исполнительной среде Pig
DEFINE	Создает псевдоним для макроса, пользовательской функции, сценария Streaming или спецификации команды
IMPORT	Импортирует макросы, определенные в отдельном файле, в сценарий

Команды не обрабатывают отношения, поэтому они не включаются в логический план; вместо этого они выполняются немедленно. Pig предоставляет команды для взаимодействия с файловыми системами Hadoop (такие команды очень удобны для перемещения данных до или после обработки Pig) и MapReduce, а также несколько вспомогательных команд (табл. 11.4).

Таблица 11.4. Команды Pig Latin

Категория	Команда	Описание
Файловая система Hadoop	cat	Выводит содержимое одного или нескольких файлов
	cd	Изменяет текущий каталог
	copyFromLocal	Копирует локальный файл или каталог в файловую систему Hadoop
	copyToLocal	Копирует файл или каталог Hadoop в локальную файловую систему
	cp	Копирует файл или каталог в другой каталог
	fs	Обращается к оболочке файловой системы Hadoop
	ls	Выдает список файлов
	mkdir	Создает новый каталог
	mv	Перемещает файл или каталог в другой каталог

продолжение ↗

Таблица 11.4 (продолжение)

Категория	Команда	Описание
Hadoop Map-Reduce	pwd	Выводит путь к текущему рабочему каталогу
	rm	Удаляет файл или каталог
	rmf	Форсированно удаляет файл или каталог (без ошибки, если файл или каталог не существует)
Hadoop Map-Reduce	kill	Уничтожает задание MapReduce
Служебные команды	exec	Выполняет сценарий в новом экземпляре Grunt в пакетном режиме
	help	Выводит сводку всех доступных команд и параметров
	quit	Завершает работу в интерпретаторе
	run	Выполняет сценарий в существующем экземпляре Grunt
	set	Задает параметры Pig и свойства задания MapReduce
	sh	Выполняет команду оболочки из Grunt

Команды файловой системы могут выполнять операции с файлами или каталогами в любой файловой системе Hadoop; они очень похожи на команды *hadoop fs* (что неудивительно, так как обе разновидности команд представляют собой простые обертки для интерфейса Hadoop `FileSystem`). Все команды оболочки файловой системы Hadoop могут выполняться с использованием команды Pig `fs`. Например, команда `fs -ls` выводит список файлов, а команда `fs -help` выводит справку по доступным командам.

Используемая файловая система Hadoop определяется свойством `fs.default.name` (см. «Интерфейс командной строки», с. 87).

Команды в основном не требуют пояснений — кроме команды `set`, которая используется для задания параметров, управляющих поведением Pig, включая произвольные свойства заданий MapReduce. Параметр `debug` используется для включения и отключения ведения журнала отладки из сценария (уровень ведения журнала также можно указать при запуске Pig при помощи параметра `-d` или `-debug`):

```
grunt>
  set debug on
```

Другой полезный параметр, `job.name`, дает заданию Pig содержательное имя, упрощающее поиск ваших заданий Pig MapReduce в общем кластере Hadoop. Если Pig выполняет сценарий (вместо выполнения интерактивного запроса из Grunt), имя задания по умолчанию определяется по имени сценария.

В табл. 11.4 представлены две команды выполнения сценариев Pig: `exec` и `run`. Команда `exec` выполняет сценарий в пакетном режиме в новом экземпляре Grunt, поэтому любые псевдонимы, определенные в сценарии, становятся недоступными после его завершения. Вместе с тем при выполнении сценария командой `run` все происходит так, словно содержимое сценария было введено вручную, а в истории команд сохраняются все инструкции сценария. Режим исполнения мультизапроса, при котором Pig выполняет пакет инструкций (см. «Мультизапросы», с. 480), используется только командой `exec`, но не командой `run`.



В Pig Latin намеренно не были включены встроенные инструкции управления потоком выполнения. В программах, требующих условной логики или циклических конструкций, рекомендуется встраивать код Pig Latin в другой язык (например, Python, JavaScript или Java), и управлять потоком команд извне. В этой модели внешний сценарий использует API Compile-Bind-Run для исполнения сценариев Pig и получения информации об их состоянии. За подробной информацией об API обращайтесь к документации Pig.

Встроенные программы Pig всегда выполняются в JVM, поэтому для сценариев Python и JavaScript используется команда Pig с именем вашего сценария, а для нее выбирается подходящее сценарное ядро Java (Jython для Python, Rhino для JavaScript).

Выражение

Выражение представляет собой конструкцию, результатом вычисления которой является значение. Выражения могут использоваться в Pig в командах, содержащих реляционные операторы. Pig поддерживает разнообразные выражения, многие из которых могут быть знакомы вам по другим языкам программирования. Выражения с краткими описаниями и примерами перечислены в табл. 11.5. Примеры выражений неоднократно встречаются нам в этой главе.

Таблица 11.5. Выражения Pig Latin

Категория	Выражения	Описание	Примеры
Константа	Литерал	Постоянное значение (также см. столбец «Примеры литералов» в табл. 11.6)	1.0, 'a'

продолжение ↗

Таблица 11.5 (продолжение)

Категория	Выражения	Описание	Примеры
Поле (заданное позицией)	\$n	Поле в позиции n (нумерация начинается с нуля)	\$0
Поле (заданное именем)	f	Поле с именем f	year
Поле (уточненное)	r::f	Поле с именем f из отношения r после группировки или соединения	A::year
Проекция	c.\$n, c.f	Поле в контейнере c (отношение, неупорядоченное множество или кортеж), заданное позицией или именем	records.\$0, records.year
Поиск по ключу в отображении	m#k	Значение, связанное с ключом k в отображении m	items#'Coat'
Преобразование типа	(t) f	Преобразование поля f к типу t	(int) year
Арифметические выражения	x + y, x - y	Сложение, вычитание	\$1 + \$2, \$1 - \$2
	x * y, x / y	Умножение, деление	\$1 * \$2, \$1 / \$2
	x % y	Остаток от деления x на y	\$1 % \$2
	+x, -x	Унарный плюс или минус	+1, -1
Условные выражения	x ? y : z	Тернарный оператор: если значение x истинно, то y, а если ложно — z	quality == 0 ? 0 : 1
Сравнение	x == y, x != y	Равно, не равно	quality == 0, temperature != 9999
	x > y, x < y	Больше, меньше	quality > 0, quality < 10
	x >= y, x <= y	Больше либо равно, меньше либо равно	quality >= 1, quality <= 9
	x matches y	Поиск по регулярному выражению	quality matches '[01459]'
	x is null	Равно null	temperature is null
	x is not null	Не равно null	temperature is not null

Категория	Выражения	Описание	Примеры
Логические выражения	x or y	Логическая операция «или»	q == 0 or q == 1
	x and y	Логическая операция «и»	q == 0 and r == 0
	not x	Логическое отрицание	not q matches '[01459]'
Функциональное выражение	fn(f1,f2,...)	Вызов функции fn для полей f1, f2 и т. д.	isGood(quality)
Уплощение	FLATTEN(f)	Удаление вложенности из неупорядоченных множеств и кортежей	FLATTEN(group)

Типы

Ранее мы уже видели некоторые простые типы Pig — такие, как `int` и `chararray`. В этом разделе встроенные типы Pig будут рассмотрены более подробно.

В Pig существуют четыре числовых типа: `int`, `long`, `float` и `double`; они идентичны своим аналогам из Java. Также определен тип `bytearray` для представления блока двоичных данных и тип `chararray`, который, как и `java.lang.String`, представляет текстовые данные в формате UTF-16 (хотя данные могут загружаться или храниться в формате UTF-8). В Pig нет типов, соответствующих примитивным типам Java `boolean`¹, `byte`, `short` или `char`. Все они легко представляются при помощи типа Pig `int` или `chararray` (для `char`).

Числовые, текстовые и двоичные типы относятся к категории простых атомарных типов. В Pig Latin также существует три составных типа для представления вложенных структур: кортежи, неупорядоченные множества и отображения. Список типов Pig Latin приведен в табл. 11.6.

Составные типы обычно загружаются из файлов или конструируются с использованием реляционных операторов. Однако учтите, что лiteralная форма в табл. 11.6 используется при создании констант в программах на Pig Latin. Низкоуровневое представление данных в файле обычно отличается при использовании стандартного загрузчика `PigStorage`. Например, представление неупорядоченного

¹ Несмотря на отсутствие логического типа данных (до версии 0.10.0), в Pig существует концепция истинности или ложности выражения при проверке условия (например, в инструкции `FILTER`). При этом Pig не позволяет сохранить результат логического выражения в поле.

множества из табл. 11.6 в файле будет иметь вид `{(1, pomegranate), (2)}` (обратите внимание на отсутствие кавычек), и с подходящей схемой оно будет загружено как отношение с одним полем и строкой.

Таблица 11.6. Типы Pig Latin

Категория	Тип	Описание	Пример литерала
Числовые типы	int	32-разрядное целое число со знаком	1
	long	64-разрядное целое число со знаком	1L
	float	32-разрядное вещественное число	1.0F
	double	64-разрядное вещественное число	1.0
Текстовые типы	chararray	Символьный массив в формате UTF-16	'a'
Двоичные типы	bytearray	Байтовый массив	—
Составные типы	tuple	Последовательность полей любого типа	(1,'pomegranate')
	bag	Неупорядоченное множество кортежей; может содержать дубликаты	<code>{(1,'pomegranate'),(2)}</code>
	map	Множество пар «ключ-значение»; ключи должны быть символьными массивами, значения могут относиться к произвольному типу	<code>['a' #'pomegranate']</code>

Pig предоставляет встроенные функции `TOTUPLE`, `TOBAG` и `TOMAP` для преобразования выражений в кортежи, неупорядоченные множества и отображения соответственно.

Хотя отношения и неупорядоченные множества концептуально эквивалентны (и то, и другое — неупорядоченные совокупности кортежей), на практике Pig рассматривает их по-разному. Отношение представляет собой конструкцию верхнего уровня, а неупорядоченное множество должно содержаться в отношении. Обычно разработчику не приходится об этом думать, но существуют некоторые ограничения, которые могут преподнести сюрпризы новичкам. Например, невозможно

создать отношение из литерала неупорядоченного множества. Таким образом, попытка выполнения следующей инструкции завершается неудачей:

```
A = {(1,2),(3,4)}; -- Error
```

Простейшее обходное решение в данном случае – загрузка данных из файла инструкцией **LOAD**.

Или другой пример: отношение невозможно интерпретировать как неупорядоченное множество с проецированием поля в новое отношение (**\$0** – обозначение первого поля **A** в позиционной записи):

```
B = A.$0;
```

Вместо этого придется воспользоваться реляционным оператором для преобразования отношения **A** в отношение **B**:

```
B = FOREACH A GENERATE $0;
```

Возможно, в будущих версиях Pig Latin эти расхождения будут устраниены, и отношения и неупорядоченные множества будут интерпретироваться одинаково.

Схемы

Отношение в Pig может быть ассоциировано со схемой, определяющей имена и типы полей отношения. Мы уже видели, как секция **AS** в инструкции **LOAD** используется для связывания схемы с отношением:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'  
>> AS (year:int, temperature:int, quality:int);  
grunt> DESCRIBE records;  
records: {year: int,temperature: int,quality: int}
```

На этот раз год объявляется с типом **int** вместо **chararray**, хотя файл, из которого загружаются данные, остался прежним. Целое число лучше подходит для арифметических операций с годом (например, для его преобразования во временную метку), тогда как представление **chararray** может оказаться более уместным для использования в качестве простого идентификатора. Гибкость Pig в объявлении схем выгодно отличается от схем традиционных баз данных SQL, которые должны быть объявлены до загрузки данных в систему. Система Pig проектировалась для анализа простых входных файлов, не ассоциированных с информацией типов, поэтому вполне естественно, что типы полей Pig могут выбираться на более поздней стадии, чем при работе с РСУБД.

Объявления типов также можно полностью опустить:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature, quality);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

Здесь указаны только имена полей схемы: `year`, `temperature` и `quality`. По умолчанию используются типы `bytearray` — самый общий тип, представляющий собой двоичную строку.

Указывать типы для каждого поля не обязательно; некоторым типам можно оставить используемый по умолчанию тип байтового массива, как для поля `year` в следующем объявлении:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

Однако если схема задается подобным образом, задавать придется каждое поле. Кроме того, невозможно задать тип поля без указания имени. Вместе с тем наличие схемы не обязательно — схему можно не указывать, опустив секцию `AS`:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

Для обозначения полей отношений, не имеющих схемы, может использоваться только позиционная запись: `$0` обозначает первое поле отношения, `$1` — второе, и так далее. По умолчанию поля относятся к типу `bytearray`:

```
grunt> projected_records = FOREACH records GENERATE $0, $1, $2;
grunt> DUMP projected_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

Иногда (особенно на ранних стадиях написания запроса) бывает удобнее использовать поля без типов. Тем не менее назначение типов повышает наглядность и эффективность программ Pig Latin, поэтому обычно рекомендуется использовать типы.



Объявление схемы как части запроса повышает его гибкость, но не способствует повторному использованию схемы. Наборы запросов Pig к одним данным часто используют одну схему, повторяющуюся для каждого запроса. Если запрос обрабатывает большое количество полей, это усложнит сопровождение кода с повторениями.

Проект Apache HCatalog (<http://incubator.apache.org/hcatalog/>) для решения этой проблемы предоставляет сервис метаданных таблиц на основе метахранилища Hive, так что запросы Pig могут ссылаться на схемы по именам вместо того, чтобы каждый раз полностью определять их заново.

Проверка и null

База данных SQL проверяет ограничения, заданные схемой таблицы, во время загрузки; например, попытка загрузить строку в столбец, объявленный с числовым типом, завершится неудачей. Если значение не может быть преобразовано к типу, объявленному в схеме, Pig заменяет его `null`. Предположим, в наших метеорологических данных целое число заменено символом «е»:

```
1950    0    1
1950    22   1
1950    e    1
1949    111  1
1949    78   1
```

При обработке поврежденной строки Pig подставляет вместо некорректного значения значение `null`, которое не отображается при выводе на экран (а также при сохранении оператором `STORE`):

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>> AS (year:chararray, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,,1)
(1949,111,1)
(1949,78,1)
```

Pig выводит предупреждение для недействительного поля (в листинге оно не приведено), но не прерывает обработку. В больших базах данных очень часто встречаются поврежденные, недействительные или просто непредвиденные данные, и в общем случае последовательное исправление всех ошибок разбора записей неприемлемо. Вместо этого можно выделить все недействительные записи за один

проход, чтобы с ними что-то сделать (например, внести изменения в программу, потому что такие записи являются признаком ошибки) или отфильтровать (потому что данные непригодны к использованию):

```
grunt> corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
```

Обратите внимание на оператор `is null`, аналогичный оператору SQL. На практике в вывод следует включить дополнительную информацию из исходной записи — например, идентификатор и значение, которое не удалось обработать, для упрощения анализа непригодных данных.

Для определения количества поврежденных записей можно воспользоваться следующей идиомой для подсчета строк в отношении:

```
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group, COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1)
```

(В разделе «GROUP» на с. 517 группировка и операция ALL рассматриваются более подробно.)

Другой полезный прием — использование оператора `SPLIT` для разбиения данных на два отношения, «хорошее» и «плохое», для последующего раздельного анализа:

```
grunt> SPLIT records INTO good_records IF temperature is not null,
>>   bad_records IF temperature is null;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

В ситуации, когда тип `temperature` оставался необъявленным, поврежденные данные не удавалось найти так легко, поскольку они не заменялись `null`:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>>   AS (year:chararray, temperature, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,e,1)
(1949,111,1)
(1949,78,1)
```

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>> (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grunt> grouped_records = GROUP filtered_records BY year;
grunt> max_temp = FOREACH grouped_records GENERATE group,
>>   MAX(filtered_records.temperature);
grunt> DUMP max_temp;
(1949,111.0)
(1950,22.0)
```

В этом случае поле `temperature` интерпретируется как `bytearray`, поэтому поврежденные данные не обнаруживаются в ходе загрузки. При передаче функции `MAX` поле `temperature` преобразуется в `double`, так как `MAX` работает только с числовыми типами. Поврежденное поле не может быть представлено в виде `double`; оно преобразуется в значение `null`, которое `MAX` просто игнорирует. Как правило, типы данных следует объявлять при загрузке, искать отсутствующие или поврежденные значения в самих отношениях перед выполнением основной обработки.

Иногда поврежденные данные из-за отсутствия полей представляются кортежами меньшего размера. Такие записи можно отфильтровать при помощи функции `SIZE`:

```
grunt> A = LOAD 'input/pig/corrupt/missing_fields';
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3)
(1,Scarf)
grunt> B = FILTER A BY SIZE(TOTUPLE(*)) > 1;
grunt> DUMP B;
(2,Tie)
(4,Coat)
(1,Scarf)
```

Слияние схем

В Pig разработчик не объявляет схему для каждого нового отношения в потоке данных. В большинстве случаев Pig может вычислить новую схему, полученную в результате реляционной операции, по схеме входного отношения.

Как схемы распространяются на новые отношения? Некоторые реляционные операторы не изменяют схему; например, отношение, созданное оператором `LIMIT` (ограничивающим максимальное количество кортежей), имеет точно такую же схему, как и отношение, к которому оно было применено. Для других операторов ситуация становится более сложной. Например, оператор `UNION` объединяет два и более отношения и пытается выполнить слияние схем входных отношений.

Если схемы несовместимы из-за различий в типах или количестве полей, то схема результата `UNION` останется неизвестной.

Схема любого отношения в потоке данных может быть выведена оператором `DESCRIBE`. Если вы захотите переопределить схему некоторых или всех полей входного отношения, используйте оператор `FOREACH...GENERATE` с секцией `AS`.

Схемы также рассматриваются в разделе «Пользовательские функции», с. 498.

Функции

Функции Pig делятся на четыре категории:

Вычисляющая функция

Функция получает одно или несколько выражений и возвращает другое выражение. Примером вычисляющей функции служитстроенная функция `MAX`, которая возвращает максимальное значение среди элементов неупорядоченного множества. Некоторые вычисляющие функции являются *агрегатными* функциями, то есть вычисляют скалярное значение для неупорядоченного множества данных; функция `MAX` также является агрегатной. Кроме того, многие агрегатные функции являются *алгебраическими*, то есть результат функции может вычисляться методом приращений. В контексте MapReduce алгебраические функции используют комбинирующую функцию и вычисляются намного эффективнее (см. «Комбинирующие функции», с. 68). Функция `MAX` также является алгебраической функцией, тогда как функция вычисления среднего значения выборки по набору значений алгебраической не является.

Фильтрующая функция

Особая разновидность вычисляющих функций, возвращающая логический результат. Как подсказывает название, фильтрующие функции используются в операторе `FILTER` для исключения нежелательных строк. Они также могут использоваться в других реляционных операторах, получающих логическое условие, а в общем случае в выражениях, использующих логические или условные подвыражения. Пример встроенной фильтрующей функции — `IsEmpty`; эта функция проверяет, что неупорядоченное множество или отображение не содержит ни одного элемента.

Функция загрузки

Функция, указывающая, как должна осуществляться загрузка данных в отношение из внешнего хранилища.

Функция сохранения

Функция, указывающая, как содержимое отношения должно сохраняться во внешнем хранилище. Функции загрузки и сохранения часто реализуются одним типом. Например, тип **PigStorage**, загружающий данные из текстовых файлов с разделителями, может сохранять данные в том же формате.

Pig содержит набор встроенных функций, часть из которых представлена в табл. 11.7. Полный список встроенных функций с многочисленными стандартными математическими и строковыми функциями можно найти в документации к выпуску Pig.

Таблица 11.7. Подборка встроенных функций Pig

Категория	Функция	Описание
Вычисляю-щие	AVG	Вычисляет среднее арифметическое значений элементов неупорядоченного множества
	CONCAT	Выполняет конкатенацию байтовых или символьных массивов
	COUNT	Вычисляет количество элементов неупорядоченного множества, отличных от null
	COUNT_STAR	Вычисляет количество элементов неупорядоченного множества, включая равные null
	DIFF	Вычисляет разность двух неупорядоченных множеств
	MAX	Вычисляет максимальное значение среди элементов неупорядоченного множества
	MIN	Вычисляет минимальное значение среди элементов неупорядоченного множества
	SIZE	Вычисляет размер типа. Размер числовых типов всегда равен 1; для символьных он равен количеству символов; для байтовых массивов он равен количеству байт; для контейнеров (кортежи, неупорядоченные множества, отображения) он равен количеству элементов
	SUM	Вычисляет сумму значений элементов неупорядоченного множества
	TOBAG	Преобразует одно или несколько выражений в кортежи, которые помещаются в неупорядоченное множество
	TOKENIZE	Преобразует символьный массив в неупорядоченное множество входящих в него слов

продолжение ↗

Таблица 11.7 (продолжение)

Категория	Функция	Описание
	TOMAP	Преобразует четное количество выражений в отображение пар «ключ-значение»
	TOTUPLE	Преобразует одно или несколько значений в кортеж
Фильтрующие	IsEmpty	Проверяет, является ли неупорядоченное множество или отображение пустым
Загрузка/сохранение	PigStorage	Загружает или сохраняет отношения в формате текстовых полей с разделителями. Каждая строка разбивается на поля по настраиваемым разделителям (по умолчанию используется символ табуляции), а поля сохраняются в полях кортежа. Этот механизм сохранения используется по умолчанию
	BinStorage	Загружает или сохраняет отношения в двоичных файлах в формате, использующем объекты Hadoop Writable
	TextLoader	Загружает отношения в формате простого текста. Каждая физическая строка соответствует кортежу, единственное поле которого содержит текст строки
	JsonLoader, JsonStorage	Загружает или сохраняет отношения в формате JSON (определяемом Pig). Каждый кортеж сохраняется в одной строке
	HBaseStorage	Загружает или сохраняет отношения в таблицах HBase

Если нужной вам встроенной функции не существует, напишите собственную функцию. Однако перед этим загляните в Piggy Bank – репозиторий функций Pig, используемых сообществом Pig. Например, в Piggy Bank имеются функции загрузки и сохранения для файлов данных Avro, файлов CSV (разделенных запятыми), файлов Hive RCFile, SequenceFile и XML. Инструкции по просмотру Piggy Bank и получению функций приведены на веб-сайте Pig. Если нужной функции нет и в Piggy Bank, напишите ее самостоятельно (а если функция достаточно универсальна, внесите ее в Piggy Bank, чтобы и другие могли ею пользоваться). Такие функции называются пользовательскими функциями, или UDF (User-Defined Functions).

Макросы

Макросы предоставляют возможность упаковки фрагментов кода Pig Latin, пригодных для повторного использования, непосредственно из Pig Latin. Например,

можно выделить фрагмент программы Pig Latin, который выполняет группировку отношения, а затем находит максимальное значение в каждой группе; макрос определяется следующим образом:

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {  
    A = GROUP $X by $group_key;  
    $Y = FOREACH A GENERATE group, MAX($X.$max_field);  
};
```

Этот макрос, которому присвоено имя `max_by_group`, получает три параметра: отношение, X и два имени полей, `group_key` и `max_field`. Он возвращает одно отношение Y. В теле макроса для ссылок на параметры и возвращаемое значение используется префикс \$ (например, \$X).

Пример использования макроса:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'  
    AS (year:chararray, temperature:int, quality:int);  
filtered_records = FILTER records BY temperature != 9999 AND  
    (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);  
max_temp = max_by_group(filtered_records, year, temperature);  
DUMP max_temp
```

Во время выполнения Pig подставляет код макроса из его определения в место вызова. После расширения программа принимает следующий вид (расширенный раздел выделен жирным шрифтом).

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'  
    AS (year:chararray, temperature:int, quality:int);  
filtered_records = FILTER records BY temperature != 9999 AND  
    (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);  
macro_max_by_group_A_0 = GROUP filtered_records by (year);  
max_temp = FOREACH macro_max_by_group_A_0 GENERATE group,  
    MAX(filtered_records.(temperature));  
DUMP max_temp
```

Обычно развернутая форма макроса остается скрытой, потому что Pig создает ее во внутреннем представлении; однако в некоторых случаях бывает полезно просмотреть ее при написании и отладке макросов. Чтобы приказать Pig только выполнить подстановку макросов (без выполнения сценария), вызовите `pig` с аргументом `-dryrun`.

Обратите внимание: переданные макросу параметры (`filtered_records`, `year` и `temperature`) в определении макроса заменены именами. Псевдонимы в определении макроса, не имеющие префикса \$ (как A в нашем примере), локальны для определения макроса; при подстановке они заменяются расширенными именами

для предотвращения конфликтов с псевдонимами в других частях программы. В нашем примере `A` в расширенной форме превращается в `macro_max_by_group_A_0`.

Для упрощения повторного использования макросы можно определить в отдельных файлах; в этом случае их необходимо импортировать в сценарий, в котором они используются:

```
IMPORT './ch11/src/main/pig/max_temp.macro';
```

Пользовательские функции

Проектировщики Pig понимали, что способность подключения пользовательского кода абсолютно необходима при любой обработке данных, за исключением разве что самых тривиальных задач. По этой причине они постарались упростить определение и использование пользовательских функций. В этом разделе рассматриваются только пользовательские функции Java, однако функции также можно писать на Python и JavaScript; и те, и другие выполняются через Java Scripting API.

Фильтрующая пользовательская функция

Давайте напишем фильтрующую функцию для исключения метеорологических записей, у которых качество температуры ниже удовлетворительного. Мы хотим заменить эту строку:

```
filtered_records = FILTER records BY temperature != 9999 AND  
    (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
```

следующей строкой:

```
filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
```

Замена преследует две цели: во-первых, сценарий Pig становится более компактным, а во-вторых, логика инкапсулируется в одном месте, что упрощает ее последующее использование в других сценариях. При написании простого одноразового запроса, вероятно, мы не стали бы тратить время на написание пользовательской функции. Возможности использования многоразовых функций открываются только тогда, когда одна и та же обработка выполняется снова и снова.

Фильтрующие пользовательские функции являются субклассами `FilterFunc`, который сам является субклассом `EvalFunc`. Класс `EvalFunc` более подробно рассматривается позднее, а сейчас достаточно сказать, что по сути `EvalFunc` выглядит примерно так:

```
public abstract class EvalFunc<T> {  
    public abstract T exec(Tuple input) throws IOException;  
}
```

Единственный абстрактный метод `EvalFunc` — `exec()` — получает кортеж и возвращает одно значение (параметризованного) типа `T`. Поля входного кортежа состоят из выражений, переданных функции, — в нашем примере это единственное целое число. Для `FilterFunc` тип `T` является логическим, так что метод должен возвращать `true` только для тех кортежей, которые не должны быть отфильтрованы.

Для фильтрации мы напишем класс `IsGoodQuality`, который расширяет `FilterFunc` и реализует метод `exec()`. Код класса приведен в листинге 11.1. Класс кортежа `Tuple` фактически представляет собой список объектов со связанными типами. Нас интересует только первое поле (так как функция имеет всего один аргумент), которое извлекается по индексу вызовом метода `get()` для `Tuple`. Поле содержит целое число; если оно отлично от `null`, мы преобразуем его, проверяем, соответствует ли температура минимальному качеству и возвращаем результат проверки `true` или `false`:

Листинг 11.1. Пользовательская реализация `FilterFunc` для удаления записей с недостаточным уровнем качества температуры

```
package com.hadoopbook.pig;  
  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.List;  
  
import org.apache.pig.FilterFunc;  
  
import org.apache.pig.backend.executionengine.ExecException;  
import org.apache.pig.data.DataType;  
import org.apache.pig.data.Tuple;  
import org.apache.pig.impl.logicalLayer.FrontendException;  
  
public class IsGoodQuality extends FilterFunc {  
  
    @Override  
    public Boolean exec(Tuple tuple) throws IOException {  
        if (tuple == null || tuple.size() == 0) {  
            return false;  
        }  
        try {  
            Object object = tuple.get(0);  
        }
```

продолжение ↗

Листинг 11.1 (продолжение)

```
if (object == null) {  
    return false;  
}  
int i = (Integer) object;  
return i == 0 || i == 1 || i == 4 || i == 5 || i == 9;  
} catch (ExecException e) {  
    throw new IOException(e);  
}  
}  
}
```

}

Чтобы использовать новую функцию, следует сначала откомпилировать ее и упаковать в файл JAR (инструкции приведены в примерах кода, прилагаемых к книге). Затем мы сообщаем Pig о существовании файла JAR оператором REGISTER, которому передается локальный путь к файлу (не заключенный в кавычки):

```
grunt> REGISTER pig-examples.jar;
```

Наконец, мы вызываем функцию:

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND  
>> com.hadoopbook.pig.IsGoodQuality(quality);
```

Pig разрешает вызовы функции, интерпретируя имя функции как имя класса Java, и пытаясь загрузить класс с заданным именем. (Кстати, именно по этой причине в именах функций различается регистр символов: потому что он различается в именах классов Java). При поиске классов Pig использует загрузчик классов, включающий в себя зарегистрированные файлы JAR. При выполнении в распределенном режиме Pig обеспечит пересылку ваших файлов JAR в кластер.

Для пользовательской функции из нашего примера Pig ищет класс с именем `com.hadoopbook.pig.IsGoodQuality`, который он находит в зарегистрированном файле JAR.

Разрешение встроенных функций осуществляется аналогичным образом с одним различием: у Pig имеется набор имен встроенных пакетов, в которых он проводит поиск, поэтому вызов функции не обязан содержать полностью уточненное имя. Например, функция MAX в действительности реализуется классом `MAX` из пакета `org.apache.pig.builtin`. Этот пакет входит в число тех, в которых Pig проводит поиск, поэтому мы можем использовать в программах Pig короткое имя `MAX` вместо полного имени `org.apache.pig.builtin.MAX`.

Чтобы включить имя своего пакета в путь поиска, передайте Grunt аргумент командной строки `-Dudf.import.list=com.hadoopbook.pig`. Также для сокращения имени функции можно определить псевдоним оператором `DEFINE`:

```
grunt> DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
grunt> filtered_records = FILTER records BY temperature != 9999 AND
    isGood(quality);
```

Псевдоним будет полезен при многократном использовании функции в одном сценарии. Кроме того, он необходимо, если вы собираетесь передавать аргументы конструктору в классе реализации пользовательской функции.

Использование типов

Фильтр работает, если поле `quality` объявлено с типом `int`, но при отсутствии информации типа попытка использования пользовательской функции приводит к ошибке! Это происходит из-за того, что по умолчанию полю назначается тип `bytearray`, представленный классом `DataByteArray`. Так как класс `DataByteArray` не является `Integer`, попытка преобразования типа завершается неудачей.

Наиболее очевидное решение — преобразование поля в целое число методом `exec()`. Однако лучше поступить иначе: сообщить Pig типы полей, ожидаемых функцией. Метод `getArgToFuncMapping()` класса `EvalFunc` предназначен именно для этой цели. Переопределив его, мы можем сообщить Pig, что первое поле должно быть целым числом:

```
@Override
public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
    List<FuncSpec> funcSpecs = new ArrayList<FuncSpec>();
    funcSpecs.add(new FuncSpec(this.getClass().getName(),
        new Schema(new Schema.FieldSchema(null, DataType.INTEGER)))); 
    return funcSpecs;
}
```

Метод возвращает объект `FuncSpec`, соответствующий каждому из полей кортежа, переданного методу `exec()`. В нашем случае поле всего одно, и мы конструируем анонимный объект `FieldSchema` (вместо имени передается `null`, так как Pig игнорирует имена при преобразовании типов). Для задания типа используется константа `INTEGER` класса Pig `DataType`.

После усовершенствования функции Pig пытается преобразовать аргумент, переданный функции, в целое число. Если преобразование невозможно, передается `null`. Метод `exec()` всегда возвращает `false` для полей `null`. В нашем приложении такое поведение уместно, так как мы хотим отфильтровать записи с некорректными значениями в поле качества.

Пример программы, использующей новую версию функции:

```
-- max_temp_filter_udf.pig
REGISTER pig-examples.jar;
```

продолжение ↗

```

DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
records = LOAD 'input/ncdc/micro-tab/sample.txt'
    AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
    MAX(filtered_records.temperature);
DUMP max_temp;

```

Вычисляющая пользовательская функция

Следующим шагом после фильтрующей функции станет написание вычисляющей функции. Возьмем пользовательскую функцию, которая удаляет начальные и конечные пропуски из значений `chararray`, как метод `trim()` класса `java.lang.String` (листинг 11.2). Мы используем эту функцию позднее в этой главе.

Листинг 11.2. Пользовательская реализация `EvalFunc` для удаления начальных и конечных пропусков из значений `chararray`

```

public class Trim extends EvalFunc<String> {

    @Override
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0) {
            return null;
        }
        try {
            Object object = input.get(0);
            if (object == null) {
                return null;
            }
            return ((String) object).trim();
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }

    @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;
    }
}

```

Вычисляющая функция расширяет класс `EvalFunc`, параметризованный по типу возвращаемого значения (`String` для `Trim`)¹. Методы `exec()` и `getArgToFuncMapping()` достаточно тривиальны, как и аналогичные методы функции `IsGoodQuality`.

При написании вычисляющей функции необходимо учесть, как выглядит схема результата. В следующей инструкции схема `B` определяется функцией `udf`:

```
B = FOREACH A GENERATE udf($0);
```

Если `udf` создает кортежи со скалярными полями, Pig может определить схему `B` посредством рефлексии (reflection). Для составных типов — неупорядоченных множеств, кортежей, отображений — Pig потребуется дополнительная информация. Вам придется реализовать метод `outputSchema()`, чтобы передать Pig информацию о выходной схеме.

Пользовательская функция `Trim` возвращает строку, которую Pig преобразует в `chararray`, как видно из следующего листинга:

```
grunt> DUMP A;
(pomegranate)
(banana)
(apple)
( lychee )
grunt> DESCRIBE A;
A: {fruit: chararray}
grunt> B = FOREACH A GENERATE com.hadoopbook.pig.Trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
grunt> DESCRIBE B;
B: {chararray}
```

`A` содержит поля `chararray`, содержащие начальные и конечные пробелы. Мы создаем `B` из `A`, применяя функцию `Trim` к первому полю в `A` (с именем `fruit`). Pig правильно определяет, что поля `B` имеют тип `chararray`.

Динамический вызов

Иногда бывает нужно использовать функцию, предоставляемую библиотекой Java, но без хлопот с написанием пользовательской функции. Механизм динамического

¹ Хотя к нашему примеру это не относится, вычисляющие функции, работающие с неупорядоченными множествами, могут дополнительно реализовать интерфейсы Pig Algebraic или Accumulator для более эффективной фрагментарной обработки множества.

вызыва позволяет вызывать методы Java прямо из сценариев Pig. К сожалению, вызовы методов осуществляются посредством рефлексии, что приводит к значительным затратам ресурсов при вызове для каждой записи большого набора данных. Соответственно, в сценариях, предназначенных для многократного выполнения, лучше определить пользовательскую функцию.

Следующий фрагмент показывает, как определить и использовать пользовательскую функцию удаления пробелов на базе класса Apache Commons Lang `StringUtils`.

```
grunt> DEFINE trim InvokeForString('org.apache.commons.lang.StringUtils.trim',
                                     'String');
grunt> B = FOREACH A GENERATE trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
```

Так как возвращаемое значение метода имеет тип `String`, используется активатор `InvokeForString`. (Также существуют активаторы `InvokeForInt`, `InvokeForLong`, `InvokeForDouble` и `InvokeForFloat`.) Первый аргумент конструктора активатора содержит имя вызываемого метода, а второй — разделенный пробелами список классов аргументов.

Пользовательская функция загрузки

В этом разделе представлена пользовательская функция загрузки, которая читает диапазоны столбцов простого текста как поля (по аналогии с командой Unix `cut`). Пример ее использования:

```
grunt> records = LOAD 'input/ncdc/micro/sample.txt'
>> USING com.hadoopbook.pig.CutLoadFunc('16-19,88-92,93-93')
>> AS (year:int, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

Строка, переданная `CutLoadFunc`, содержит спецификацию столбцов; каждый из диапазонов, разделенных запятыми, определяет поле, которому присваивается имя и тип в секции `AS`. Рассмотрим реализацию `CutLoadFunc`, приведенную в листинге 11.3.

Листинг 11.3. Пользовательская реализация LoadFunc для загрузки полей кортежа из диапазонов столбцов

```
public class CutLoadFunc extends LoadFunc {

    private static final Log LOG = LogFactory.getLog(CutLoadFunc.class);

    private final List<Range> ranges;
    private final TupleFactory tupleFactory = TupleFactory.getInstance();
    private RecordReader reader;

    public CutLoadFunc(String cutPattern) {
        ranges = Range.parse(cutPattern);
    }

    @Override
    public void setLocation(String location, Job job)
        throws IOException {
        FileInputFormat.setInputPaths(job, location);
    }

    @Override
    public InputFormat getInputFormat() {
        return new TextInputFormat();
    }

    @Override
    public void prepareToRead(RecordReader reader, PigSplit split) {
        this.reader = reader;
    }

    @Override
    public Tuple getNext() throws IOException {
        try {
            if (!reader.nextKeyValue()) {
                return null;
            }
            Text value = (Text) reader.getCurrentValue();
            String line = value.toString();
            Tuple tuple = tupleFactory.newTuple(ranges.size());
            for (int i = 0; i < ranges.size(); i++) {
                Range range = ranges.get(i);
                if (range.getEnd() > line.length()) {
                    LOG.warn(String.format(
                        "Range end (%s) is longer than line length (%s)",
```

продолжение ↗

Листинг 11.3 (продолжение)

```
        range.getEnd(), line.length())));
    continue;
}
tuple.set(i, new DataByteArray(range.getSubstring(line)));
}
return tuple;
} catch (InterruptedException e) {
    throw new ExecException(e);
}
}
}
```

В Pig, как и в Hadoop, загрузка данных происходит до запуска задач отображения, поэтому желательно, чтобы входные данные могли быть разбиты на части, независимо обрабатываемые каждой задачей (см. «Входные сплиты и записи», с. 309).

Начиная с Pig 0.7.0, интерфейсы функций загрузки и сохранения были переработаны для согласования с классами Hadoop `InputFormat` и `OutputFormat`. Функции, написанные для предыдущих версий Pig, приходится переписывать (рекомендации по переработке приведены по адресу <http://wiki.apache.org/pig/LoadStoreMigrationGuide>). В реализации `LoadFunc` объект `InputFormat` обычно используется для создания записей, а `LoadFunc` предоставляет логику преобразования записей в кортежи Pig.

Экземпляр `CutLoadFunc` конструируется строкой, задающей диапазоны столбцов для каждого поля. Логика разбора строки и создания списка внутренних объектов `Range`, в которых инкапсулируются диапазоны, содержится в классе `Range` и здесь не приводится (обращайтесь к примерам кода, прилагаемым к книге).

Pig вызывает `setLocation()` для `LoadFunc`, чтобы передать загрузчику местонахождение входных данных. Так как `CutLoadFunc` использует `TextInputFormat` для разбиения входных данных на строки, мы просто используем статический метод для `FileInputFormat`.



Pig использует новый MapReduce API, поэтому в коде используются входные/выходные форматы и сопутствующие классы из пакета `org.apache.hadoop.mapreduce`.

Затем Pig вызывает метод `getInputFormat()` для создания объекта `RecordReader` для каждого сплита, как и в MapReduce. Pig передает каждый объект `RecordReader` методу `prepareToRead()` реализации `CutLoadFunc`, где ссылка на него сохраняется для последующего использования в методе `getNext()` при переборе записей.

Исполнительная среда Pig многократно вызывает `getNext()`, а функция загрузки читает кортежи до тех пор, пока не будет достигнута последняя запись в сплите. В этот момент функция возвращает `null` (признак того, что кортежей для чтения не осталось).

Реализация `getNext()` отвечает за преобразование строк входного файла в объекты `Tuple`. Для этого она использует `TupleFactory` — класс Pig для создания экземпляров `Tuple`. Метод `newTuple()` создает новый кортеж с требуемым числом полей, которое определяется количеством классов `Range`; поля заполняются содержимым подстрок, определяемых объектами `Range`.

Необходимо заранее решить, что делать, когда строка оказывается слишком короткой для запрашиваемого диапазона. Первый вариант — выдать исключение и прервать обработку. Такое решение уместно в том случае, если неполные или поврежденные записи критичны для работы вашего приложения. Во многих ситуациях лучше вернуть кортеж с полями `null` и предоставить сценарию Pig возможность поступить с неполными данными так, как он считает нужным. Именно так мы поступаем в данном случае: прерывая цикл `for`, если конец диапазона выходит за границы строки, мы оставляем текущему полю и всем последующим полям кортежа значение по умолчанию `null`.

Использование схемы

А теперь займемся типом загружаемых полей. Если пользователь задал схему, то поля должны быть преобразованы к соответствующим типам. Однако Pig откладывает преобразование до момента использования, так что загрузчик всегда должен конструировать кортежи типа `bytearray` с использованием типа `DataByteArray`. Впрочем, функция-загрузчик все равно имеет возможность выполнить преобразование — для этого следует переопределить метод `getLoadCaster()` так, чтобы он возвращал пользовательскую реализацию интерфейса `LoadCaster`, предоставляющего набор методов преобразования для этой цели:

```
public interface LoadCaster {  
    public Integer bytesToInteger(byte[] b) throws IOException;  
    public Long bytesToLong(byte[] b) throws IOException;  
    public Float bytesToFloat(byte[] b) throws IOException;  
    public Double bytesToDouble(byte[] b) throws IOException;  
    public String bytesToCharArray(byte[] b) throws IOException;  
    public Map<String, Object> bytesToMap(byte[] b) throws IOException;  
    public Tuple bytesToTuple(byte[] b) throws IOException;  
    public DataBag bytesToBag(byte[] b) throws IOException;  
}
```

`CutLoadFunc` не переопределяет `getLoadCaster()`, потому что реализация по умолчанию возвращает объект `Utf8StorageConverter`, обеспечивающий стандартные преобразования между данными в кодировке UTF-8 и типами данных Pig.

В некоторых случаях сама функция загрузки может определить схему. Например, при загрузке самоописываемых данных (например, в формате XML или JSON) схему для Pig можно было бы создать на основе анализа данных. Также функция загрузки может определить схему другим способом — скажем, по внешнему файлу, или по информации, переданной в конструкторе. Для поддержки таких ситуаций функция загрузки должна реализовать интерфейс `LoadMetadata` (кроме интерфейса `LoadFunc`), чтобы он мог предоставить схему исполнительной среде Pig. Однако следует учесть, что схема, переданная пользователем в секции AS оператора LOAD, имеет более высокий приоритет, чем схема, заданная через интерфейс `LoadMetadata`.

Функция загрузки может дополнительно реализовать интерфейс `LoadPushDown` для определения запрашиваемых столбцов. Это может быть полезной оптимизацией для хранения информации в формате, ориентированном на работу со столбцами, чтобы загрузчик загружал только те столбцы, которые нужны запросу. В `CutLoadFunc` не существует очевидного способа загрузки подмножества столбцов, так как для каждого кортежа читается вся строка, поэтому мы эту оптимизацию не используем.

Операторы обработки данных

Загрузка и сохранение

В этой главе было показано, как загружать данные из внешнего хранилища для обработки в Pig. Сохранение результатов тоже выполняется достаточно тривиально. В следующем примере класс `PigStorage` используется для хранения кортежей в виде текстовых значений, разделенных двоеточиями (:):

```
grunt> STORE A INTO 'out' USING PigStorage(':' );
grunt> cat out
Joe:cherry:2
Ali:apple:3
Joe:banana:2
Eve:apple:7
```

Другие встроенные функции сохранения описаны в табл. 11.7.

Фильтрация данных

После того, как данные будут загружены в отношение, следующим шагом часто становится их фильтрация для исключения из обработки данных, которые вас не интересуют. Ранняя фильтрация в конвейере обработки сводит к минимуму

объем данных, проходящих через систему, что может способствовать повышению эффективности.

FOREACH...GENERATE

Вы уже видели пример удаления строк из отношения простым оператором `FILTER` с простыми выражениями и пользовательской функцией. Оператор `FOREACH...GENERATE` выполняет операцию с каждой строкой отношения; он может использоваться для удаления существующих и генерирования новых полей. В следующем примере мы делаем и то, и другое:

```
grunt> DUMP A;
(Joe,cherry,2)
(Ali,apple,3)
(Joe,banana,2)
(Eve,apple,7)
grunt> B = FOREACH A GENERATE $0, $2+1, 'Constant';
grunt> DUMP B;
(Joe,3,Constant)
(Ali,4,Constant)
(Joe,3,Constant)
(Eve,8,Constant)
```

Мы создаем новое отношение `B` с тремя полями. Первое поле является проекцией первого поля (`$0`) отношения `A`. Второе поле `B` содержит третье поле `A` (`$2`), увеличенное на 1.

Третье поле `B` содержит константу (у всех записей в `B` его значение одинаково) `Constant` типа `chararray`.

У оператора `FOREACH...GENERATE` существует вложенная форма для поддержки более сложных операций. В следующем примере мы вычисляем различные статистические характеристики набора метеорологических данных:

```
-- year_stats.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
records = LOAD 'input/ncdc/all/19{1,2,3,4,5}*'
  USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,16-19,88-92,93-93')
  AS (usaf:chararray, wban:chararray, year:int, temperature:int, quality:int);

grouped_records = GROUP records BY year PARALLEL 30;
year_stats = FOREACH grouped_records {
  uniq_stations = DISTINCT records.usaf;
  good_records = FILTER records BY isGood(quality);
```

продолжение ↗

```
GENERATE FLATTEN(group), COUNT(uniq_stations) AS station_count,
    COUNT(good_records) AS good_record_count, COUNT(records) AS record_count;
}

DUMP year_stats;
```

При помощи пользовательской функции «нарезки», разработанной ранее, мы загружаем различные поля из входного набора данных в отношение `records`, которое затем группируется по годам. Обратите внимание на ключевое слово `PARALLEL`, задающее количество используемых сверток; оно очень важно при выполнении в кластере. Затем каждая группа обрабатывается вложенным оператором `FOREACH...GENERATE`. Первая вложенная инструкция создает отношение для различающихся идентификаторов метеостанций (ключевое слово `DISTINCT`). Вторая вложенная инструкция создает отношение для записей с «хорошими» значениями, используя оператор `FILTER` и пользовательскую функцию. Последняя вложенная инструкция, `GENERATE` (последней вложенной инструкцией `FOREACH...GENERATE` всегда должна быть инструкция `GENERATE`) генерирует сводку с использованием сгруппированных записей и отношений, созданных во вложенном блоке.

Для данных за несколько лет будет получен следующий результат:

```
(1920,8L,8595L,8595L)
(1950,1988L,8635452L,8641353L)
(1930,121L,89245L,89262L)
(1910,7L,7650L,7650L)
(1940,732L,1052333L,1052976L)
```

Поля содержат год, количество уникальных станций, общее количество «хороших» показаний и общее количество показаний. Как видно из результатов, количество метеостанций и показаний растет со временем.

STREAM

Оператор `STREAM` позволяет преобразовать данные в отношение с использованием внешней программы или сценария. Имя выбрано по аналогии с интерфейсом Hadoop Streaming, предоставляющим аналогичную возможность для MapReduce (см. «Hadoop Streaming», с. 71).

Оператор `STREAM` может использовать встроенные команды с аргументами. В следующем примере команда Unix `cut` используется для извлечения второго поля каждого кортежа `A`. Обратите внимание: команда и ее аргументы заключены в обратные апострофы:

```
grunt> C = STREAM A THROUGH `cut -f 2`;
grunt> DUMP C;
```

```
(cherry)
(apple)
(banana)
(apple)
```

Оператор **STREAM** использует **PigStorage** для сериализации/десериализации отношений со стандартными входными/выходными потоками программы. Кортежи из А преобразуются в строки, разделенные табуляциями, которые передаются сценарию. Вывод сценария читается по строкам и разбивается по табуляциям для создания новых кортежей для выходного отношения С. Вы можете предоставить пользовательский сериализатор и десериализатор, реализующие интерфейсы **PigToStream** и **StreamToPig** соответственно (оба интерфейса находятся в пакете **org.apache.pig**), используя оператор **DEFINE**.

Потоковый механизм Pig приносит наибольшую пользу при написании пользовательских сценариев обработки. Следующий сценарий Python отфильтровывает некорректные метеорологические данные:

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

import re
import sys

for line in sys.stdin:
    (year, temp, q) = line.strip().split()
    if (temp != "9999" and re.match("[01459]", q)):
        print "%s\t%s" % (year, temp)
```

Чтобы использовать сценарий, необходимо доставить его в кластер. Эта задача решается секцией **DEFINE**, которая также создает псевдоним для команды **STREAM**. После этого инструкция **STREAM** может использовать ссылку на псевдоним, как показывает следующий сценарий Pig:

```
-- max_temp_filter_stream.pig
DEFINE is_good_quality `is_good_quality.py`
SHIP ('ch11/src/main/python/is_good_quality.py');
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = STREAM records THROUGH is_good_quality
AS (year:chararray, temperature:int);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

Группировка и соединение данных

Соединение наборов данных в MapReduce требует определенных усилий со стороны программиста (см. «Соединения», с. 368), тогда как имеет очень хорошую встроенную поддержку операций соединения, которая существенно упрощает их выполнение. Так как большие наборы данных, подходящие для анализа Pig (и MapReduce вообще) не нормализованы, соединения в Pig используются реже, чем в SQL.

JOIN

Рассмотрим пример внутреннего соединения. Допустим, имеются отношения A и B:

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

Два отношения можно соединить по числовому полю (идентификатору), присутствующему в каждом отношении:

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

Это классический пример внутреннего соединения, в котором каждое соответствие между двумя отношениями представлено строкой в результате. Поля результата состоят из всех полей всех входных отношений.

Используйте обобщенный оператор соединения, когда все соединяемые отношения слишком велики, чтобы поместиться в памяти. Если одно из отношений достаточно компактно, существует особая разновидность соединения — так называемое *фрагментарное репликационное соединение*. Оно реализуется распределением компактных входных данных по всем отображениям с последующим выполнением соединения на стороне отображения по таблице соответствий, находящейся

в памяти, с (фрагментированным) большим отношением. В Pig существует особый синтаксис для выполнения фрагментарных репликационных соединений¹:

```
grunt> C = JOIN A BY $0, B BY $1 USING "replicated";
```

Первое отношение должно быть большим, а за ним должны следовать одно или несколько малых (все они должны помещаться в памяти).

Pig также поддерживает внешние соединения с синтаксисом, сходным с синтаксисом SQL (эта тема рассматривается для Hive в разделе «Внешние соединения», с. 564). Пример:

```
grunt> C = JOIN A BY $0 LEFT OUTER, B BY $1;
grunt> DUMP C;
(1,Scarf,,)
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

COGROUP

JOIN всегда создает «плоскую» структуру: простое множество кортежей. Инструкция COGROUP похожа на JOIN, но создает она вложенные наборы выходных кортежей. Это может быть полезно, если структура будет использоваться в следующих инструкциях:

```
grunt> D = COGROUP A BY $0, B BY $1;
grunt> DUMP D;
(0,{},{{(Ali,0)}})
(1,{{(1,Scarf)}},{})
(2,{{(2,Tie)},{{(Joe,2),(Hank,2)}}}
(3,{{(3,Hat)}},{{(Eve,3)}})
(4,{{(4,Coat)}},{{(Hank,4)}})
```

COGROUP генерирует кортеж для каждого уникального ключа группировки. Первое поле каждого кортежа содержит ключ, а остальные поля содержат неупорядоченные множества кортежей с совпадающим ключом. Первое неупорядоченное множество содержит соответствующие кортежи из отношения А с тем же ключом, а второе — соответствующие кортежи из множества В с тем же ключом.

¹ В секции USING могут использоваться и другие ключевые слова — например, «skewed» (для больших наборов данных с неоднородным пространством ключей) и «merge» (для выполнения слияния для входных данных, уже отсортированных по ключу объединения). За подробной информацией о том, как использовать эти специализированные объединения, обращайтесь к документации Pig.

Если для некоторого ключа в отношении не существует соответствующего ключа, неупорядоченное множество для этого отношения остается пустым (как, например, второе неупорядоченное множество кортежа для идентификатора 1). Это пример внешнего соединения — эта разновидность используется по умолчанию для COGROUP. Внешнее соединение также можно явно обозначить ключевым словом OUTER; следующая инструкция COGROUP эквивалентна предыдущей:

```
D = COGROUP A BY $0 OUTER, B BY $1 OUTER;
```

Для подавления строк с пустыми неупорядоченными множествами можно воспользоваться ключевым словом INNER, которое наделяет инструкцию COGROUP семантикой внутреннего соединения. Ключевое слово INNER применяется на уровне отношений, поэтому следующий фрагмент подавляет строки только в том случае, если отношение A не имеет совпадения (с исключением неизвестного продукта 0):

```
grunt> E = COGROUP A BY $0 INNER, B BY $1;
grunt> DUMP E;
(1,{{(1,Scarf)}},{})
(2,{{(2,Tie)}},{{(Joe,2),(Hank,2)}})
(3,{{(3,Hat)}},{{(Eve,3)}})
(4,{{(4,Coat)}},{{(Hank,4)}})
```

Чтобы узнать, кто купил каждый из предметов из отношения A, структуру можно «сплющить»:

```
grunt> F = FOREACH E GENERATE FLATTEN(A), B.$0;
grunt> DUMP F;
(1,Scarf,{})
(2,Tie,{{(Joe),(Hank)}})
(3,Hat,{{(Eve)}})
(4,Coat,{{(Hank)}})
```

Внутреннее соединение можно имитировать при помощи COGROUP, INNER и FLATTEN (для удаления вложенных структур):

```
grunt> G = COGROUP A BY $0 INNER, B BY $1 INNER;
grunt> H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);
grunt> DUMP H;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

Результат будет таким же, как при выполнении инструкции JOIN A BY \$0, B BY \$1.

Если ключ соединения состоит из нескольких полей, все эти поля можно указать в секциях BY инструкции JOIN или COGROUP. Проследите за тем, чтобы количества полей в каждой секции BY были одинаковыми.

Другой пример соединения в Pig — в сценарии для вычисления максимальной температуры для каждой станции за период времени, определяемый входными данными:

```
-- max_temp_station_name.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt'
    USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')
    AS (usaf:chararray, wban:chararray, name:chararray);

trimmed_stations = FOREACH stations GENERATE usaf, wban,
    com.hadoopbook.pig.Trim(name);

records = LOAD 'input/ncdc/all/191*'
    USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93')
    AS (usaf:chararray, wban:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30;
max_temp = FOREACH grouped_records GENERATE FLATTEN(group),
    MAX(filtered_records.temperature);
max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf, wban)
    PARALLEL 30;
max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2;

STORE max_temp_result INTO 'max_temp_by_station';
```

Пользовательская функция, разработанная нами ранее, используется для загрузки одного отношения, содержащего идентификаторы (USAF и WBAN) и названия станций, и другого отношения, содержащего все метеорологические записи, ключами которых являются идентификаторы станций. Отфильтрованные метеорологические записи группируются по идентификатору станции и агрегируются по максимальной температуре перед соединением с данными станций. В завершение мы задаем поля, которые должны войти в окончательный результат.

Пример выходных данных:

228020	99999	SORTAVALA	322
029110	99999	VAASA AIRPORT	300
040650	99999	GRIMSEY	378

Для повышения эффективности запроса можно было воспользоваться фрагментарным репликационным соединением, так как метаданные станций невелики.

CROSS

Pig Latin включает в себя оператор перекрестного произведения (также называемого декартовым произведением), в котором каждый кортеж отношения соединяется с каждым кортежем другого отношения (и каждым кортежем последующих отношений, если они указаны). Размер выходных данных равен произведению размеров входных данных, то есть потенциально может быть очень большим:

```
grunt> I = CROSS A, B;
grunt> DUMP I;
(2,Tie,Joe,2)
(2,Tie,Hank,4)
(2,Tie,Ali,0)
(2,Tie,Eve,3)
(2,Tie,Hank,2)
(4,Coat,Joe,2)
(4,Coat,Hank,4)
(4,Coat,Ali,0)
(4,Coat,Eve,3)
(4,Coat,Hank,2)
(3,Hat,Joe,2)
(3,Hat,Hank,4)
(3,Hat,Ali,0)
(3,Hat,Eve,3)
(3,Hat,Hank,2)
(1,Scarf,Joe,2)
(1,Scarf,Hank,4)
(1,Scarf,Ali,0)
(1,Scarf,Eve,3)
(1,Scarf,Hank,2)
```

При работе с большими наборами данных следует по возможности избегать операций, генерирующих промежуточные представления квадратичного (и выше) размера. Вычисление перекрестных произведений по всему входному набору данных редко бывает действительно необходимым.

Например, на первый взгляд может показаться, что для определения попарного сходства документов перед вычислением метрик нужно сначала сгенерировать каждую пару документов. Но если учесть, что у большинства пар документов степень сходства равна нулю (то есть документы не связаны), становится ясно, что нужно искать более эффективный алгоритм.

В нашем случае нужно сосредоточиться на сущностях, используемых для вычисления сходства (например, терминах, использованных в документе), и поставить их в центр алгоритма. Также стоит исключить условия, неучаствующие в оценке сходства (незначащие слова) — это способствует дополнительному сокращению

пространства задачи. При использовании этого приема для анализа приблизительно одного миллиона (10^6) документов генерируется около миллиарда (10^9) промежуточных пар¹ — вместо триллиона (10^{12}) пар, которые были бы сгенерированы при примитивном подходе (генерировании перекрестного произведения входных данных) или без удаления незначащих слов.

GROUP

В отличие от инструкции **COGROUP**, группирующей данные в двух и более отношениях, инструкция **GROUP** группирует данные в одном отношении. Для поддержки группировки **GROUP** не ограничивается равенством ключей: в качестве ключа группировки может использоваться выражение или пользовательская функция. Для примера возьмем следующее отношение **A**:

```
grunt> DUMP A;
(Joe, cherry)
(Ali, apple)
(Joe, banana)
(Eve, apple)
```

Выполним группировку по количеству символов во втором поле:

```
grunt> B = GROUP A BY SIZE($1);
grunt> DUMP B;
(5, {(Ali, apple), (Eve, apple)})
(6, {(Joe, cherry), (Joe, banana)})
```

GROUP создает отношение, первым полем которого является поле группировки, которому назначается псевдоним **group**. Второе поле содержит сгруппированные поля с такой же схемой, как у исходного отношения (**A** в нашем случае).

Также существует две специальные операции группировки: **ALL** и **ANY**. **ALL** группирует все кортежи отношения в одну группу, как если бы функция **GROUP** была константой:

```
grunt> C = GROUP A ALL;
grunt> DUMP C;
(all, {(Joe, cherry), (Ali, apple), (Joe, banana), (Eve, apple)})
```

Обратите внимание на отсутствие **BY** в этой форме инструкции **GROUP**. Группировка **ALL** обычно используется для подсчета кортежей в отношениях, как показано в разделе «Проверка и null», с. 491.

¹ . Elsayed, Lin, and Oard. 2008. «Pairwise Document Similarity in Large Collections with MapReduce». Proceedings of the 46th Annual Meeting of the Association of Computational Linguistics (ACL 2008): 2650268.

Ключевое слово ANY выполняет случайную группировку кортежей в отношениях, что может быть полезно для формирования тестовых выборок.

Сортировка данных

Отношения Pig не упорядочены. Возьмем следующее отношение A:

```
grunt> DUMP A;  
(2,3)  
(1,2)  
(1,2)  
(2,4)
```

Порядок обработки строк не гарантирован. В частности, при выборке содержимого A оператором DUMP или STORT строки могут быть записаны в любом порядке. Если вы хотите установить конкретный порядок выходных данных, используйте оператор ORDER для сортировки отношения по одному или нескольким полям. Порядок сортировки по умолчанию сравнивает поля одного типа с использованием естественного упорядочения, а разным типам назначается детерминированный порядок (например, кортеж всегда «меньше, чем» неупорядоченное множество).

Следующий пример сортирует A по возрастанию первого поля, а затем по убыванию второго поля:

```
grunt> B = ORDER A BY $0, $1 DESC;  
grunt> DUMP B;  
(1,2)  
(2,4)  
(2,3)
```

Сохранение порядка отсортированного отношения при последующей обработке не гарантировано. Пример:

```
grunt> C = FOREACH B GENERATE *;
```

Комбинирование и разбиение данных

Иногда требуется объединить несколько отношений в одно. Для этой цели используется оператор UNION, например:

```
grunt> DUMP A;  
(2,3)  
(1,2)  
(1,2)  
(2,4)  
grunt> DUMP B;
```

```
(z,x,8)
(w,y,1)
grunt> C = UNION A, B;
grunt> DUMP C;
(2,3)
(1,2)
(2,4)
(z,x,8)
(w,y,1)
```

С является объединением отношений А и В. Так как отношения А и В не упорядочены, порядок кортежей С не определен. Объединение может быть образовано от двух отношений с разными схемами или разным количеством полей, как в нашем случае. Pig пытается объединить схемы отношений, с которыми работает UNION. В нашем случае они несовместимы, поэтому у отношения С схемы нет:

```
grunt> DESCRIBE A;
A: {f0: int,f1: int}
grunt> DESCRIBE B;
B: {f0: chararray,f1: chararray,f2: int}
grunt> DESCRIBE C;
Schema for C unknown.
```

Если у выходного отношения нет схемы, ваш сценарий должен уметь обрабатывать кортежи с переменными количествами и(или) типами полей.

Оператор `SPLIT` по смыслу противоположен `UNION`: он разбивает отношение на два или более отношения. Пример использования приведен в разделе «Проверка и null», с. 491.

Практическое использование Pig

Каждый, кто занимается разработкой и выполнением программ Pig, должен знать некоторые практические приемы, описанные в этой главе.

Параллелизм

В режиме MapReduce очень важно, чтобы степень параллелизма соответствовала размеру набора данных. По умолчанию Pig выбирает количество сверток по размеру входных данных, выделяя по одной задаче свертки на 1 Гбайт ввода вплоть до максимального количества 999. Параметры можно переопределить, задавая свойства `pig.exec.reducers.bytes.per.reducer` (по умолчанию 1 000 000 000 байт) и `pig.exec.reducers.max` (по умолчанию 999).

Чтобы явно задать количество сверток для задания, используйте секцию **PARALLEL** для операторов, выполняемых в фазе свертки. К их числу принадлежат все операторы группировки и соединения (**GROUP**, **COGROUP**, **JOIN**, **CROSS**), а также **DISTINCT** и **ORDER**. Следующая строка устанавливает для **GROUP** количество сверток, равное 30:

```
grouped_records = GROUP records BY year PARALLEL 30;
```

Также можно задать переменную **default_parallel**, которая будет действовать для всех последующих заданий:

```
grunt>
set default_parallel 30
```

Хорошее значение количества задач свертки немногим меньше количества слотов свертки в кластере. За дополнительной информацией обращайтесь к разделу «Выбор количества задач свертки», с. 305.

Количество задач отображения определяется размером входных данных (одно отображение на блок HDFS) и не зависит от секции **PARALLEL**.

Подстановка параметров

Если некоторый сценарий Pig регулярно запускается в вашей системе, обычно появляется необходимость запускать его с разными параметрами. Например, ежедневно запускаемый сценарий может использовать дату для определения файлов, подлежащих обработке. Pig поддерживает механизм *подстановки параметров*, то есть замены параметров в сценариях значениями, переданными во время выполнения. Параметры обозначаются идентификаторами с префиксом \$; например, в следующем сценарии **\$input** и **\$output** используются для задания входных и выходных путей:

```
-- max_temp_param.pig
records = LOAD '$input' AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR
   quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
STORE max_temp into '$output';
```

Параметры могут задаваться при запуске Pig с ключом **-param** (по одному ключу для каждого параметра):

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
>     -param output=/tmp/out \
>     ch11/src/main/pig/max_temp_param.pig
```

Параметры также можно сохранить в файле и передать их Pig с ключом *-param_file*. Пример достижения того же результата, что и в предыдущей команде, с определением параметров в файле:

```
# Входной файл
input=/user/tom/input/ncdc/micro-tab/sample.txt
# Выходной файл
output=/tmp/out
```

Команда запуска *pig* принимает следующий вид:

```
% pig -param_file ch11/src/main/pig/max_temp_param.param \
>     ch11/src/main/pig/max_temp_param.pig
```

Чтобы задать несколько файлов с параметрами, включите в командную строку несколько ключей *-param_file*. Допускается использование комбинаций *-param* и *-param_file*, а если какой-либо параметр определяется и в файле параметров, и в командной строке, используется последнее значение в командной строке.

Динамические параметры

Для параметров, задаваемых с ключом командной строки *-param*, легко организовать динамическое определение значений с выполнением команды или сценария. Многие командные процессоры Unix поддерживают подстановку команд, заключенных в обратные апострофы; например, этот механизм можно использовать для выбора выходного каталога с учетом текущей даты:

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
>     -param output=/tmp/^date "+%Y-%m-%d"/out \
>     ch11/src/main/pig/max_temp_param.pig
```

Pig также поддерживает обратные апострофы в файлах параметров — заключенная в них команда выполняется, а полученный вывод используется как подставляемое значение. Если выполнение команды или сценария завершается с ненулевым кодом, выводится сообщение об ошибке, и выполнение прерывается.

Возможность включения обратных апострофов в файлы параметров весьма полезна; она означает, что параметры могут одинаково определяться в файле и в командной строке.

Обработка подстановки параметров

Подстановка параметров выполняется в фазе препроцессора перед выполнением сценария. Чтобы увидеть подстановки, выполненные препроцессором, запустите Pig с ключом *-dryrun*. В режиме «холостого запуска» Pig выполняет подстановку параметров (и расширение макросов) и генерирует копию исходного сценария с подставленными значениями, но не выполняет ее. Вы можете просмотреть сконвертированный сценарий и убедиться в том, что подстановки выглядят нормально (например, если они генерируются динамически), прежде чем запускать его в обычном режиме.

На момент написания книги подстановка параметров не поддерживалась Grunt.

12 Hive

В своей книге, посвященной данным¹, Джейф Хаммербахер описывает информационные платформы как «средоточие всей работы по поглощению, обработке и генерированию информации в организациях», а также показывает, как они «содействуют ускорению процесса извлечения полезной информации из эмпирических данных».

Одним из главных компонентов информационной платформы, построенной группой Джейффа в Facebook, была система Hive, предназначенная для организации информационных хранилищ на базе Hadoop. Технология Hive выросла из потребности в управлении и извлечении информации из огромных объемов данных, ежедневно производимых Facebook в стремительно растущей социальной сети. Опробовав несколько разных систем, группа выбрала Hadoop для хранения и обработки информации, так как эта технология была экономичной и удовлетворяла их потребности в масштабировании.

Система Hive создавалась для того, чтобы аналитики, хорошо владеющие SQL (но слабо разбирающиеся в программировании на Java), могли выполнять запросы к гигантским объемам данных, хранимых Facebook в HDFS. Сегодня Hive — успешный проект Apache, используемый во многих организациях как универсальная и масштабируемая платформа обработки данных.

Конечно, язык SQL не идеален для некоторых задач больших данных — например, он плохо подходит для построения сложных алгоритмов машинного самообучения.

¹ «Beautiful Data: The Stories Behind Elegant Data Solutions», Тоби Сегаран (Toby Segaran) и Джейф Хаммербахер (Jeff Hammerbacher) (O'Reilly, 2009)

Тем не менее во многих аналитических областях он прекрасно работает; еще одно огромное преимущество — его известность в отрасли. Кроме того, SQL является общепринятым языком инструментов бизнес-аналитики, так что интеграция Hive с этими продуктами абсолютно уместна.

Эта глава знакомит читателя с основами Hive. Предполагается, что у вас уже имеются практические знания SQL и общей архитектуры баз данных; рассматривая и Hive, мы будем часто сравнивать их с аналогами из традиционных РСУБД.

Установка Hive

При нормальном использовании Hive преобразует запрос SQL в серию заданий MapReduce, выполняемых в кластере Hadoop. Hive упорядочивает данные в таблицы, а это открывает возможности структурирования данных, хранящихся в HDFS. Метаданные — например, схемы таблиц — хранятся в базе данных, называемой *метахранилищем*.

В процессе изучения Hive удобно разместить метахранилище на вашей локальной машине. В этой конфигурации, используемой по умолчанию, создаваемые вами определения таблиц Hive будут локальными для вашей машины, так что вы не сможете использовать их совместно с другими пользователями. О том, как настроить общее удаленное метахранилище (нормальное в средах реальной эксплуатации), рассказано в разделе «Метахранилище», с. 533.

Процесс установки Hive тривиален. На машине должен быть заранее установлен пакет Java 6, в Windows вам также понадобится Cygwin. Кроме того, локально должна быть установлена та же версия Hadoop, которая работает в кластере¹. Конечно, во время освоения Hive вы можете запустить Hadoop локально — в автономном или псевдораспределенном режиме.

Загрузите новую версию по адресу <http://hive.apache.org/releases.html> и распакуйте архив в подходящий каталог на рабочей станции:

```
% tar xzf hive-x.y.z.tar.gz
```

Путь к Hive рекомендуется включить в переменные окружения, чтобы упростить запуск:

```
% export HIVE_INSTALL=/home/tom/hive-x.y.z-dev
% export PATH=$PATH:$HIVE_INSTALL/bin
```

¹ Предполагается, что рабочая станция имеет сетевое подключение к кластеру Hadoop. Протестируйте его перед запуском Hive — установите Hadoop локально и попробуйте выполнить какие-нибудь операции HDFS командой hadoop fs.

Ведите команду *hive*, чтобы запустить оболочку Hive:

```
% hive  
hive>
```

С КАКИМИ ВЕРСИЯМИ HADOOP РАБОТАЕТ HIVE?

Любая версия Hive предназначена для работы с разными версиями Hadoop. Как правило, Hive работает с последней стабильной версией Hadoop, а также поддерживает ряд более старых версий, перечисленных в замечаниях к выпуску. Чтобы сообщить Hive, какую версию Hadoop вы используете, вам не придется делать ничего особенного — разве что проследить за тем, чтобы исполняемый файл hadoop находился в известном каталоге, или задать переменную окружения HADOOP_HOME.

Оболочка Hive

Взаимодействие с Hive в основном происходит через программную оболочку, в которой вводятся команды на HiveQL — языке запросов Hive, диалекте SQL. Этот язык создавался под значительным влиянием MySQL, так что если вы владеете MySQL, то и язык Hive покажется вам знакомым.

Чтобы проверить работоспособность Hive при первом запуске, можно запросить список таблиц (ни одной таблицы быть не должно). Чтобы команда была выполнена Hive, завершите ее символом «;»:

```
hive> SHOW TABLES;  
OK  
Time taken: 10.425 seconds
```

Как и SQL, HiveQL обычно игнорирует регистр символов (кроме сравнений строк), поэтому команда `show tables;` сработает с таким же успехом. Клавиша Tab автоматически завершает ключевые слова и функции Hive.

Первое выполнение команды займет несколько секунд, так как команда создает базу данных метадат на вашей машине. (Файлы базы данных хранятся в каталоге с именем *metastore_db*, местонахождение которого определяется относительно каталога, из которого была запущена команда *hive*.)

Оболочку Hive можно запустить в неинтерактивном режиме. С ключом *-f* команда выполняет команды из заданного файла — в следующем примере это файл *script.q*:

```
% hive -f script.q
```

Для коротких сценариев можно воспользоваться параметром `-e` с указанием встроенной команды; в этом случае завершающий символ «;» не обязателен:

```
% hive -e 'SELECT * FROM dummy'  
Hive history file=/tmp/tom/hive_job_log_tom_201005042112_1906486281.txt  
OK  
X  
Time taken: 4.734 seconds
```



Полезно иметь небольшую таблицу данных для тестирования запросов — например, проверки функций в выражениях `SELECT` с использованием литеральных данных (см. «Операторы и функции», с. 542). Один из способов заполнения таблицы с одной строкой:

```
% echo 'X' > /tmp/dummy.txt  
% hive -e "CREATE TABLE dummy (value STRING); \  
LOAD DATA LOCAL INPATH '/tmp/dummy.txt' \  
OVERWRITE INTO TABLE dummy"
```

Как в интерактивном, так и в неинтерактивном режиме Hive в ходе операции выводит информацию в стандартный поток ошибок — например, время выполнения запроса. Чтобы подавить эти сообщения, включите параметр `-S` при запуске — тогда выводиться будут только результаты запросов:

```
% hive -S -e 'SELECT * FROM dummy'  
X
```

У оболочки Hive есть и другие полезные возможности — например, выполнение команд операционной системы с использованием префикса `!` или обращение к файловым системам Hadoop командой `dfs`.

Пример

Рассмотрим пример использования Hive для выполнения запроса к набору метеорологических данных, рассмотренному в предыдущих главах. Прежде всего необходимо загрузить данные в управляемое хранилище Hive. Сейчас мы воспользуемся для хранения локальной файловой системой; позднее будет показано, как хранить таблицы в HDFS.

Hive, как и традиционные РСУБД, хранит данные в таблицах. Таблица для хранения метеорологических данных создается инструкцией `CREATE TABLE`:

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';
```

Первая строка объявляет таблицу `records` с тремя столбцами: `year`, `temperature` и `quality`. Тип каждого столбца должен быть задан в объявлении. В нашем примере столбец `year` относится к строковому типу, а два других столбца содержат целые числа.

Пока что код SQL выглядит знаком. Впрочем, секция `ROW FORMAT` появилась только в HiveQL. В данном случае объявление сообщает, что каждая строка файла данных содержит текст, разделенный табуляциями. Hive ожидает, что в каждой строке хранятся три поля, соответствующие столбцам таблицы; поля разделяются табуляциями, а строки данных — символами новой строки.

Далее таблица Hive заполняется данными. Мы воспользуемся небольшой выборкой для исследовательских целей:

```
LOAD DATA LOCAL INPATH 'input/ncdc/micro-tab/sample.txt'
OVERWRITE INTO TABLE records;
```

Команда приказывает Hive поместить указанный локальный файл в каталог хранилища. Выполняется простая операция файловой системы. Например, мы не пытаемся разобрать файл и сохранить его во внутреннем формате баз данных, так как Hive не требует применения какого-то конкретного формата. Файлы сохраняются в исходном виде; они не модифицируются Hive.

В данном примере мы сохраняем таблицы Hive в локальной файловой системе (свойство `fs.default.name` имеет значение по умолчанию `file:///`). Таблицы сохраняются в каталогах внутри каталога хранилища Hive, который определяется свойством `hive.metastore.warehouse.dir` (по умолчанию используется значение `/user/hive/warehouse`).

Итак, файлы таблицы `records` хранятся в каталоге `/user/hive/warehouse/records` локальной файловой системы:

```
% ls /user/hive/warehouse/records/
sample.txt
```

В нашем случае используется только один файл `sample.txt`, но в общем случае файлов может быть больше. При запросе к таблице Hive читает данные из всех файлов.

Ключевое слово `OVERWRITE` в инструкции `LOAD DATA` приказывает Hive удалить все существующие файлы в каталоге таблицы. Если это ключевое слово не указано, новые файлы просто добавляются в каталог таблицы (при совпадении имен происходит замена старых файлов).

Теперь, когда данные хранятся в Hive, к ним можно обратиться с запросом:

```
hive> SELECT year, MAX(temperature)
    > FROM records
    > WHERE temperature != 9999
    > AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR
           quality = 9)
    > GROUP BY year;
1949      111
1950      22
```

Запрос SQL внешне ничем не примечателен: инструкция SELECT с условием GROUP BY для группировки записей по годам, использующая агрегатную функцию MAX() для поиска максимальной температуры по каждой группе лет. Примечательно здесь то, что Hive преобразует запрос в задание MapReduce, которое выполняет за нас, после чего выводит результаты на консоль. При этом приходится учитывать некоторые нюансы — например, конструкции SQL, поддерживаемые Hive, формат запрашиваемых данных (мы рассмотрим некоторые из них в этой главе), но мощь Hive проявляется именно в возможности выполнения запросов SQL к необработанным данным.

Администрирование Hive

В этом разделе рассматриваются некоторые практические аспекты администрирования Hive, включая настройку Hive для работы в кластере Hadoop и использования общего метахранилища. При этом мы поближе познакомимся с архитектурой Hive.

Настройка конфигурации Hive

Настройка конфигурации Hive, как и настройка Hadoop, осуществляется с использованием конфигурационного файла в формате XML. Файл с именем *hive-site.xml* находится в каталоге Hive *conf*. В этом файле задаются свойства, которые должны задаваться при каждом запуске Hive. В этом же каталоге хранится файл *hive-default.xml*, в котором документированы свойства, предоставляемые Hive, и их значения по умолчанию.

Каталог конфигурации, в котором Hive ищет файл *hive-site.xml*, можно переопределить, передавая команде *hive* параметр *-config*:

```
% hive --config /Users/tom/dev/hive-conf
```

Обратите внимание: параметр определяет каталог, а не сам файл *hive-site.xml*. Он может быть полезен при наличии множественных файлов конфигурации (скажем, для разных кластеров), которые время от времени сменяют друг друга. Также каталог конфигурации можно задать в переменной окружения `HIVE_CONF_DIR`.

Файл *hive-site.xml* естественно использовать для размещения подробной информации о подключении к кластеру: файловая система и трекер заданий задаются при помощи стандартных свойств `fs.default.name` и `mapred.job.tracker`. Если они не заданы, по умолчанию используется локальная файловая система и локальный (внутрипроцессный) исполнитель заданий (как и в Hadoop) — что очень удобно для тестирования Hive на небольших наборах данных. Настройки конфигурации метахранилища (см. «Метахранилище», с. 533) также обычно размещаются в *hive-site.xml*.

Hive также позволяет задавать свойства на уровне отдельных сеансов, для чего команде *hive* передается ключ `-hiveconf`. Например, следующая команда задает кластер (в данном случае псевдораспределенный) для сеанса:

```
% hive -hiveconf fs.default.name=localhost  
      -hiveconf mapred.job.tracker=localhost:8021
```



Если вы планируете совместное использование кластера Hadoop несколькими пользователями Hive, сделайте каталоги, используемые Hive, доступными для записи для всех пользователей. Следующие команды создают каталоги и задают их разрешения соответствующим образом:

```
% hadoop fs -mkdir /tmp  
% hadoop fs -chmod a+w /tmp  
% hadoop fs -mkdir /user/hive/warehouse  
% hadoop fs -chmod a+w /user/hive/warehouse
```

Если все пользователи входят в одну группу, достаточно задать разрешения `g+w` для каталога хранения информации.

Настройки также можно изменить во время сеанса командой `SET`. Например, это может быть полезно для изменения настроек задания Hive или MapReduce для конкретного запроса. Так, следующая команда обеспечивает гнездовую группировку данных в соответствии с определением таблицы (см. «Гнезда», с. 547):

```
hive> SET hive.enforce.bucketing=true;
```

Чтобы узнать текущее значение любого свойства, введите команду `SET` с именем свойства:

```
hive> SET hive.enforce.bucketing;  
hive.enforce.bucketing=true
```

Без параметров команда `SET` выводит список всех свойств (и их значений), установленных в Hive. В список не включаются свойства Hadoop по умолчанию, если только они не были явно переопределены одним из способов, описанных в этом разделе. Команда `SET -v` выводит список всех свойств в системе, включая свойства Hadoop со значениями по умолчанию.

Разные способы задания свойств имеют разные приоритеты. В следующем списке они перечислены по убыванию приоритетов:

1. Команда Hive `SET`.
2. Параметр командной строки `-hiveconf`.
3. `hive-site.xml`
4. `hive-default.xml`
5. `hadoop-site.xml` (или эквивалентные файлы `core-site.xml`, `hdfs-site.xml` и `mapred-site.xml`).
6. `hadoop-default.xml` (или эквивалентные файлы `core-default.xml`, `hdfs-default.xml` и `mapred-default.xml`).

Журналы

Журнал ошибок Hive хранится в локальной файловой системе в файле `/tmp/$USER/hive.log`. Он может быть очень полезен при диагностике проблем конфигурации и других типов ошибок. Журналы задач MapReduce также являются полезным источником информации для диагностики; о том, где их искать, рассказано в разделе «Журналы Hadoop», с. 240.

Конфигурация журнала хранится в файле `conf/hive-log4j.properties`; вы можете отредактировать этот файл, чтобы изменить уровни регистрации и задать другие параметры, относящиеся к журналам. Однако часто бывает более удобно задать конфигурацию журнала для конкретного сеанса. Например, со следующей командой отладочные сообщения будут отправляться на консоль:

```
% hive -hiveconf hive.root.logger=DEBUG,console
```

Сервисные функции Hive

Оболочка Hive — всего лишь одна из нескольких сервисных функций, которые могут запускаться командой `hive`. Нужная функция выбирается при помощи параметра `--service`. Команда `hive --service` выводит список всех доступных сервисных функций; самые полезные из них перечислены в следующем списке.

cli

Интерфейс командной строки Hive (оболочка); используется по умолчанию.

hiveserver

Hive запускается в режиме сервера, предоставляющего сервис Thrift для доступа из клиентов, написанных на разных языках. Приложениям, использующим механизмы подключения Thrift, JDBC и ODBC, для взаимодействия с Hive необходим сервер Hive. Порт, на котором ведет прослушивание сервер, задается переменной окружения `HIVE_PORT` (по умолчанию 10 000).

hwi

Веб-интерфейс Hive. См. «Веб-интерфейс Hive (HWI)», с. 531.

jar

Hive-эквивалент команды `hadoop jar`; удобный способ запуска приложений Java, у которых в путь к классам входят как классы Hadoop, так и классы Hive.

metastore

По умолчанию метахранилище работает в одном процессе с Hive. Эта сервисная функция позволяет запустить метахранилище в виде автономного (удаленного) процесса. Порт, на котором ведет прослушивание сервер, задается переменной окружения `METASTORE_PORT`.

ВЕБ-ИНТЕРФЕЙС HIVE (HWI)

Вместо оболочки вы также можете работать с Hive через простой веб-интерфейс. Запустите его следующими командами:

```
% export ANT_LIB=/path/to/ant/lib  
% hive --service hwi
```

(Переменную окружения `ANT_LIB` необходимо задать только в том случае, если библиотека Ant в вашей системе не находится в каталоге `/opt/ant/lib`.) Откройте в браузере страницу по адресу `http://localhost:9999/hwi`. Здесь вы можете просматривать схемы баз данных Hive и создавать сеансы для ввода команд и запросов.

Веб-интерфейс также можно запустить в виде общедоступного сервиса, чтобы предоставить пользователям организации доступ к Hive без установки клиентских программ. Подробности приведены в вики Hive по адресу <https://cwiki.apache.org/confluence/display/Hive/HiveWebInterface>.

Клиенты Hive

Если Hive работает в режиме сервера (`hive --service hiveserver`), существует ряд различных механизмов для подключения к нему из приложений. Отношения между клиентами и сервисными функциями Hive представлены на рис. 12.1.

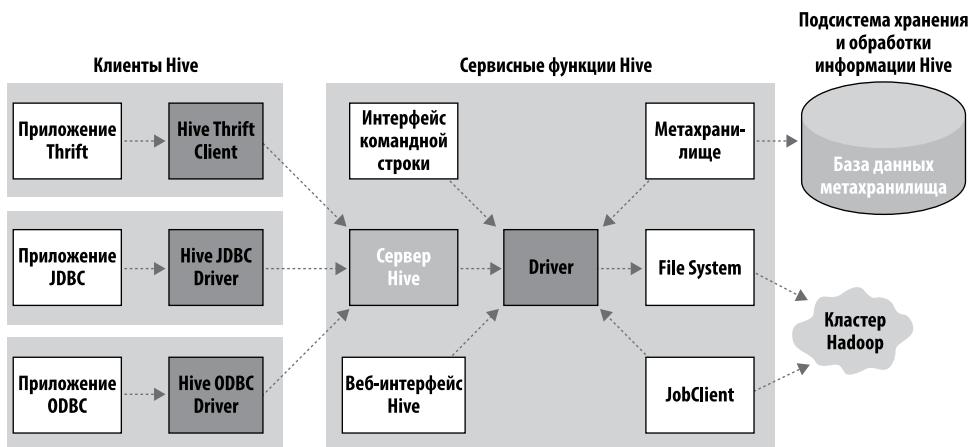


Рис. 12.1. Архитектура Hive

Клиент Thrift

Клиент Thrift для Hive упрощает выполнение команд Hive из многих языков программирования. Привязки Thrift для Hive доступны для C++, Java, PHP, Python и Ruby. Они находятся в подкаталоге `src/service/src` дистрибутива Hive.

Драйвер JDBC

Hive предоставляет драйвер JDBC типа 4 (реализованный полностью на Java), определенный в классе `org.apache.hadoop.hive.jdbc.HiveDriver`. С JDBC URI в форме `jdbc:hive://хост:порт/имя_базы_данных` приложение Java подключится к серверу Hive, работающему в отдельном процессе, с заданным хостом и портом.

Возможно также подключение к Hive через JDBC во встроенным режиме, с использованием URI вида `jdbc:hive://`. В этом режиме Hive выполняется в одной JVM с приложением, которое обратилось с вызовом; соответственно в запуске в виде автономного сервера нет необходимости, поскольку ни сервис Thrift, ни клиент Thrift для Hive не используются.

Драйвер ODBC

Драйвер ODBC для Hive позволяет приложениям, поддерживающим протокол ODBC, подключаться к Hive. (Как и драйвер JDBC, драйвер ODBC

использует Thrift для взаимодействия с сервером Hive.) Драйвер ODBC все еще находится в процессе разработки; за информацией о его построении и запуске обращайтесь к вики Hive.

Более подробная информация об использовании этих клиентов в вики Hive доступна по адресу <https://cwiki.apache.org/confluence/display/Hive/HiveClient>.

Метахранилище

Метахранилище является центральным репозиторием метаданных Hive. Метахранилище разделено на две части: службу и собственно хранилище данных. По умолчанию служба метахранилища работает в одной JVM с Hive и содержит встроенный экземпляр базы данных Derby, использующий локальный диск. Это называется конфигурацией встроенного метахранилища (рис. 12.2).

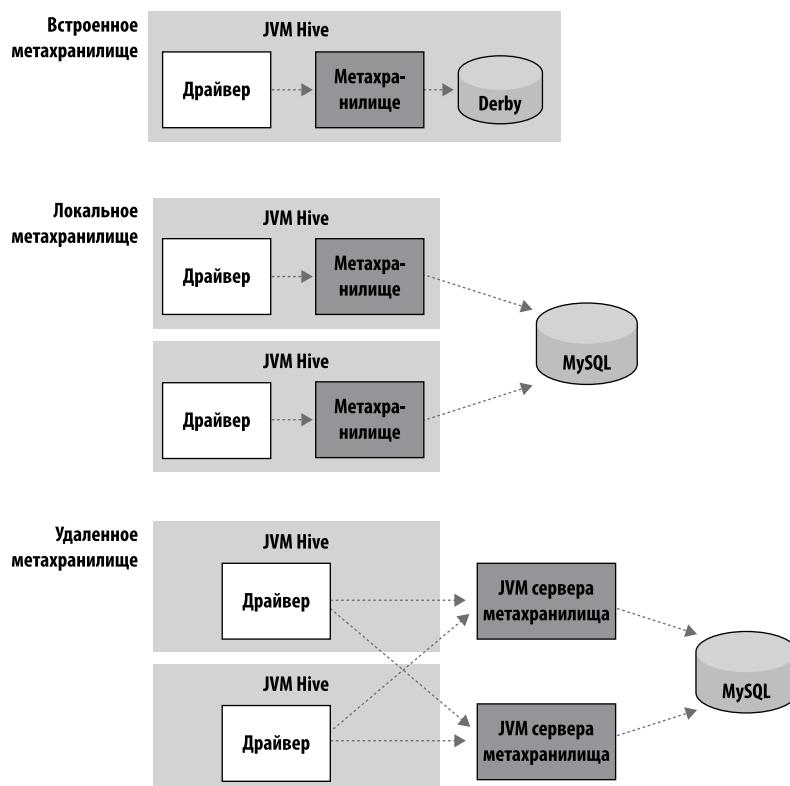


Рис. 12.2. Конфигурации метахранилища

Встроенное метахранилище упрощает работу с Hive на ранних стадиях; однако при этом только одна встроенная база данных Derby в любой момент времени может обращаться к файлам базы данных на диске. Это означает, что в любой момент времени может быть открыт только один сеанс Hive, работающий с метахранилищем. При запуске второго сеанса попытка открытия подключения к метахранилищу приводит к ошибке:

```
Failed to start database 'metastore_db'
```

Для поддержки множественных сеансов (а следовательно, множественных пользователей) следует использовать автономную базу данных. Эта конфигурация называется локальным метахранилищем, поскольку сервис метахранилища также работает в одном процессе с сервисом Hive, но подключается к базе данных, работающей в отдельном процессе — либо на той же, либо на другой машине. Возможно использование любой JDBC-совместимой базы данных, для чего следует задать свойства конфигурации `javax.jdo.option.*`, перечисленные в табл. 12.1¹.

Для организации автономного метахранилища часто используется MySQL. В этом случае свойству `javax.jdo.option.ConnectionURL` задается значение `jdbc:mysql://host/dbname?createDatabaseIfNotExist=true`, а свойству `javax.jdo.option.ConnectionDriverName` — значение `com.mysql.jdbc.Driver`. (Конечно, имя пользователя и пароль тоже должны быть заданы.)

Файл JAR драйвера JDBC для MySQL (Connector/J) должен находиться в пути к классам Hive, для чего достаточно просто поместить его в каталог Hive *lib*.

Также существует конфигурация метахранилища, называемая *удаленным метахранилищем*, при которой один или несколько серверов метахранилища работают в отдельных процессах. Такая организация упрощает управление и повышает безопасность, так как уровень базы данных может быть полностью изолирован, а у клиентов отпадает необходимость в дополнительных удостоверениях для работы с базой данных.

Чтобы настроить Hive на использование удаленного метахранилища, задайте `hive.metastore.local` значение `false`, а `hive.metastore.uris` — URI сервера метахранилища (разделенные запятыми, если их несколько). URI сервера метахранилища имеют форму `thrift://хост:порт`, где *порт* соответствует порту, заданному переменной `METASTORE_PORT` при запуске сервера метахранилища (см. «Сервисные функции Hive», с. 530).

¹ Свойства имеют префикс `javax.jdo`, потому что для сохранения объектов Java реализация метахранилища использует JDO (Java Data Objects) API — а точнее, реализацию JDO DataNucleus.

Таблица 12.1. Важные свойства конфигурации метахранилища

Имя свойства	Тип	Значение по умолчанию	Описание
hive.metastore.warehouse.dir	URI	/user/hive/warehouse	Каталог, заданный относительно fs.default.name, в котором хранятся управляемые таблицы
hive.metastore.local	boolean	true	Признак использования встроенного сервера метахранилища (true) или подключения к удаленному экземпляру (false). Если свойство равно false, то должно быть задано свойство hive.metastore.uris
hive.metastore.uris	URI, разделенные запятыми	–	URI удаленных серверов метахранилища. Подключение клиентов происходит по принципу циклического перехода
javax.jdo.option.ConnectionURL	URI	jdbc:derby:;databaseName=metastore_db;create=true	JDBC URL базы данных метахранилища
javax.jdo.option.ConnectionDriverName	String	org.apache.derby.jdbc.EmbeddedDriver	Имя класса драйвера JDBC
javax.jdo.option.ConnectionUserName	String	APP	Имя пользователя JDBC
javax.jdo.option.ConnectionPassword	String	mine	Пароль JDBC

Сравнение с традиционными базами данных

Хотя Hive во многих отношениях напоминает традиционные базы данных (например, поддержкой интерфейса SQL), из тесной связи этой технологии с HDFS и MapReduce вытекает целый ряд архитектурных различий, которые напрямую влияют на функциональность Hive — которая, в свою очередь, влияет на возможное применение Hive.

Проверка схемы при чтении и записи

В традиционных базах данных схема таблицы проверяется во время загрузки данных. Если загружаемые данные не соответствуют схеме, они отвергаются. Такой метод иногда называется *проверкой схемы при записи*, потому что данные проверяются в момент записи в базу данных.

С другой стороны, Hive проверяет данные не при загрузке, а при выдаче запроса. Это называется *проверкой схемы при чтении*.

У каждого из двух методов есть свои достоинства и недостатки. Проверка схемы при чтении заметно ускоряет начальную загрузку, поскольку данные не нужно читать, разбирать и сериализовать на диск во внутреннем формате базы данных. Операция загрузки сводится к копированию или перемещению файла. Кроме того, улучшается гибкость обработки данных: представьте, что вы можете использовать две схемы для одного набора данных в зависимости от выполняемого анализа. (В Hive это возможно при использовании внешних таблиц; см. «Управляемые и внешние таблицы», с. 543.)

Проверка схемы при записи ускоряет выполнение запросов, потому что база данных может проиндексировать столбцы и выполнить сжатие данных. Вместе с тем процедура загрузки данных в базу выполняется медленнее. Кроме того, схема часто неизвестна на стадии загрузки; в таком случае индексирование невозможно, потому что запросы еще не были сформулированы. В таких ситуациях достоинства Hive проявляются особенно ярко.

Обновления, транзакции и индексы

Обновления, транзакции и индексы — фундаментальные аспекты традиционных баз данных. До недавнего времени эти возможности не входили в набор функций Hive. Дело в том, что технология Hive строилась для работы с данными HDFS с использованием MapReduce, где полный перебор таблиц является нормой, а обновление таблицы реализуется преобразованием данных в новую таблицу. Для приложений, работающих с большими частями наборов данных, такое решение подходит хорошо.

В Hive нет операций обновления (или удаления), но поддерживается синтаксис `INSERT INTO`, так что добавление новых записей в существующую таблицу все же возможно.

С выходом версии 0.7.0 в Hive появилась поддержка индексов, ускоряющих выполнение запросов в некоторых ситуациях. Например, запрос вида `SELECT * from t WHERE x = a` может использовать индекс по столбцу `x`, и просматривать придется

лишь небольшую часть файлов таблицы. В настоящее время реализованы два типа индексов: *компактные* и *поразрядные*. (Архитектура предусматривает возможность замены реализации индекса, так что в будущем ожидается появление разных реализаций для разных сценариев использования.)

Компактные индексы хранят для каждого значения номера блоков HDFS (вместо смещений в файлах); соответственно, они не занимают много места на диске, но эффективно работают при группировке значений в близлежащих строках. Поразрядные индексы используют сжатые битовые карты для эффективного хранения строк таблицы, в которых присутствует конкретное значение; они более эффективны для столбцов с малой кардинальностью (например, «пол» или «страна»).

В версии 0.7.0 в Hive также появилась блокировка уровня таблиц и разделов. Например, блокировка предотвращает удаление таблицы одним процессом, в то время как другой процесс читает из нее данные. В ZooKeeper управление блокировками осуществляется прозрачно, так что пользователю не приходится получать или снимать блокировку (хотя для получения информации об установленных блокировках можно воспользоваться инструкцией `SHOW LOCKS`). По умолчанию блокировки отключены.

Изменения также приходят с другого направления: интеграции с HBase. HBase (глава 13) по своим характеристикам хранения данных отличается от HDFS: в частности, HBase поддерживает возможность обновления строк и индексации столбцов, так что в будущем можно рассчитывать на появление этих возможностей в Hive. Уже сейчас можно обращаться к таблицам HBase из Hive; за дополнительной информацией обращайтесь по адресу <https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration>.

HiveQL

HiveQL — диалект SQL, используемый в Hive, — не поддерживает спецификацию SQL-92 полностью, так как задача совместимости с SQL-32 никогда явно не ставилась в проекте. Вместо этого функции добавлялись разработчиками для удовлетворения потребностей пользователей, а поддержка SQL в Hive расширялась со временем. Кроме того, в Hive имеются расширения, отсутствующие в SQL-92 и появившиеся под влиянием синтаксиса других систем управления базами данных, прежде всего MySQL. Собственно, в первом приближении HiveQL больше всего напоминает диалект SQL из MySQL.

Некоторые расширения SQL-92 в Hive появились под влиянием MapReduce (как, например, многотабличная вставка — см. «Многотабличная вставка», с. 558) и секции `TRANSFORM`, `MAP` и `REDUCE` (см. «Сценарии MapReduce», с. 561).

Эта глава не содержит полного справочника по HiveQL; за справочной информацией обращайтесь к документации Hive по адресу <https://cwiki.apache.org/confluence/display/Hive/LanguageManual>. Вместо этого мы сосредоточимся на часто используемых функциях, обратив особое внимание на отклонения от SQL-92 или популярных баз данных (таких, как MySQL). В табл. 12.2 приведено общее сравнение SQL и HiveQL.

Таблица 12.2. Общее сравнение SQL и HiveQL

Возможность	SQL	HiveQL	См.
Обновления	UPDATE, INSERT, DELETE	INSERT	«Вставка», с. 557; «Обновления, транзакции и ин- дексы», с. 536
Транзакции	Поддерживается	Поддерживается (на уровне таблиц и раз- делов)	
Индексы	Поддерживается	Поддерживается	
Задержка	Доли секунд	Минуты	
Типы данных	Целые, с плавающей точкой, с фиксирован- ной точкой, текстовые и двоичные строки, время/дата	Целые, с плавающей точкой, логический, текстовые и двоичные строки, временная мет- ка, массив, отображе- ние, структура	«Типы данных», с. 539
Функции	Сотни встроенных функций	Десятки встроенных функций	«Операторы и функции», с. 542
Многотаблич- ная вставка	Не поддерживается	Поддерживается	«Многотабличная вставка», с. 558
CREATE TABLE... AS SELECT	Не является допусти- мой синтаксической конструкцией SQL-92, но используется в не- которых базах данных	Поддерживается	«CREATE TABLE... AS SELECT», с. 558
SELECT	SQL-92	Одна таблица или пред- ставление в секции FROM; SORT BY для ча- стичного упорядочения. LIMIT для ограничения количества возвращае- мых записей	«Запросы к дан- ным», с. 560

Возможность	SQL	HiveQL	См.
Соединения	SQL-92 или варианты (соединение таблиц в секции <code>FROM</code> , условие соединения в секции <code>WHERE</code>)	Внутренние соединения, внешние соединения; полусоединения, отображающие соединения (синтаксис SQL-92)	«Соединения», с. 368
Подзапросы	В любой секции (коррелированные или некоррелированные)	Только в секции <code>FROM</code> (коррелированные подзапросы не поддерживаются)	«Подзапросы», с. 566
Представления	Обновляемые (материализованные или нематериализованные)	Только для чтения (материализованные представления не поддерживаются)	«Представления», с. 567
Точки расширения	Пользовательские функции. Хранимые процедуры	Пользовательские функции. Сценарии MapReduce	«Пользовательские функции», с. 498 «Сценарии MapReduce», с. 561

Типы данных

Hive поддерживает примитивные и составные типы данных. К примитивным относятся числовые, логические и строковые типы, а также типы временных меток; к составным — массивы, отображения и структуры. Типы данных Hive перечислены в табл. 12.3. Обратите внимание: литералы приведены в том виде, в котором они используются в HiveQL, а не в сериализованной форме, используемой в формате хранения таблицы (см. «Форматы хранения данных», с. 550).

Таблица 12.3. Типы данных Hive

Категория	Тип	Описание	Примеры литералов
Примитивные	TINYINT	8-разрядное (1 байт) целое число со знаком, от -128 до 127	1
	SMALLINT	16-разрядное (2 байта) целое число со знаком, от -32768 до 32767	1

продолжение ↗

Таблица 12.3 (продолжение)

Категория	Тип	Описание	Примеры литералов
	INT	32-разрядное (4 байта) целое число со знаком, от -2147483648 до 2147483647	1
	BIGINT	64-разрядное (8 байт) целое число со знаком, от -9223372036854775808 до 9223372036854775807	1
	FLOAT	32-разрядное (4 байта) число с плавающей точкой одинарной точности	1.0
	DOUBLE	64-разрядное (8 байт) число с плавающей точкой двойной точности	1.0
	BOOLEAN	Истина/ложь	TRUE
	STRING	Цепочка символов	'a', "a"
	BINARY	Массив байтов	Не поддерживается
	TIME-STAMP	Временная метка с наносекундной точностью	1325502245000, '2012-01-02 03:04:05.123456789'
Составные	ARRAY	Упорядоченный набор полей. Все поля должны относиться к одному типу.	array(1, 2) ^a
	MAP	Неупорядоченный набор пар «ключ-значение». Ключи должны относиться к примитивному типу, тип значения может быть произвольным. В каждом конкретном отображении ключи (а также значения) должны относиться к одному типу.	map('a', 1, 'b', 2)
	STRUCT	Набор именованных полей. Поля могут относиться к разным типам	struct('a', 1, 1.0) ^b

^a Литеральные формы массивов, отображений и структур реализованы как функции. Иначе говоря, `array()`, `map()` и `struct()` являются встроенными функциями Hive.

^b Столбцы обозначаются `col1`, `col2`, `col3` и т. д.

Примитивные типы

Примитивные типы Hive примерно соответствуют примитивным типам Java, хотя некоторые имена были выбраны под влиянием имен типов MySQL (которые, в свою очередь, перекрываются с SQL-92). Всего существуют четыре целочисленных типа со знаком: **TINYINT**, **SMALLINT**, **INT** и **BIGINT**, эквивалентные примитивным типам Java **byte**, **short**, **int** и **long** соответственно; их длина составляет 1, 2, 4 и 8 байт соответственно.

Типы Hive с плавающей точкой **FLOAT** и **DOUBLE** соответствуют типам Java **float** и **double** – 32-разрядному и 64-разрядному. В отличие от некоторых баз данных, Hive не позволяет управлять количеством значащих цифр или разрядов дробной части.

Hive поддерживает тип **BOOLEAN** для хранения значений **true** и **false**.

Единственный текстовый тип данных Hive **STRING** предназначен для хранения строк переменной длины. Тип Hive **STRING** похож на тип **VARCHAR** других баз данных, хотя для **STRING** не существует объявления максимального количества символов. (Теоретически максимальный размер **STRING** составляет 2 Гбайт, хотя на практике хранить текст такой длины было бы неэффективно. В Sqoop поддерживаются большие объекты; см. «Импортирование больших объектов», с. 661).

Тип данных **BINARY** предназначен для хранения двоичных данных переменной длины.

Тип данных **TIMESTAMP** хранит временные метки с наносекундной точностью. В Hive включены пользовательские функции преобразования между временными метками Hive, временными метками Unix (секунды от начала эпохи) и строками, упрощающие выполнение основных операций с датами.

Тип **TIMESTAMP** не инкапсулирует данные часового пояса, но функции **to_utc_timestamp** и **from_utc_timestamp** позволяют выполнять необходимые преобразования.

Составные типы

В Hive определены три составных типа: **ARRAY**, **MAP** и **STRUCT**. Типы **ARRAY** и **MAP** аналогичны одноименным типам Java, а тип **STRUCT** инкапсулирует набор именованных полей. Составные типы поддерживают произвольный уровень вложенности. В объявлениях составных типов должен быть указан тип полей набора; для этой цели используется синтаксис с угловыми скобками, как в следующем определении таблицы с тремя столбцами (по одному для каждого составного типа):

```
CREATE TABLE complex (
    col1 ARRAY<INT>,
    col2 MAP<STRING, INT>,
    col3 STRUCT<a:STRING, b:INT, c:DOUBLE>
);
```

Предположим, таблица была заполнена одной строкой данных для `ARRAY`, `MAP` и `STRUCT`, как показано в столбце «Примеры литералов» табл. 12.3 (необходимый формат файлов представлен в разделе «Форматы хранения данных», с. 550). Следующий запрос демонстрирует операторы обращения к полям для каждого типа:

```
hive> SELECT col1[0], col2['b'], col3.c FROM complex;
1    2    1.0
```

Операторы и функции

Hive поддерживает стандартный набор операторов SQL: операторы отношений (например, `x = 'a'` для проверки равенства, `x IS NULL` для проверки неопределенности, `x LIKE 'a%` для поиска по шаблону), математические операторы (например, `x + 1` для сложения) и логические операторы (например, `x OR y` для операции логического ИЛИ). Обозначения операторов соответствуют набору MySQL; он отличается от набора SQL-92, в котором `||` используется для операции логического ИЛИ, а не для конкатенации строк. Для выполнения конкатенации в MySQL и Hive используется функция `concat`.

Hive содержит большое количество встроенных функций (слишком большое, чтобы перечислять их), разделенных на категории: математические и статистические функции, строковые функции, функции даты (для работы со строковыми представлениями дат), условные функции, агрегатные функции и функции для работы с XML и JSON.

Чтобы увидеть полный список функций из оболочки Hive, введите команду `SHOW FUNCTIONS`¹. Для получения краткого описания конкретной функции используется команда `DESCRIBE`:

```
hive> DESCRIBE FUNCTION length;
length(str | binary) - Returns the length of str or number of bytes in binary data
```

Если встроенной функции для выполнения нужной операции не существует, напишите собственную реализацию; см. «Пользовательские функции», с. 498.

Преобразования

Примитивные типы образуют иерархию, которая определяет неявные преобразования типов, выполняемых Hive в выражениях с функциями и операторами.

¹ Или обратитесь к справочнику функций Hive по адресу <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF>.

Например, `TINYINT` преобразуется в `INT`, если выражение должно получать `INT`; однако обратное преобразование не выполняется, и без использования оператора `CAST` Hive вернет ошибку.

Неявные преобразования типов выполняются по следующим правилам: любой целый числовой тип может быть неявно преобразован в тип большей разрядности. Все целые числовые типы, `FLOAT` и (как ни странно) `STRING` неявно преобразуются в `DOUBLE`. `TINYINT`, `SMALLINT` и `INT` могут быть неявно преобразованы во `FLOAT`. Тип `BOOLEAN` не преобразуется ни в какой другой тип, и никакие неявные преобразования с ним в выражениях невозможны. `TIMESTAMP` неявно преобразуется в `STRING`.

Явные преобразования типов выполняются оператором `CAST`. Например, `CAST('1' AS INT)` преобразует строку '1' в целое значение 1. Если попытка преобразования завершается неудачей (как, например, в случае `CAST('X' AS INT)`), выражение возвращает `NULL`.

Таблицы

Таблицы Hive состоят из хранимых данных и метаданных, описывающих структуру данных в таблице. Данные обычно хранятся в HDFS, хотя они могут храниться в любой файловой системе Hadoop, включая локальную файловую систему или S3.

Hive хранит метаданные в реляционной базе данных, а не, скажем, в HDFS (см. «Метахранилище», с. 533).

В этом разделе рассматривается процесс создания таблиц, различные физические форматы хранения, поддерживаемые Hive, и способы импортирования данных.

Управляемые и внешние таблицы

При создании таблицы Hive по умолчанию выбирает *управляемую таблицу*; это означает, что Hive перемещает данные в свой каталог хранения информации. Также можно приказать Hive создать *внешнюю таблицу*; в этом случае Hive будет работать с данными в существующем каталоге за пределами каталога хранения информации.

Различия между двумя типами таблиц проявляются в семантике операций `LOAD` и `DROP`. Начнем с управляемой таблицы.

Данные, загружаемые в управляемую таблицу, перемещаются в каталог хранения информации Hive. Например, команды

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

перемещают файл `hdfs://user/tom/data.txt` в каталог хранения информации Hive для таблицы `managed_table`, то есть `hdfs://user/hive/warehouse/managed_table1`.



Операция загрузки выполняется очень быстро, потому что она реализуется простым перемещением или переименованием в файловой системе. Однако следует помнить, что Hive не проверяет файлы в каталоге таблицы на соответствие схеме, объявленной для таблицы, даже для управляемых таблиц. Все несоответствия проявятся при обработке запросов — часто для отсутствующих полей запрос будет возвращать NULL. Чтобы убедиться в правильности обработки данных, выполните простую инструкцию `SELECT` для выборки нескольких строк данных непосредственно из таблицы.

При последующем удалении таблицы командой

```
DROP TABLE managed_table;
```

уничтожаются как данные, так и метаданные. Стоит повторить, что исходная операция `LOAD` выполняет перемещение, поэтому после удаления `DROP` данные нигде не существуют. Так Hive поступает с управляемыми данными.

С внешними таблицами операции работают несколько иначе. Вы сами управляете созданием и уничтожением данных. Местонахождение внешних данных задается в момент создания таблицы:

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
  LOCATION '/user/tom/external_table';
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

С ключевым словом `EXTERNAL` Hive знает, что данными управлять не следует, поэтому данные не перемещаются в каталог хранения информации. Hive даже не проверяет, существует ли внешнее хранилище данных на момент определения. И это может быть полезно, потому что данные могут быть созданы позднее, уже после создания таблицы.

При удалении внешней таблицы Hive удаляет только метаданные, а данные остаются.

¹ Перемещение будет выполнено успешно только при совпадении исходной и приемной файловых систем. Ключевое слово `LOCAL` является особым случаем: Hive *копирует* данные из локальной файловой системы в каталог хранения информации Hive, даже если он тоже находится в той же локальной файловой системе. Во всех остальных случаях операция `LOAD` является операцией перемещения, и ее следует рассматривать именно таким образом.

Как же выбрать тип таблицы? Обычно особых различий между типами не видно (конечно, не считая различий в семантике `DROP`), так что выбор является делом вкуса. Как правило, если вся обработка данных выполняется в Hive, стоит выбирать управляемые таблицы, а если с одним набором используется и Hive, и другие программы — выбирайте внешние таблицы. На практике внешняя таблица часто используется для обращения к исходному набору данных, хранящемуся в HDFS (созданному другим процессом), с последующим применением преобразования Hive для перемещения данных в управляемую таблицу Hive. Процесс работает и в обратном направлении; внешняя таблица (не обязательно находящаяся в HDFS) может использоваться для экспортации данных из Hive для других приложений¹.

Внешние таблицы также применяются в ситуациях, когда требуется связать несколько схем с одним набором данных.

Разделы и гнезда

Hive создает в таблицах *разделы* (partitions); этот механизм позволяет разделить таблицу на блоки по значению некоторого столбца (например, даты). Разделы ускоряют выполнение запросов.

Таблицы или разделы могут дополнительно делиться на *гнезда* (buckets), обеспечивающие дополнительное структурирование данных для повышения эффективности запросов. Например, деление на гнезда по идентификатору пользователя позволяет быстро выполнить запрос для случайной выборки из общего множества пользователей.

Разделы

Представьте себе журнальный файл, в котором каждая запись содержит временную метку. Если разделить данные по дате, то записи с одинаковой датой будут находиться в одном разделе. Преимущество такой схемы состоит в том, что запросы, относящиеся к определенной дате или набору дат, будут обрабатываться значительно эффективнее — поиск будет осуществляться только в файлах разделов, к которым относится запрос. При этом разделы не препятствуют выполнению более общих запросов ко всему набору данных по многим разделам.

¹ Конструкция `INSERT OVERWRITE DIRECTORY` может использоваться для экспортации данных в файловую систему Hadoop, но, в отличие от внешних таблиц, вы не можете управлять выходным форматом — текстовыми файлами с разделителем `Control+A`. Сложные типы данных сериализуются с использованием представления JSON.

Разделы могут определяться для таблицы по нескольким «измерениям». Например, в каждом разделе по дате также могут определяться дополнительные подразделы по странам, повышающие эффективность запросов.

Разделы определяются в момент создания таблицы¹ при помощи условия **PARTITIONED BY** со списком определений столбцов. В нашем гипотетическом примере с журнальными файлами можно определить таблицу с записями, состоящими из временной метки и строки сообщения:

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

При загрузке данных в таблицу с разделами значения задаются явно:

```
LOAD DATA LOCAL INPATH 'input/hive/partitions/file1'
INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

На уровне файловой системы разделы представляются вложенными подкаталогами в каталоге таблицы.

После загрузки еще нескольких файлов в таблицу `logs` структура каталогов может выглядеть примерно так:

```
/user/hive/warehouse/logs
├── dt=2001-01-01/
│   ├── country=GB/
│   │   ├── file1
│   │   └── file2
│   └── country=US/
│       └── file3
└── dt=2001-01-02/
    ├── country=GB/
    │   └── file4
    └── country=US/
        ├── file5
        └── file6
```

Таблица `logs` содержит два раздела дат, `2001-01-01` и `2001-01-02`, соответствующие подкаталогам `dt=2001-01-01` и `dt=2001-01-02`, а также два подраздела стран, `GB` и `US`, соответствующие вложенным подкаталогам `country=GB` и `country=US`. Файлы данных хранятся в листовых каталогах.

¹ Разделы также можно добавлять или удалять после создания таблицы инструкцией `ALTER TABLE`.

Данные из разделов таблицы запрашиваются командой `SHOW PARTITIONS`:

```
hive> SHOW PARTITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-02/country=GB
dt=2001-01-02/country=US
```

Следует помнить, что определения столбцов в разделе `PARTITIONED BY` представляют собой полноценные столбцы, называемые *столбцами разделов*; однако в файлах данных значения этих столбцов отсутствуют, поскольку они определяются по именам каталогов.

Столбцы разделов могут использоваться в инструкциях `SELECT` обычным образом. Hive проверяет входные данные и просматривает только нужные разделы. Например, инструкция

```
SELECT ts, dt, line
FROM logs
WHERE country='GB';
```

просканирует только файлы *file1*, *file2* и *file4*. Значения столбца `dt`, возвращенные запросом, Hive прочитает из имен каталогов, так как они не хранятся в файлах данных.

Гнезда

Существует две причины для определения *гнезд* (buckets) в таблицах или разделах. Первая причина — повышение эффективности запросов. Гнезда образуют в таблице дополнительную структуру, которая может использоваться при выполнении некоторых запросов. В частности, соединение двух таблиц, в которых определены гнезда по одинаковым столбцам (включающим в себя столбцы соединения), эффективно реализуется в форме соединения на стороне отображения.

Вторая причина — повышение эффективности выборки. Если вы работаете с большими наборами данных, в процессе разработки или уточнения запроса бывает очень полезно опробовать его на небольшом подмножестве набора данных. О том, как эффективно организовать выборку, рассказано в конце этого раздела.

Но сначала давайте посмотрим, как сообщить Hive о том, что в таблице следует определить гнезда. Столбцы для определения гнезд и их количество задаются в условии `CLUSTERED BY`:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

Идентификатор пользователя используется для определения гнезда (которое Hive реализует хешированием значения и вычислением остатка от деления на количество гнезд), так что любое конкретное гнездо фактически содержит случайный набор пользователей.

В случае соединения на стороне отображения, если две таблицы используют одинаковое гнездовое разбиение, задача отображения, обрабатывающая гнездо левой таблицы, знает, что соответствующие строки правой таблицы находятся в соответствующем гнезде, поэтому для выполнения соединения ей достаточно произвести выборку только этого гнезда (в котором хранится лишь малая часть данных, хранящихся в правой таблице). Такая оптимизация работает и в том случае, если количество гнезд одной таблицы кратно количеству гнезд другой таблицы; количества гнезд не обязаны в точности совпадать. Код HiveQL для соединения двух гнездовых таблиц приведен в разделе «Соединение в памяти», с. 566.

Данные в гнездах могут дополнительно сортироваться по одному или нескольким столбцам. Это позволит дополнительно повысить эффективность соединений на стороне отображения, так как соединение по каждому гнезду выполняется как эффективная сортировка слиянием. Синтаксис объявления таблицы с отсортированными гнездами выглядит так:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

Как обеспечить гнездовое разбиение данных таблицы? Можно загрузить в таблицу данные, сгенерированные за пределами Hive, но обычно бывает проще поручить гнездовое разбиение Hive (особенно для уже существующей таблицы).



Hive не проверяет соответствие гнезд в файлах данных на диске с гнездами в определении таблицы (по количеству или составу столбцов). В случае несоответствия запрос приведет к ошибке или непредсказуемому поведению, поэтому выполнение операции лучше поручить Hive.

Допустим, имеется следующая таблица без гнезд:

```
hive> SELECT * FROM users;
0    Nat
2    Joe
3    Kay
4    Ann
```

Чтобы заполнить таблицу с гнездами, необходимо задать свойству `hive.enforce.bucketing` значение `true`; Hive создаст количество гнезд, объявленное в определении таблицы. Дальше остается только выполнить команду `INSERT`:

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

Физически каждое гнездо представляет собой файл в каталоге таблицы (или раздела). Имя файла несущественно, но гнезду n в лексикографическом порядке соответствует n -й файл. Собственно, гнезда соответствуют разделам выходных файлов MapReduce: задание создает столько гнезд (выходных файлов), сколько существует задач свертки. Чтобы убедиться в этом, достаточно взглянуть на структуру только что созданной нами таблицы `bucketed_users`. Команда

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

показывает, что были созданы четыре файла со следующими именами (сгенерированными Hive):

```
000000_0
000001_0
000002_0
000003_0
```

В первое гнездо входят пользователи с идентификаторами 0 и 4, так как для INT хеш-код равен самому целому числу, а значение вычисляется как остаток от деления на количество гнезд — 4 в данном случае¹:

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
0Nat
4Ann
```

Тот же результат можно получить при использовании условия TABLESAMPLE, ограничивающего запрос частью гнезд таблицы (вместо всей таблицы):

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
0    Nat
4    Ann
```

Нумерация гнезд начинается с 1, поэтому запрос возвращает всех пользователей, находящихся в первом из четырех гнезд. Для большого, равномерно распределенного набора данных будет возвращена приблизительно одна четверть строк таблицы. Также можно осуществить выборку из нескольких гнезд, указав другую пропорцию (которая не обязана быть в точности кратной количеству гнезд, так

¹ При отображении поля словно следуют вплотную друг к другу, потому что в качестве разделителя в выходных данных используется непечатаемыйправляющий символ (см. следующий раздел).

как выборка не обязана быть точной операцией). Например, следующий запрос возвращает половину гнезд:

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
0    Nat
4    Ann
2    Joe
```

Выборка осуществляется очень эффективно, потому что запросу достаточно прочитать только те гнезда, которые соответствуют условию TABLESAMPLE. Сравните с выборкой из таблицы без гнезд с использованием функции `rand()`; в этом случае сканируется весь входной набор, хотя реально необходимо лишь небольшое подмножество:

```
hive> SELECT * FROM users
    > TABLESAMPLE(BUCKET 1 OUT OF 4 ON rand());
2    Joe
```

Форматы хранения данных

Формат хранения таблиц в Hive имеет два «измерения»: формат строк и формат файлов. Формат строк определяет способ хранения строк и полей конкретной строки. В контексте Hive формат строк определяется реализацией интерфейса `SerDe` (`SERIALIZER/DESERIALIZER`, то есть «сериализатор/десериализатор»).

При выполнении функции десериализатора (при запросе к таблице) `SerDe` десериализует строку данных из байтов в файле в объекты, используемые во внутренней реализации Hive для выполнения операций с этой строкой данных. В режиме сериализатора (при выполнении инструкции `INSERT` или `CTAS` — см. «Импортирование данных», с. 557) реализация `SerDe` таблицы сериализует внутреннее представление строки данных Hive в байты, записываемые в выходной файл.

Формат файлов определяет формат контейнера полей строки данных. Простейший формат — простой текстовый файл, однако доступны и двоичные форматы, ориентированные на строки и столбцы.

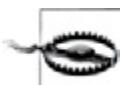
Формат по умолчанию: текст с разделителями

При создании таблицы без секций `ROW FORMAT` или `STORED AS` по умолчанию используется формат текста с разделителями, по одной строке данных на строку текста.

По умолчанию в качестве разделителя используется не символ табуляции, а символ `Control+A` из набора управляющих ASCII-кодов (ASCII-код 1). Выбор

символа Control+A, иногда обозначаемого в документации ^A, объясняется тем, что он с меньшей вероятностью встречается в тексте полей, чем символ табуляции. В Hive не предусмотрен механизм экранирования (escaping) символов-разделителей, поэтому важно выбрать символ, не встречающийся в полях данных.

Для разделения элементов коллекций (элементы **ARRAY** или **STRUCT**, пары «ключ-значение» в **MAP**) по умолчанию используется символ Control+B, а для разделения ключа и значения в **MAP** — символ Control+C. Строки в таблицах разделяются символом новой строки.



Приведенное описание разделителей относится к стандартным «плоским» структурам данных, в которых составные типы содержат только примитивные типы. Для вложенных типов оно не дает полной картины; разделитель определяется уровнем вложенности.

Например, для массива массивов разделителями внешнего массива, как и ожидается, являются символы Control+B, а во внутреннем массиве используются символы Control+C (следующий разделитель из списка). Если вы не уверены в том, какой символ Hive будет использовать для конкретной вложенной структуры, выполните команду вида

```
CREATE TABLE nested
AS
SELECT array(array(1, 2), array(3, 4))
FROM dummy;
```

после чего при помощи hexdump или аналогичной программы определите разделители в выходном файле.

Hive поддерживает только восемь уровней разделителей, соответствующих ASCII-кодам 1, 2, ..., 8, но переопределить можно только первые три.

Итак, инструкция:

```
CREATE TABLE ...;
```

эквивалентна более подробной инструкции:

```
CREATE TABLE ...
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Обратите внимание на возможность использования восьмеричного представления разделителей – 001 for Control+A, например.

Во внутреннем представлении Hive использует для этого формата реализацию SerDe с именем `LazySimpleSerDe`, наряду со строчно-ориентированными текстовыми форматами входных и выходных данных, представленными в главе 7. Префикс `Lazy` («отложенный») объясняется тем, что десериализация полей осуществляется в отложенном режиме — только при обращении. Однако данный формат не является компактным, потому что поля хранятся в тестовом формате; таким образом, логическое значение, например, записывается в виде лiteralной строки `true` или `false`.

Простота формата обладает многими достоинствами (например, этот формат легко обрабатывается другими средствами, включая программы MapReduce и Streaming), однако существуют и более компактные и производительные двоичные реализации SerDe. Некоторые из них перечислены в табл. 12.4.



Двоичные реализации SerDe не должны использоваться с форматом `TEXTFILE` (по умолчанию или с явным указанием условия `STORED AS TEXTFILE`). Всегда существует вероятность того, что в двоичных данных присутствует символ новой строки, из-за которого Hive усечет данные с последующим сбоем во время десериализации.

Таблица 12.4. Реализации SerDe в Hive

Имя	Пакет Java	Описание
<code>LazySimpleSerDe</code>	<code>org.apache.hadoop.hive.serde2.lazy</code>	Реализация SerDe по умолчанию. Текстовый формат с разделителями, отложенный доступ к полям
<code>LazyBinarySerDe</code>	<code>org.apache.hadoop.hive.serde2.lazybinary</code>	Более эффективная версия <code>LazySimpleSerDe</code> . Двоичный формат с отложенным доступом к полям. Используется во внутренних операциях — например, для временных таблиц
<code>BinarySortableSerDe</code>	<code>org.apache.hadoop.hive.serde2.binarysortable</code>	Двоичная реализация, как и <code>LazyBinarySerDe</code> , но оптимизированная для сортировки в ущерб компактности (хотя и остается намного более компактной, чем <code>LazySimpleSerDe</code>)

Имя	Пакет Java	Описание
ColumnarSerDe	org.apache.hadoop.hive.serde2.columnar	Разновидность LazySimpleSerDe для столбцового формата хранения с RCFile
RegexSerDe	org.apache.hadoop.hive.contrib.serde2	Реализация SerDe для чтения текстовых данных, в которых столбцы задаются регулярным выражением. Также поддерживает запись данных с использованием выражения форматирования. Реализация полезна для чтения файлов журналов, но недостаточно эффективна, и не подходит для задач общего назначения
ThriftByteStreamTypedSerDe	org.apache.hadoop.hive.serde2.thrift	Реализация SerDe для чтения двоичных данных в кодировке Thrift
HBaseSerDe	org.apache.hadoop.hive.hbase	Реализация SerDe для хранения данных в таблице HBase. См. https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration

Двоичные форматы хранения: SequenceFile, файлы данных Avro и RCFile

Формат последовательных файлов Hadoop (см. «SequenceFile», с. 186) представляет собой двоичный формат общего назначения для хранения последовательностей записей (пары «ключ-значение»). Чтобы использовать формат последовательных файлов в Hive, включите в инструкцию CREATE TABLE объявление STORED AS SEQUENCEFILE.

Одним из главных преимуществ последовательных файлов является поддержка сжатия с разбиением. Если у вас имеется набор последовательных файлов, созданных вне Hive, то Hive прочитает их без дополнительной настройки. Если же вы хотите, чтобы для хранения таблиц, заполненных в Hive, использовались сжатые последовательные файлы, необходимо задать несколько свойств для включения сжатия (см. «Использование сжатия в MapReduce», с. 137):

```
hive> CREATE TABLE compressed_users (id INT, name STRING)
      > STORED AS SEQUENCEFILE;
hive> SET hive.exec.compress.output=true;
```

продолжение ↗

```
hive> SET mapred.output.compress=true;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.
GzipCodec;
hive> INSERT OVERWRITE TABLE compressed_users
> SELECT * FROM users;
```

Последовательные файлы являются строчно-ориентированными, то есть поля каждой строки данных хранятся вместе как содержимое одной записи последовательного файла.

Файлы данных Avro во многом похожи на последовательные файлы (разбиение, сжатие, строковая ориентация и т. д.), но они поддерживают эволюцию схем и привязок на нескольких языках (см. «Avro», с. 161). Hive может читать и записывать данные Avro с использованием реализации `SerDe`, которая называется Haivvreo (<https://github.com/jghoman/haivvreo>).

Hive также поддерживает еще один двоичный формат хранения данных, называемый `RCFile` (сокращение от Record Columnar File). Файлы `RCFile` похожи на последовательные файлы, если не считать того, что при хранении данных они используют столбцовую ориентацию. Таблица разбивается на сегменты; внутри каждого сегмента сначала хранятся значения первого столбца каждой строки, за ними следуют значения второго столбца каждой строки и т. д. (рис. 12.3).

Столбцово-ориентированный формат позволяет пропускать столбцы, не задействованные в запросе. Для примера возьмем запрос к таблице на рис. 12.3, который обрабатывает только столбец 2. При строчно-ориентированном хранении в память загружается вся строка данных (хранящаяся в записи последовательного файла), хотя фактически используется только второй столбец. Отложенная десериализация экономит вычислительные ресурсы, так как десериализуются только реально используемые поля, но затрат на чтение байтов каждой строки данных с диска не избежать. При столбцово-ориентированном хранении в память необходимо прочитать только части файла, содержащие данные столбца 2 (выделены рамкой на рис. 12.3).

Как правило, столбцово-ориентированные форматы хорошо работают для запросов, обращающихся к небольшому подмножеству столбцов таблицы. И наоборот, строчно-ориентированные форматы уместны тогда, когда для обработки необходимо большое количество столбцов одной строки. Если свободное пространство позволяет, можно относительно легко сравнить по скорости два формата для ваших рабочих данных — создайте копию таблицы с другим форматом хранения, используя инструкцию `CREATE TABLE...AS SELECT` (с. 558).

Для применения столбцово-ориентированного хранения в Hive используется инструмент `CREATE TABLE` следующего вида:

```
CREATE TABLE ...
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe'
STORED AS RCFILE;
```

Логическая таблица

	столбец1	столбец2	столбец3
строка1	1	2	3
строка2	4	5	6
строка3	7	8	9
строка4	10	11	12

Строчно-ориентированное хранение (SequenceFile)

строка1	строка2	строка3	строка4
1 2 3	4 5 6	7 8 9	10 11 12

Строчно-ориентированное хранение (SequenceFile)

строчный сегмент 1			строчный сегмент 2		
столбец1	столбец2	столбец3	столбец1	столбец2	столбец3
1 4	2 5	3 6	7 10	8 11	9 12

Рис. 12.3. Форматы хранения таблиц: строчно-ориентированный и столбцово-ориентированный

Пример: RegexSerDe

Посмотрим, как использовать для хранения данных другую реализацию SerDe. Мы воспользуемся реализацией с поддержкой регулярных выражений для чтения из текстового файла метаданных станций, имеющих фиксированную ширину:

```
CREATE TABLE stations (usaf STRING, wban STRING, name STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
продолжение ↗
```

```
WITH SERDEPROPERTIES (
    "input.regex" = "(\\d{6}) (\\d{5}) (.{29}) .*"
);
```

В предыдущих примерах для обозначения текста с разделителями в секции `ROW FORMAT` использовалось ключевое слово `DELIMITED`. В нашем примере реализация `SerDe` задается ключевым словом `SERDE` и полным именем класса Java, реализующего `SerDe` — `org.apache.hadoop.hive.contrib.serde2.RegexSerDe`.

Дополнительная настройка `SerDe` осуществляется при помощи дополнительных свойств в секции `WITH SERDEPROPERTIES`. В данном случае мы задаем свойство `input.regex`, специфическое для `RegexSerDe`.

Свойство `input.regex` определяет регулярное выражение, которое должно использоваться в процессе десериализации для преобразования строки текста в набор столбцов. Для поиска совпадений используется синтаксис регулярных выражений Java (см. <http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html>), а столбцы формируются по сохраняющим группам круглых скобок¹. В нашем примере используются три сохраняющие группы для полей `usaf` (идентификатор из 6 цифр), `wban` (идентификатор из пяти цифр) и `name` (столбец с фиксированной шириной 29 символов).

Для заполнения таблицы, как и прежде, используется инструкция `LOAD DATA`:

```
LOAD DATA LOCAL INPATH "input/ncdc/metadata/stations-fixed-width.txt"
INTO TABLE stations;
```

Как говорилось ранее, инструкция `LOAD DATA` копирует или перемещает файлы в каталог Hive (в нашем случае выполняется копирование, потому что источником является локальная файловая система). Реализация `SerDe` таблицы при загрузке не используется.

При выборке данных из таблицы `SerDe` используется для десериализации, как видно из следующего простого запроса, правильно разбирающего поля каждой строки данных:

```
hive> SELECT * FROM stations LIMIT 4;
010000    99999    BOGUS NORWAY
010003    99999    BOGUS NORWAY
010010    99999    JAN MAYEN
010013    99999    ROST
```

¹ Иногда круглые скобки в регулярных выражениях используются для конструкций, которые не должны считаться сохраняющими группами — как, например, в шаблоне `(ab)+`. Проблема решается использованием несохраняющих групп, у которых после открывающей круглой скобки следует вопросительный знак. Также существуют другие несохраняющие групповые конструкции (см. документацию Java), но в нашем примере проще использовать конструкцию `(?:ab)+` для того, чтобы предотвратить сохранение совпадения в столбце Hive.

Импортирование данных

Вы уже знаете, как использовать операцию `LOAD DATA` для импортирования данных в таблицу (или раздел) Hive посредством копирования или перемещения файлов в каталог таблицы. Таблицу также можно заполнить данными из другой таблицы Hive с использованием инструкции `INSERT` или же в момент создания с использованием конструкции `CTAS` (сокращение от `CREATE TABLE...AS SELECT`).

Если вам потребуется импортировать данные из реляционной базы непосредственно в Hive, обратите внимание на программу Sqoop (см. «Импортирование данных и Hive», с. 658).

Вставка

Пример инструкции `INSERT`:

```
INSERT OVERWRITE TABLE target
SELECT col1, col2
  FROM source;
```

Для таблиц с разделами можно задать раздел, в который должны вставляться данные, в секции `PARTITION`:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2001-01-01')
SELECT col1, col2
  FROM source;
```

Ключевое слово `OVERWRITE` означает, что содержимое целевой таблицы (первый пример) или раздел `2001-01-01` (второй пример) заменяется результатами инструкции `SELECT`. Если вам потребуется добавить записи в уже заполненную таблицу без разделов или в раздел, используйте инструкцию `INSERT INTO TABLE`.

Раздел может задаваться динамически, с определением значения из команды `SELECT`:

```
INSERT OVERWRITE TABLE target
PARTITION (dt)
SELECT col1, col2, dt
  FROM source;
```

Такая операция называется *вставкой с динамическим разделом*. Данная возможность отключена по умолчанию, поэтому ее сначала необходимо включить, задав свойству `hive.exec.dynamic.partition` значение `true`.



В отличие от других баз данных, Hive (в настоящее время) не поддерживает разновидность инструкции `INSERT` для вставки набора записей, заданного в запросе в лiteralной форме — другими словами, инструкции вида `INSERT INTO...VALUES...` недопустимы.

Многотабличная вставка

В HiveQL инструкцию `INSERT` с таким же успехом можно начать с секции `FROM`:

```
FROM source
INSERT OVERWRITE TABLE target
    SELECT col1, col2;
```

Причина для введения такого синтаксиса становится очевидной, если учесть, что один запрос может содержать несколько секций `INSERT`. Так называемая многотабличная вставка эффективнее нескольких инструкций `INSERT`, потому что для получения нескольких несвязанных результатов достаточно однократного просмотра исходной таблицы.

Пример вычисления различных статистических характеристик для набора метеорологических данных:

```
FROM records2
INSERT OVERWRITE TABLE stations_by_year
    SELECT year, COUNT(DISTINCT station)
    GROUP BY year
INSERT OVERWRITE TABLE records_by_year
    SELECT year, COUNT(1)
    GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
    SELECT year, COUNT(1)
    WHERE temperature != 9999
        AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
    GROUP BY year;
```

Исходная таблица всего одна (`records2`), но для нее создаются три таблицы с результатами трех разных запросов к одному источнику.

CREATE TABLE...AS SELECT

Часто бывает удобнее сохранить результат запроса Hive в новой таблице — например потому, что он слишком велик для вывода на консоль, или потому, что результат требует дополнительной обработки.

Определения столбцов новой таблицы делаются по столбцам, полученным инструкцией `SELECT`. В следующем запросе таблица `target` содержит два столбца

с именами `col1` и `col2`, типы которых совпадают с типами столбцов исходной таблицы:

```
CREATE TABLE target
AS
SELECT col1, col2
FROM source;
```

Операция CTAS является атомарной, и, если запрос `SELECT` по какой-то причине завершится неудачей, таблица создана не будет.

Модификация таблиц

Гибкость метода проверки схемы при чтении, используемого Hive, позволяет изменить определение таблицы после ее создания. Однако следует учесть, что во многих случаях вам придется позаботиться о модификации данных в соответствии с новой структурой.

Для переименования таблицы используется инструкция `ALTER TABLE`:

```
ALTER TABLE source RENAME TO target;
```

Кроме обновления метаданных таблицы, `ALTER TABLE` переименовывает базовый каталог таблицы в соответствии с новым именем. В нашем примере `/user/hive/warehouse/` переименовывается в `/user/hive/warehouse/target`. (Базовый каталог внешней таблицы не перемещается; обновляются только метаданные.)

Hive позволяет изменять определения столбцов, добавлять новые столбцы и даже заменять все существующие столбцы таблицы новым набором столбцов.

Например, добавление нового столбца выполняется так:

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

Новый столбец `col3` добавляется после существующих (не входящих в разделы) столбцов. Файлы данных не обновляются, поэтому запросы в качестве всех значений `col3` будут возвращать `null` (если, конечно, дополнительные поля не присутствуют в файле). Так как Hive не поддерживает обновление существующих записей, вам придется организовать обновление базовых файлов другими средствами, поэтому на практике чаще всего создается новая таблица, которая определяет новые столбцы и заполняет их инструкцией `SELECT`.

Изменение метаданных столбца (например, имени или типа данных) выполняется проще — при условии, что старый тип данных может интерпретироваться как новый тип данных.

За дополнительной информацией об изменении структуры таблицы, добавлении и удалении разделов, изменении и замене столбцов, модификации свойств таблиц и SerDe, обращайтесь к вики Hive по адресу <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL>.

Удаление таблиц

Инструкция `DROP TABLE` удаляет данные и метаданные таблицы. В случае внешних таблиц удаляются только метаданные, а данные остаются.

Если вы хотите удалить все данные таблицы, но сохранить ее определение (по аналогии с `DELETE` или `TRUNCATE` в MySQL), просто удалите файлы данных. Например:

```
hive>
dfs -rmdir /user/hive/warehouse/my_table;
```

Hive интерпретирует отсутствие файлов (или даже каталога таблицы) как пустую таблицу.

Добиться того же результата можно созданием новой, пустой таблицы с такой же схемой, как у первой, с использованием ключевого слова `LIKE`:

```
CREATE TABLE new_table LIKE existing_table;
```

Запросы к данным

В этом разделе рассматриваются различные формы инструкции `SELECT` для выборки данных из Hive.

Сортировка и агрегирование

Сортировка данных в Hive может быть реализована стандартным условием `ORDER BY`, но здесь кроется ловушка. `ORDER BY`, как и ожидалось, создает полностью отсортированный результат, но при этом устанавливает количество сверток равным 1, вследствие чего такое решение крайне неэффективно для больших наборов данных. (Ожидается, что в будущих версиях Hive будут применены механизмы, описанные в разделе «Полная сортировка» на с. 357, для поддержки эффективной параллельной сортировки.)

Если глобальная сортировка не нужна (а это бывает достаточно часто), можно воспользоваться нестандартным расширением Hive `SORT BY`, которое создает по отсортированному файлу на задачу свертки.

Иногда требуется управлять тем, в какую свертку попадет конкретная строка данных — чаще всего для выполнения последующего обобщения. В этом вам поможет секция Hive `DISTRIBUTE BY`. В следующем примере метеорологические данные сортируются по году и температуре таким образом, чтобы все записи года попали в один раздел свертки¹:

```
hive> FROM records2
    > SELECT year, temperature
    > DISTRIBUTE BY year
    > SORT BY year ASC, temperature DESC;
1949    111
1949     78
1950     22
1950      0
1950    -11
```

Следующий запрос (или запрос, который использует данный как подзапрос; см. «Подзапросы», с. 566) сможет использовать тот факт, что температуры за каждый год были сгруппированы и отсортированы (по убыванию) в одном файле.

Если столбцы `SORT BY` и `DISTRIBUTE BY` совпадают, используйте сокращенную запись `CLUSTER BY`.

Сценарии MapReduce

Секции `TRANSFORM`, `MAP` и `REDUCE` предоставляют возможность запуска внешних сценариев или программ из Hive; принцип их работы отчасти напоминает Hadoop Streaming. Предположим, вы хотите при помощи сценария отфильтровать записи, не удовлетворяющие некоторому условию, — как это делает листинг 12.1, удаляющий записи с плохим показателем качества данных.

Листинг 12.1. Сценарий Python для исключения метеорологических данных с низким качеством

```
#!/usr/bin/env python
import re
import sys
for line in sys.stdin:
    (year, temp, q) = line.strip().split()
    if (temp != "9999" and re.match("[01459]", q)):
        print "%s\t%s" % (year, temp)
```

¹ См. «Вторичная сортировка», с. 361.

Использование сценария продемонстрировано в листинге 12.2:

Листинг 12.2. Использование сценария Python

```
hive> ADD FILE /Users/tom/book-workspace/hadoop-book/ch12/src/main/python
      /is_good_quality.py;
hive> FROM records2
      > SELECT TRANSFORM(year, temperature, quality)
      > USING 'is_good_quality.py'
      > AS year, temperature;
1950    0
1950    22
1950   -11
1949   111
1949   78
```

Прежде чем запускать сценарий, необходимо зарегистрировать его в Hive — этим вы сообщите Hive о том, что сценарий необходимо доставить в кластер Hadoop (см. «Распределенный кэш», с. 375).

Сам запрос передает поля `year`, `temperature` и `quality` сценарию `is_good_quality.py` в виде строки, разделенной табуляциями, и выделяет из результата поля `year` и `temperature` для формирования вывода запроса.

В этом примере свертки не используются. При использовании вложенной формы запроса мы могли бы задать функции отображения и свертки. В данном случае мы используем ключевые слова `MAP` и `REDUCE`, но в обоих случаях инструкция `SELECT TRANSFORM` привела бы к тому же результату. Исходный код сценария `max_temperature_reduce.py` приведен в листинге 2.11:

```
FROM (
  FROM records2
  MAP year, temperature, quality
  USING 'is_good_quality.py'
  AS year, temperature) map_output
REDUCE year, temperature
USING 'max_temperature_reduce.py'
AS year, temperature;
```

Соединения

Одним из достоинств Hive по сравнению с низкоуровневыми программами MapReduce является то, что Hive упрощает выполнение многих стандартных операций. Хорошим примером служат операции соединения — особенно если вспомнить, сколько проблем возникает с их реализацией в MapReduce (см. «Соединения», с. 368).

Внутренние соединения

Простейшая разновидность соединений — внутреннее соединение, при котором для каждого совпадения во входных таблицах создается строка выходных данных. Для примера возьмем две таблицы: в таблице `sales` перечисляются имена людей и идентификаторы купленных ими предметов, а в таблице `things` — идентификаторы и названия предметов:

```
hive> SELECT * FROM sales;
Joe    2
Hank   4
Ali    0
Eve    3
Hank   2
hive> SELECT * FROM things;
2     Tie
4     Coat
3     Hat
1     Scarf
```

Внутреннее соединение двух таблиц выполняется следующим образом:

```
hive> SELECT sales.* , things.* 
      > FROM sales JOIN things ON (sales.id = things.id);
Joe    2    2    Tie
Hank   2    2    Tie
Eve    3    3    Hat
Hank   4    4    Coat
```

Таблица в секции `FROM (sales)` соединяется с таблицей в секции `JOIN (things)`, с использованием предиката из секции `ON`. Hive поддерживает только эквисоединения, то есть в предикате может использоваться только условие равенства (в нашем примере равенства столбцов `id` в обеих таблицах).



Некоторые базы данных — например, MySQL и Oracle — позволяют соединять таблицы в секции `FROM`, с указанием условия соединения в секции `WHERE` инструкции `SELECT`. Однако в Hive такой синтаксис не поддерживается, поэтому при попытке выполнения следующего фрагмента произойдет ошибка:

```
SELECT sales.* , things.* 
  FROM sales, things
 WHERE sales.id = things.id;
```

В Hive секция `FROM` должна содержать только одну таблицу, а соединение должно выполняться в соответствии с синтаксисом `JOIN` из стандарта SQL-92.

Hive позволяет выполнить соединение по нескольким столбцам в предикате соединения, для чего указывается серия выражений, связанных ключевыми словами `AND`. Также можно соединить более двух таблиц, включив в запрос дополнительные секции `JOIN...ON....` Hive стремится свести к минимуму количество заданий MapReduce для реализации соединений.

Простое соединение реализуется как одиночное задание MapReduce, но множественные соединения могут выполняться менее чем в одном задании, если в условии соединения используются одинаковые столбцы¹. Чтобы увидеть, сколько заданий MapReduce Hive использует для конкретного запроса, начните его с префикса `EXPLAIN`:

```
EXPLAIN  
SELECT sales.* , things.*  
FROM sales JOIN things ON (sales.id = things.id);
```

Вывод `EXPLAIN` содержит многочисленные подробности о плане исполнения запроса, включая дерево абстрактного синтаксиса, граф зависимости фаз и информацию о каждой фазе. Фазами могут быть задания MapReduce или такие операции, как перемещения файлов. Если вам потребуется еще более подробная информация, начните запрос с префикса `EXPLAIN EXTENDED`.

В настоящее время Hive использует оптимизатор запросов на основе правил, но возможно, в будущем к нему добавится оптимизатор на основе затрат.

Внешние соединения

Внешние соединения позволяют находить несовпадения в соединяемых таблицах. В нашем примере при выполнении внутреннего соединения строка для `Ali` не была включена в вывод, потому что соответствующий идентификатор отсутствовал в таблице `things`. Если заменить тип соединения на `LEFT OUTER JOIN`, запрос вернет строку для каждой строки в левой таблице даже в том случае, если у нее нет соответствующей строки в таблице, с которой она соединяется (`things`):

```
hive> SELECT sales.* , things.*  
> FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);  
Ali      0      NULL    NULL  
Joe      2      2      Tie  
Hank     2      2      Tie  
Eve      3      3      Hat  
Hank     4      4      Coat
```

¹ Порядок следования таблиц в секциях `JOIN` важен. Обычно лучше расположить самую большую таблицу на последнем месте, но по адресу <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins> вы найдете более подробную информацию (в том числе и о том, как дать рекомендации планировщику Hive).

Обратите внимание: строка с именем Ali присутствует в выводе, а столбцы таблицы `things` содержат NULL из-за отсутствия совпадения.

Hive поддерживает правые внешние соединения, в которых таблицы меняются ролями относительно левого соединения. В этом случае включаются все данные из таблицы `things` — даже те, у которых нет покупателя из таблицы `sales`:

```
hive> SELECT sales.* , things.*  
    > FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);  
NULL    NULL 1      Scarf  
Joe     2      2      Tie  
Hank    2      2      Tie  
Eve     3      3      Hat  
Hank    4      4      Coat
```

Наконец, в полном внешнем соединении в выходные данные включается строка для каждой строки из обеих таблиц:

```
hive> SELECT sales.* , things.*  
    > FROM sales FULL OUTER JOIN things ON (sales.id = things.id);  
Ali     0      NULL NULL  
NULL    NULL 1      Scarf  
Joe     2      2      Tie  
Hank    2      2      Tie  
Eve     3      3      Hat  
Hank    4      4      Coat
```

Полусоединения

Hive не поддерживает подзапросы IN (на момент написания книги), но левое полу-соединение (`LEFT SEMI JOIN`) позволяет добиться того же результата.

Возьмем подзапрос IN, находящий все предметы из таблицы `things`, присутствующие в таблице `sales`:

```
SELECT *  
FROM things  
WHERE things.id IN (SELECT id from sales);
```

Его можно переписать следующим образом:

```
hive> SELECT *  
    > FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);  
2      Tie  
3      Hat  
4      Coat
```

Для запросов `LEFT SEMI JOIN` действует ограничение: правая таблица (`sales`) может присутствовать только в секции `ON` (например, она не может упоминаться в выражении `SELECT`).

Соединение в памяти

Если одна таблица достаточно мала, чтобы поместиться в памяти, Hive может загрузить ее в память для выполнения соединения в каждой задаче отображения. Синтаксис такого соединения встраивается в комментарий в стиле C:

```
hive> SELECT /*+ MAPJOIN(things) */ sales.* , things.*  
    > FROM sales JOIN things ON (sales.id = things.id);  
Joe    2      2      Tie  
Hank   4      4      Coat  
Eve    3      3      Hat  
Hank   2      2      Tie
```

Задание для выполнения запроса не имеет сверток, поэтому запрос не будет работать для соединений `RIGHT` или `FULL OUTER JOIN` — отсутствие соответствий может быть обнаружено только в процессе агрегирования (то есть свертки) по всем входным данным.

Соединения в памяти могут использовать преимущества гнездовых таблиц (см. «Гнезда», с. 547), поскольку задача отображения, работающей с гнездом левой таблицы, для выполнения соединения достаточно загрузить только соответствующие гнезда правой таблицы. Синтаксис выглядит так же, как для приведенного выше соединения в памяти, однако оптимизацию необходимо дополнительно включить следующей командой:

```
SET hive.optimize.bucketmapjoin=true;
```

Подзапросы

Подзапрос представляет собой инструкцию `SELECT`, встроенную в другую инструкцию SQL. Поддержка подзапросов в Hive ограничена: они разрешены только в секции `FROM` инструкции `SELECT`.



Другие базы данных допускают подзапросы практически везде, где разрешены выражения — например, в списке значений, получаемых из инструкции `SELECT` или в секции `WHERE`. Во многих случаях подзапросы могут быть переписаны как соединения, поэтому, если вам когда-либо потребуется использовать подзапрос там, где Hive его не поддерживает, проверьте, нельзя ли его выразить в виде соединения. Например, подзапрос `IN` можно переписать в виде полусоединения или внутреннего соединения (см. «Соединения», с. 368).

Следующий запрос находит среднюю максимальную температуру по всем годам и метеорологическим станциям:

```
SELECT station, year, AVG(max_temperature)
FROM (
    SELECT station, year, MAX(temperature) AS max_temperature
    FROM records2
    WHERE temperature != 9999
        AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
    GROUP BY station, year
) mt
GROUP BY station, year;
```

Подзапрос используется для определения максимальной температуры для каждой комбинации «метеостанция/дата», после чего внешний запрос использует агрегатную функцию `AVG` для вычисления среднего значения максимальных температур по всем комбинациям.

Внешний запрос обращается к результатам подзапроса как к таблице, поэтому подзапросу должен быть назначен псевдоним (`mt`). Столбцы подзапроса должны иметь уникальные имена, чтобы они могли использоваться во внешнем запросе.

Представления

Представление (view) — своего рода «виртуальная таблица», определяемая инструкцией `SELECT`. Представления могут использоваться для отображения данных в виде, отличном от вида их фактического хранения на диске. Данные из существующих таблиц часто упрощаются и обобщаются некоторым образом, удобным для их последующей обработки. Представления также используются для ограничения доступа пользователей к некоторым таблицам.

В Hive созданное представление не материализуется на диске; вместо этого при выполнении инструкции, содержащей ссылку на представление, выполняется инструкция `SELECT` этого представления. Если представление выполняет масштабные преобразования таблиц базы данных или используется достаточно часто, возможно, вам стоит материализовать его созданием новой таблицы с содержимым представления (см. «`CREATE TABLE...AS SELECT`», с. 558).

Давайте переработаем запрос из предыдущего раздела (для вычисления средней максимальной температуры по всем годам и метеостанциям) с использованием представлений. Начнем с создания представления для действительных записей, то есть записей с приемлемым качеством данных:

```
CREATE VIEW valid_records
AS
SELECT *
FROM records2
WHERE temperature != 9999
    AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9);
```

При создании представления запрос не выполняется, а всего лишь сохраняется в метахранилище. Представления включаются в результаты команды `SHOW TABLES`. Вы можете вывести подробную информацию по конкретному представлению, включая запрос, использованный для его определения, командой `DESCRIBE EXTENDED имя_представления`.

Затем создается второе представление с максимальными температурами для каждой станции и года. Оно базируется на представлении `valid_records`:

```
CREATE VIEW max_temperatures (station, year, max_temperature)
AS
SELECT station, year, MAX(temperature)
FROM valid_records
GROUP BY station, year;
```

В этом определении имена столбцов задаются явно. Дело в том, что столбец максимальной температуры является агрегатным выражением и в противном случае Hive генерирует псевдоним столбца (например, `_c2`). Для присваивания имени столбцу с таким же успехом можно воспользоваться секцией `AS` в инструкции `SELECT`.

Когда представления будут готовы, можно использовать их при выполнении запроса:

```
SELECT station, year, AVG(max_temperature)
FROM max_temperatures
GROUP BY station, year;
```

Запрос выдает такой же результат, что и запрос, использующий подзапрос. В частности, Hive создает для обоих запросов одинаковое количество заданий MapReduce: два в каждом случае, по одному для каждого столбца в `GROUP BY`. Данный пример показывает, что Hive может объединить запрос к представлению в последовательность заданий, эквивалентных написанию запроса без представления. Иначе говоря, Hive не станет без необходимости материализовать запрос, даже во время выполнения. Представления в Hive доступны только для чтения, поэтому вставка или загрузка данных в базовую таблицу через представление невозможна.

Пользовательские функции

Иногда нужный запрос слишком трудно (или невозможно) выразить при помощи встроенных функций Hive. Hive позволяет легко подключить ваш собственный код в виде пользовательских функций (UDF, User-Defined Functions) и вызывать его из запроса Hive.

Пользовательские функции должны быть написаны на Java — языке, на котором написан код Hive. Для других языков стоит воспользоваться запросом `SELECT TRANSFORM`, позволяющим пропустить данные через пользовательский сценарий (см. «Сценарии MapReduce», с. 561).

Пользовательские функции в Hive делятся на три типа: обычные (UDF), пользовательские агрегатные функции (UDAF, User-Defined Aggregate Functions) и пользовательские функции генерирования таблиц (UDTF, User-Defined Table Generating Functions). Они различаются по количеству строк данных, получаемых на входе и производимых на выходе:

- UDF получает одну строку входных данных и создает одну строку выходных данных. Большинство функций, включая математические и строковые функции, относится к этому типу.
- UDAF получает одну строку входных данных и создает одну строку выходных данных. К категории агрегатных функций относятся такие функции, как `COUNT` и `MAX`.
- UDTF получает одну строку входных данных и создает несколько строк выходных данных — таблицу.

Функции генерирования таблиц менее известны, чем две другие категории; давайте рассмотрим пример. Возьмем таблицу с одним столбцом `x`, содержащим массивы строк. Будет полезно взглянуть, как определяется и заполняется эта таблица:

```
CREATE TABLE arrays (x ARRAY<STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002';
```

Обратите внимание: в секции `ROW FORMAT` указано, что элементы массива разделяются символами Control+B. Файл, который мы загрузим, содержит следующие данные (^B — представление символа Control+B, пригодное для печати):

```
a^Bb
c^Bd^Be
```

После выполнения команды `LOAD DATA` следующий запрос подтверждает, что данные были загружены правильно:

```
hive> SELECT * FROM arrays;
["a","b"]
["c","d","e"]
```

Далее мы используем UDTF `explode` для преобразования таблицы. Эта функция выводит строку данных для каждого элемента массива; таким образом, в нашем случае выходной столбец `u` будет иметь тип `STRING`. В результате таблица превращается в пять строк:

```
hive> SELECT explode(x) AS y FROM arrays;
a
b
c
d
e
```

Для инструкций `SELECT`, использующих UDTF, устанавливаются некоторые ограничения, которые снижают их практическую ценность. По этой причине Hive поддерживает запросы `LATERAL VIEW`, обладающие более широкими возможностями. Запросы `LATERAL VIEW` здесь не рассматриваются, но вы можете найти дополнительную информацию о них по адресу <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+LateralView>.

Написание пользовательской функции

Для демонстрации процесса написания и использования пользовательских функций мы напишем простую функцию `strip` для отсечения символов с концов строк. Код класса Java `Strip` приведен в листинге 12.3.

Листинг 12.3. Пользовательская функция для отсечения символов с концов строк

```
package com.hadoopbook.hive;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class Strip extends UDF {
    private Text result = new Text();

    public Text evaluate(Text str) {
        if (str == null) {
            return null;
        }
        result.set(StringUtils.strip(str.toString()));
        return result;
    }
    public Text evaluate(Text str, String stripChars) {
        if (str == null) {
            return null;
        }
        result.set(StringUtils.strip(str.toString(), stripChars));
        return result;
    }
}
```

Класс пользовательской функции должен удовлетворять двум условиям:

- Он должен быть субклассом `org.apache.hadoop.hive.ql.exec.UDF`.
- Он должен реализовать хотя бы один метод `evaluate()`.

Метод `evaluate()` не определяется интерфейсом, так как он может получать произвольное количество аргументов произвольных типов, а также возвращать значение произвольного типа. Hive анализирует класс UDF и ищет в нем метод `evaluate()`, соответствующий вызванной функции Hive.

Класс `Strip` содержит два метода `evaluate()`. Первый удаляет из входных данных начальные и конечные пробелы, а второй может удалить произвольный набор заданных символов в конце строки. Фактическая обработка строки поручается классу `StringUtils` из проекта Apache Commons, так что в коде заслуживает внимания разве что использование `Text` из библиотеки Hadoop `Writable`. Вообще говоря, Hive поддерживает примитивы Java в пользовательских функциях (а также другие типы — например, `java.util.List` и `java.util.Map`), поэтому сигнатура вида

```
public String evaluate(String str)
```

тоже будет работать. Однако использование `Text` позволяет использовать объекты повторно, что способствует повышению эффективности, поэтому этот метод в общем случае предпочтителен.

Чтобы использовать UDF в Hive, необходимо упаковать откомпилированный класс Java в файл JAR и зарегистрировать файл в Hive:

```
ADD JAR /path/to/hive-examples.jar;
```

Также необходимо создать псевдоним для имени класса Java:

```
CREATE TEMPORARY FUNCTION strip AS 'com.hadoopbook.hive.Strip';
```

Ключевое слово `TEMPORARY` подчеркивает тот факт, что пользовательские функции создаются только на время сеанса Hive (они не сохраняются в метахранилище). На практике это означает, что вам придется добавить файл JAR и либо определить функцию в начале каждого сценария, либо создать в домашнем каталоге файл `hiverc`, содержащий эти команды.

Теперь пользовательскую функцию можно вызывать, как и любую встроенную функцию:

```
hive> SELECT strip(' bee ') FROM dummy;
bee
hive> SELECT strip('banana', 'ab') FROM dummy;
nan
```

Учтите, что регистр символов в имени пользовательской функции игнорируется:

```
hive> SELECT STRIP(' bee ') FROM dummy;
bee
```

Написание UDAF

Написать агрегатную пользовательскую функцию сложнее, чем обычную. Значения агрегируются во фрагменты (теоретически распространяющиеся на несколько задач отображения или свертки), поэтому реализация должна быть способна объединить частичные агрегатные данные в окончательный результат. Код лучше пояснить на конкретном примере; рассмотрим реализацию простой агрегатной функции для вычисления максимума в наборе целых чисел (листинг 12.4).

Листинг 12.4. Агрегатная функция для вычисления максимума в наборе целых чисел

```
package com.hadoopbook.hive;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAEvaluator;
import org.apache.hadoop.io.IntWritable;

public class Maximum extends UDAF {
    public static class MaximumIntUDAEvaluator implements UDAFEvaluator {

        private IntWritable result;

        public void init() {
            result = null;
        }

        public boolean iterate(IntWritable value) {
            if (value == null) {
                return true;
            }
            if (result == null) {
                result = new IntWritable(value.get());
            } else {
                result.set(Math.max(result.get(), value.get()));
            }
            return true;
        }

        public IntWritable terminatePartial() {
```

```
    return result;
}

public boolean merge(IntWritable other) {
    return iterate(other);
}

public IntWritable terminate() {
    return result;
}
}
```

Структура класса UDAF слегка отличается от структуры классов UDF. Класс UDAF должен субклассировать `org.apache.hadoop.hive.ql.exec.UDAF` (обратите внимание на букву «A» в `UDAF`) и содержать один или несколько вложенных статических классов, реализующих интерфейс `org.apache.hadoop.hive.ql.exec.UDAPEvaluator`. В нашем примере имеется всего один вложенный класс `MaximumIntUDAPEvaluator`, но мы могли бы добавить и другие — `MaximumLongUDAPEvaluator`, `MaximumFloatUDAPEvaluator` и т. д., которые будут предоставлять перегруженные формы UDAF для поиска максимума в наборах `long`, `float` и т. д.

Вычислитель (реализация `UDAPEvaluator`) должен реализовать пять методов, перечисленных ниже (схема передачи управления представлена на рис. 12.4).

`init()`

Метод `init()` инициализирует вычислитель и выполняетброс его внутреннего состояния. Для `MaximumIntUDAPEvaluator` мы задаем объекту `IntWritable`, содержащему окончательный результат, значение `null`. Это значение указывает на то, что агрегирование данных еще не выполнялось (и у него есть полезный побочный эффект: максимальное значение пустого набора равно `NULL`).

`iterate()`

Метод `iterate()` вызывается каждый раз, когда возникает необходимость в агрегировании нового значения. Вычислитель должен обновить свое внутреннее состояние промежуточным результатом. Аргументы, получаемые методом `iterate()`, соответствуют аргументам функции Hive, из которой он был вызван. В нашем примере аргумент всего один. Сначала значение проверяется на `null`, и если проверка окажется положительной — значение игнорируется. В противном случае переменной экземпляра `result` присваивается целое значение `value` (если это первое обрабатываемое значение) или большее из текущих значений `result` и `value` (если одно или несколько значений уже были обработаны). Мы возвращаем `true`, чтобы показать, что входное значение было действительным.

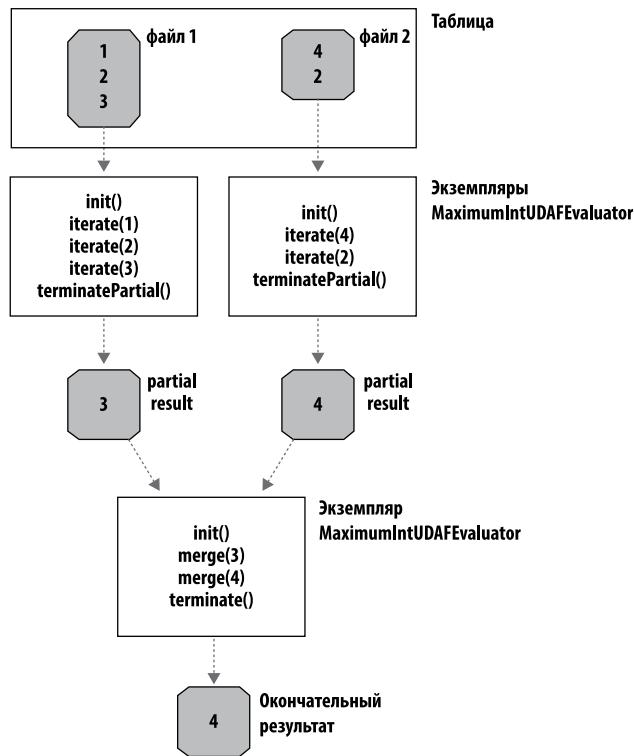


Рис. 12.4. Схема обработки частичных результатов UDAF

`terminatePartial()`

Метод `terminatePartial()` вызывается тогда, когда Hive хочет получить результат для частичного агрегирования. Метод должен вернуть объект, инкапсулирующий состояние агрегирования. В нашем случае достаточно объекта `IntWritable`, так как в нем инкапсулируется либо максимальное обработанное значение, либо `null`, если значения еще не обрабатывались.

`merge()`

Метод `merge()` вызывается тогда, когда Hive решает скомбинировать одно частичное агрегирование с другим. Метод получает один объект, тип которого должен соответствовать возвращаемому значению метода `terminatePartial()`. В нашем примере метод `merge()` может просто делегировать выполнение операции методу `iterate()`, потому что частичное агрегирование представляется так же, как и агрегируемое значение. Обычно это не так (более общий пример будет рассмотрен позднее), а метод должен реализовать логику объединения состояния вычислителя с состоянием частичного агрегирования.

terminate()

Метод `terminate()` вызывается для получения окончательного результата агрегирования. Вычислитель должен вернуть свое состояние как значение. В нашем примере возвращается переменная экземпляра `result`.

Протестируем новую функцию:

```
hive> CREATE TEMPORARY FUNCTION maximum AS 'com.hadoopbook.hive.Maximum';
hive> SELECT maximum(temperature) FROM records;
111
```

Более сложный пример UDAF

Предыдущий пример нетипичен тем, что частичное агрегирование может быть представлено тем же типом (`IntWritable`), что и конечный результат. Для более сложных агрегирующих функций это обычно не так; в этом легко убедиться на примере UDAF для вычисления среднего арифметического набора значений `double`. Математически невозможно объединить частичные средние арифметические в окончательное среднее арифметическое (см. «Комбинирующие функции», с. 68). Вместо этого можно представить частичное агрегирование парой чисел: накопленной суммой значений, обработанных до настоящего момента, и количеством значений.

Идея реализована в листинге 12.5. Обратите внимание: частичное агрегирование реализуется в виде вложенного статического класса `PartialResult`; Hive может сериализовать и десериализовать экземпляры этого типа, потому что мы используем типы полей, которые Hive умеет обрабатывать (примитивы Java в данном случае).

В данном примере метод `merge()` отличен от `iterate()`, потому что он объединяет частичные суммы и частичные счетчики попарным суммированием. Кроме того, возвращаемым типом `terminatePartial()` является `PartialResult` (конечно, он остается невидимым для пользователя, вызывающего функцию), тогда как возвращаемым типом `terminate()` является `DoubleWritable` — окончательный результат, видимый пользователю.

Листинг 12.5. Агрегатная функция для вычисления среднего арифметического для набора чисел

```
package com.hadoopbook.hive;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.hive.serde2.io.DoubleWritable;

public class Mean extends UDAF {
```

продолжение ↗

Листинг 12.5 (продолжение)

```
public static class MeanDoubleUDAFEvaluator implements UDAFEvaluator {  
    public static class PartialResult {  
        double sum;  
        long count;  
    }  
  
    private PartialResult partial;  
    public void init() {  
        partial = null;  
    }  
  
    public boolean iterate(DoubleWritable value) {  
        if (value == null) {  
            return true;  
        }  
        if (partial == null) {  
            partial = new PartialResult();  
        }  
        partial.sum += value.get();  
        partial.count++;  
        return true;  
    }  
  
    public PartialResult terminatePartial() {  
        return partial;  
    }  
  
    public boolean merge(PartialResult other) {  
        if (other == null) {  
            return true;  
        }  
        if (partial == null) {  
            partial = new PartialResult();  
        }  
        partial.sum += other.sum;  
        partial.count += other.count;  
        return true;  
    }  
  
    public DoubleWritable terminate() {  
        if (partial == null) {  
            return null;  
        }  
        return new DoubleWritable(partial.sum / partial.count);  
    }  
}
```

13 HBase

Джонатан Грей и Майкл Стек

Знакомство с HBase

HBase — распределенная столбцово-ориентированная база данных, построенная на основе HDFS. Это приложение Hadoop, используемое в ситуациях, когда вам необходимо организовать произвольный доступ для чтения/записи к очень большим наборам данных в реальном времени.

Существуют бесчисленные стратегии и реализации хранения и выборки из баз данных, однако большинство решений — особенно из реляционной категории — строится без учета специфики очень больших масштабов и распределенного доступа. Многие фирмы предлагают инструменты репликации и разбиения, позволяющие базам данных выйти за пределы одного узла, но обычно такие дополнения строятся задним числом, сложны в установке и сопровождении. Кроме того, они также неполноценны в отношении функциональности РСУБД. Соединения, сложные запросы, триггеры, представления, ограничения внешнего ключа либо требуют недопустимо больших затрат ресурсов в крупномасштабной РСУБД, либо вовсе не работают.

HBase пытается решить проблему масштабирования по-другому. Эта система изначально строилась с расчетом на линейное масштабирование простым добавлением узлов. HBase не является реляционной базой данных и не поддерживает SQL, но в подходящем пространстве задачи она может сделать то, на что не способны традиционные РСУБД: организовать работу с очень большими разреженными таблицами в кластерах, состоящих из оборудования среднего уровня.

Каноническая область применения HBase — *веб-таблица* с информацией о веб-страницах, собранных в процессе обхода, и их атрибутах (например, язык и тип

MIME). Ключами веб-таблицы являются URL-адреса веб-страниц. Веб-таблица очень велика, счет строк в ней идет на миллиарды. Задания MapReduce постоянно работают с веб-таблицей, вычисляя статистику и добавляя новые столбцы с проверенным контентом для последующего индексирования поисковой системой. К таблице одновременно обращается множество ботов, работающих с разной скоростью и обновляющих случайные строки, а данные случайных веб-страниц поставляются в реальном времени, когда пользователи запрашивают данные кэшированных страниц.

В этой главе содержится вводная информация по использованию HBase. Более подробную информацию можно найти в книге Ларса Джорджа (Lars George) «HBase: The Definitive Guide» (O'Reilly, 2011).

История

Проект HBase запустили Чед Уолтерс (Chad Walters) и Джим Келлерман (Jim Kellerman) из Powerset. Прототип был описан в только что опубликованной статье Google «Bigtable: A Distributed Storage System for Structured Data» (Chang et al., (<http://labs.google.com/papers/bigtable.html>)). В феврале 2007 года Майк Кафарела (Mike Cafarella) написал заготовку кода системы, дальнейшей разработкой которой занимался Джим Келлерман (Jim Kellerman).

Первая версия HBase была включена в поставку Hadoop 0.15.0 в октябре 2007 года. В мае 2010 года система HBase перешла из категории подпроектов Hadoop в категорию проектов верхнего уровня Apache. К числу постоянных пользователей HBase относятся Adobe, StumbleUpon, Twitter и группы Yahoo!.

Концепции

В этом разделе приводится краткий обзор основных концепций HBase. Даже краткое знакомство с ними упростит восприятие дальнейшего материала¹.

Краткий обзор модели данных

Приложения хранят данные в таблицах, состоящих из строк и столбцов. Для ячеек таблицы (пересечения строк и столбцов) существует контроль версии. По умолчанию в качестве версии используется временная метка, автоматически назначаемая HBase на момент вставки. Содержимое ячейки представляет собой неинтерпретируемый массив байтов.

¹ За более подробной информацией обращайтесь к странице «HBase Architecture» в вики HBase.

Ключи строк таблицы тоже являются байтовыми массивами, поэтому теоретически ключом строки может быть что угодно — от строк до двоичных представлений `long` и даже сериализованных структур данных. Строки таблицы сортируются по ключу строк (первичному ключу таблицы). Сортировка осуществляется в порядке следования байтов. Все обращения к таблице выполняются по первичному ключу¹.

Столбцы объединяются в *семейства столбцов*. Все члены семейства столбцов имеют общий префикс, так что, например, столбцы `temperature:air` и `temperature:dew_point` принадлежат семейству `temperature`, а `station:identifier` принадлежит семейству `station`. Префикс семейства столбцов должен состоять из печатаемых символов. Завершающая часть (*квалификатор*) может состоять из произвольных байтов.

Семейства столбцов таблицы должны быть заданы заранее как часть определения схемы таблицы, однако новые члены семейств могут добавляться по мере необходимости. Например, новый столбец `station:address` может быть передан клиенту как часть обновления, и его значение будет успешно сохраняться — при условии, что семейство столбцов `station` уже существует в таблице.

Физически все члены семейств столбцов хранятся вместе в файловой системе. Таким образом, хотя ранее мы описывали HBase как столбцово-ориентированное хранилище информации, точнее было бы сказать «ориентированное на семейства столбцов». Так как настройки и спецификации задаются на уровне семейств столбцов, желательно, чтобы все члены семейств имели сходные схемы доступа и характеристики размеров.

Итак, таблицы HBase похожи на таблицы РСУБД, только ячейки имеют версии, строки сортируются, а столбцы могут добавляться клиентом «на ходу» при условии, что семейство столбцов, к которому они принадлежат, уже существует.

Регионы

HBase автоматически производит горизонтальную разбивку таблиц на регионы. Каждый регион образует подмножество строк таблицы. Регион определяется таблицей, которой он принадлежит, своей первой строкой (включительно) и последней строкой (без включения). Изначально таблица состоит из одного региона, но с ростом размера региона после превышения настраиваемого порогового размера он разбивается на два новых региона приблизительно равных размеров. До первого разбиения вся загрузка данных будет осуществляться на одном сервере, на котором размещен исходный регион. По мере роста таблицы увеличивается количество ее регионов. Регионы являются единицами, распределяемыми в кластере HBase.

¹ На момент написания книги на github было не менее двух проектов по поддержке вторичных индексов в HBase.

Если таблица оказывается слишком большой для одного отдельного сервера, она может обслуживаться кластером серверов, на каждом узле которого размещается подмножество регионов таблицы. Кроме того, регионы обеспечивают распределение нагрузки на таблицу. Совокупность отсортированных регионов, доступных по сети, образует общее содержимое таблицы.

Блокировка

Обновления строк выполняются атомарно независимо от того, сколько столбцов задействовано в транзакции уровня строки. Это упрощает модель блокировки.

Реализация

По аналогии с тем, как системы HDFS и MapReduce состоят из клиентов, подчиненных узлов и координаторов (узлов имен и узлов данных в HDFS, трекеров задач и трекеров задач в MapReduce), модель HBase состоит из *главного узла* HBase, управляющего кластером из одного или нескольких подчиненных *серверов регионов* (рис. 13.1). Главный узел HBase отвечает за активизацию «чистой» установки, за назначение регионов зарегистрированным серверам регионов, а также за восстановление серверов регионов после сбоев. Нагрузка на главный узел относительно невелика. Серверы регионов содержат нуль и более регионов и обслуживают клиентские запросы чтения/записи. Кроме того, они управляют разбиением регионов, сообщая главному узлу HBase о появлении новых дочерних регионов, чтобы главный узел организовал отключение родительских регионов и назначение дочерних регионов-заменителей.

HBase использует ZooKeeper (глава 14), и по умолчанию управляет экземпляром ZooKeeper как авторитетный источник информации о состоянии кластера. HBase хранит такую важнейшую информацию, как местонахождение корневой таблицы каталогов и адрес текущего главного узла кластера. Назначение регионов осуществляется при посредничестве ZooKeeper на случай, если на серверах-участниках в процессе назначения произойдет сбой. Хранение состояния транзакции в ZooKeeper позволяет продолжить процесс восстановления с назначения, на котором сбойный сервер вышел из строя. Как минимум при активизации клиентского подключения к кластеру HBase клиенту должно быть передано местонахождение ансамбля ZooKeeper. Далее клиент обращается к иерархии ZooKeeper за такими атрибутами кластера, как местонахождение серверов¹.

Подчиненные узлы серверов регионов перечислены в файле HBase *conf/regionservers* по аналогии с тем, как узлы данных и трекеры задач перечисляются

¹ HBase также можно настроить на использование существующего кластера ZooKeeper.

в файле Hadoop *conf/slaves*. Сценарии запуска и остановки аналогичны тем, что используются в Hadoop, и используют тот же механизм на базе SSH для выполнения удаленных команд. Конфигурационные данные кластера хранятся в файлах HBase *conf/hbase-site.xml* и *conf/hbase-env.sh*, имеющих такой же формат, как и их эквиваленты в родительском проекте Hadoop (см. главу 9).

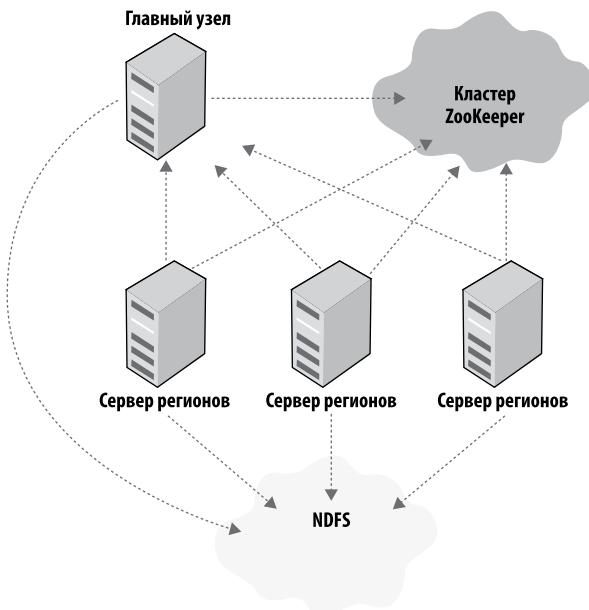


Рис. 13.1. Кластер HBase



Там, где присутствует сходство, HBase напрямую использует или субклассирует родительскую реализацию Hadoop, будь то сервис или тип. Если прямое использование неприемлемо, HBase старается следовать модели Hadoop, насколько это возможно. Например, HBase использует систему Hadoop Configuration, так что конфигурационные файлы имеют тот же формат. Это означает, что при изучении HBase вы можете использовать любое сходство с Hadoop. HBase отходит от этого правила только при добавлении собственных специализаций.

HBase сохраняет данные через API файловой системы Hadoop. Существует несколько реализаций интерфейса файловой системы — для локальной файловой системы, для файловой системы KFS, Amazon S3 и HDFS. Большинство пользователей HBase хранит данные в HDFS, хотя по умолчанию и без явного указания

HBase записывает данные в локальную файловую систему. Локальная файловая система хорошо подходит для первых экспериментов с HBase, но в дальнейшем изменение настройки в кластере HBase обычно начинается с выбора кластера HDFS, который должен использоваться для работы.

HBase в действии

Во внутренней реализации HBase существуют специальные таблицы каталогов с именами **-ROOT-** и **.META.**, в которых хранится текущий список, данные состояния и местонахождение всех действующих регионов в кластере. В таблице **-ROOT-** содержится список регионов таблицы **.META..** В таблице **.META.** содержится список всех регионов пользовательского пространства. Ключом записей в этих таблицах является имя региона, которое состоит из имени таблицы, начальной строки региона, времени создания и, наконец, контрольной суммы MD5 всех предшествующих данных¹. Как упоминалось ранее, ключи строк отсортированы, так что поиск региона, к которому относится конкретная строка, сводится к поиску первой записи, ключ которой больше запрашиваемого ключа либо равен ему. При изменения состояния регионов — разбиении, отключении/включении, удалении, перемещении распределителем нагрузки или перемещении из-за сбоя сервера регионов — таблицы каталогов обновляются так, чтобы в них хранилось актуальное состояние всех регионов кластера.

Клиенты сначала подключаются к кластеру ZooKeeper, чтобы узнать местонахождение **-ROOT-**. Клиент обращается к **-ROOT-**, чтобы узнать местонахождение региона **.META..**, в который входит запрашиваемая строка. Затем клиент по данным найденного региона **.META..** находит регион пользовательского пространства и его местонахождение. В дальнейшем клиент напрямую взаимодействует с сервером региона, обслуживающим нужный регион.

Чтобы избежать трех лишних обращений при каждой операции со строкой, клиент кэширует всю информацию, полученную в процессе обращений к **-ROOT-** и **.META..**, а также начальную и конечную строки регионов пользовательского пространства, чтобы клиент мог сам определять нужные регионы без обращения к таблице **.META..**. Клиенты продолжают использовать кэшированные данные вплоть до сбоя при очередном обращении. Когда это произойдет (то есть когда регион переместился), клиент снова обращается к **.META..** за информацией о новом местонахождении. Если регион **.META..** переместился, происходит повторное обращение к **-ROOT-**.

¹ Пример имени региона из таблицы TestTable с начальной строкой xyz: TestTable,xyz,1279729913622.1b6e176fb8d8aa88fd4ab6bc80247ece. Имя таблицы, начальная строка и временная метка разделяются запятыми. Имя всегда завершается точкой.

Операции записи, поступающие на сервер регионов, сначала регистрируются в журнале изменений, а затем добавляются в *хранилище в памяти* (memstore). Когда хранилище в памяти заполняется, его содержимое закрепляется в файловой системе.

Журнал изменений размещается в HDFS и остается доступным в случае сбоя сервера регионов. Когда главный узел замечает, что сервер регионов стал недоступным (обычно из-за истечения срока действия z-узла сервера в ZooKeeper), он разбивает журнал изменений недоступного сервера по регионам. При переназначении и до начала функционирования регионы, находившиеся на недоступном сервере регионов, получают свою часть файла несохраненных изменений и воспроизводят их, чтобы актуализировать свое состояние на момент, непосредственно предшествовавший сбою.

Чтение начинается с обращения к хранилищу в памяти региона. Если нужные версии обнаруживаются при чтении хранилища в памяти, обработка запроса на этом завершается. В противном случае сначала проверяются файлы выгрузки (flush files) от самого нового к самому старому, пока не будут найдены версии, достаточные для удовлетворения запроса, или пока не будут перебраны все файлы.

Фоновый процесс следит за количеством файлов выгрузки и, когда оно превышает заданный порог, уплотняет их, переписывая несколько файлов в один (чем меньше файлов просматривается в процессе чтения, тем эффективнее выполняется операция).

Установка

Загрузите стабильную версию с сайта загрузки Apache и распакуйте ее в своей локальной файловой системе. Например:

```
% tar xzf hbase-x.y.z.tar.gz
```

Как и в случае с Hadoop, сначала необходимо сообщить HBase, где в вашей системе находится установка Java. Если переменной окружения `JAVA_HOME` присвоено значение, указывающее на подходящую установку Java, оно будет использовано, и настраивать дополнительного ничего не придется. В противном случае установка Java, используемая HBase, задается редактированием файла HBase `conf/hbase-env.sh` и настройкой переменной `JAVA_HOME`.



Для HBase, как и для Hadoop, необходима установка Java 6.

Для удобства включите двоичный каталог HBase в путь командной строки, например:

```
% export HBASE_HOME=/home/hbase/hbase-x.y.z
% export PATH=$PATH:$HBASE_HOME/bin
```

Чтобы получить справку по использованию HBase, введите команду

```
% hbase
Usage: hbase <command>
where <command> is one of:
  shell          run the HBase shell
  master         run an HBase HMaster node
  regionserver   run an HBase HRegionServer node
  zookeeper      run a Zookeeper server
  rest           run an HBase REST server
  thrift          run an HBase Thrift server
  avro           run an HBase Avro server
  migrate        upgrade an hbase.rootdir
  hbck           run the hbase 'fsck' tool
or
  CLASSNAME      run the class named CLASSNAME
Most commands print help when invoked w/o parameters.
```

Пробный запуск

Чтобы запустить временный экземпляр HBase, использующий каталог */tmp* локальной файловой системы для хранения данных, введите следующую команду:

```
% start-hbase.sh
```

Команда запускает автономный экземпляр HBase, который сохраняет данные в локальной файловой системе; по умолчанию HBase записывает в */tmp/hbase-\$USERID*.

Для выполнения административных операций с вашим экземпляром HBase, запустите оболочку HBase командой

```
% hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version: 0.89.0-SNAPSHOT, ra4ea1a9a7b074a2e5b7b24f761302d4ea28ed1b2, Sun Jul 18
15:01:50 PDT 2010 hbase(main):001:0>
```

Команда запускает интерпретатор JRuby с некоторыми специфическими командами HBase. Чтобы просмотреть список команд, сгруппированных по категориям,

введите команду `help` и нажмите RETURN. Команда `help группа_команд` выводит справку по категории, а команда `help команда` – справку по конкретной команде с примерами использования.

Создадим простую таблицу, добавим в нее данные, а затем удалим их.

Чтобы создать таблицу, вы должны указать ее имя и определить схему. Схема таблицы состоит из атрибутов и списка семейств столбцов. Семейства столбцов сами обладают атрибутами, которые тоже должны задаваться во время определения схемы. Например, к атрибутам семейств столбцов относятся признак сжатия содержимого семейства в файловой системе и количество хранимых версий ячейки. Схему можно отредактировать позднее; для этого следует отключить таблицу командой `disable`, внести необходимые изменения командой `alter`, а затем вернуть таблицу в режим использования командой `enable`.

Следующая команда создает таблицу с именем `test` и одним семейством столбцов `data`, со значениями по умолчанию для атрибутов таблицы и семейств столбцов:

```
hbase(main):007:0> create 'test', 'data'  
0 row(s) in 1.3066 seconds
```



Если при выполнении команды оболочки выводит сообщение об ошибке и трассировку стека – значит, при установке была допущена ошибка. Информацию о возникших проблемах стоит поискать в журналах главного узла – по умолчанию они хранятся в каталоге `${HBASE_HOME}/logs`.

Примеры с добавлением атрибутов таблиц и семейств столбцов при указании схемы приведены в выходных данных команды `help`.

Чтобы убедиться в том, что новая таблица была создана успешно, выполните команду `list`. Команда выводит все таблицы в пользовательском пространстве:

```
hbase(main):019:0> list  
test  
1 row(s) in 0.1485 seconds
```

Следующие команды вставляют данные в три разных строки и столбца семейства `data`, после чего выводят содержимое таблицы:

```
hbase(main):021:0> put 'test', 'row1', 'data:1', 'value1'  
0 row(s) in 0.0454 seconds  
hbase(main):022:0> put 'test', 'row2', 'data:2', 'value2'  
0 row(s) in 0.0035 seconds  
hbase(main):023:0> put 'test', 'row3', 'data:3', 'value3'  
0 row(s) in 0.0090 seconds
```

продолжение ↗

```
hbase(main):024:0> scan 'test'
ROW                                COLUMN+CELL
  row1                           column=data:1, timestamp=1240148026198,
  value=value1
  row2                           column=data:2, timestamp=1240148040035,
  value=value2
  row3                           column=data:3, timestamp=1240148047497,
  value=value3
3 row(s) in 0.0825 seconds
```

Обратите внимание: три новых столбца добавлены без изменения схемы.

Чтобы удалить таблицу, необходимо сначала вывести ее из использования:

```
hbase(main):025:0> disable 'test'
09/04/19 06:40:13 INFO client.HBaseAdmin: Disabled test
0 row(s) in 6.0426 seconds
hbase(main):026:0> drop 'test'
09/04/19 06:40:17 INFO client.HBaseAdmin: Deleted test
0 row(s) in 0.0210 seconds
hbase(main):027:0> list
0 row(s) in 2.0645 seconds
```

Завершите выполнение своего экземпляра HBase командой

```
% stop-hbase.sh
```

За информацией о том, как настроить HBase в распределенном режиме и связать систему с действующим экземпляром HDFS, обращайтесь к разделу «Getting Started» документации HBase.

Клиенты

Существует несколько вариантов организации клиентских взаимодействий с кластером HBase.

Java

Система HBase, как и Hadoop, написана на Java. В листинге 13.1 представлена реализация операций, описанных в разделе «Пробный запуск», на языке Java.

Листинг 13.1. Основные административные операции и обращение к таблице

```
public class ExampleClient {
    public static void main(String[] args) throws IOException {
        Configuration config = HBaseConfiguration.create();

        // Создание таблицы
        HBaseAdmin admin = new HBaseAdmin(config);
        HTableDescriptor htd = new HTableDescriptor("test");
        HColumnDescriptor hcd = new HColumnDescriptor("data");
        htd.addFamily(hcd);
        admin.createTable(htd);
        byte [] tablename = htd.getName();
        HTableDescriptor [] tables = admin.listTables();
        if (tables.length != 1 && Bytes.equals(tablename, tables[0].getName())) {
            throw new IOException("Failed create of table");
        }

        // Выполнение операций (запись, чтение, сканирование) с таблицей
        HTable table = new HTable(config, tablename);
        byte [] row1 = Bytes.toBytes("row1");
        Put p1 = new Put(row1);
        byte [] databytes = Bytes.toBytes("data");
        p1.add(databytes, Bytes.toBytes("1"), Bytes.toBytes("value1"));
        table.put(p1);
        Get g = new Get(row1);
        Result result = table.get(g);
        System.out.println("Get: " + result);
        Scan scan = new Scan();
        ResultScanner scanner = table.getScanner(scan);
        try {
            for (Result scannerResult: scanner) {
                System.out.println("Scan: " + scannerResult);
            }
        } finally {
            scanner.close();
        }

        // Удаление таблицы
        admin.disableTable(tablename);
        admin.deleteTable(tablename);
    }
}
```

Класс содержит только метод `main`. Для краткости мы не указываем ни имя пакета, ни директивы импортирования. Работа класса начинается с создания экземпляра `org.apache.hadoop.conf.Configuration`. Мы обращаемся к классу `org.apache.hadoop.hbase.HBaseConfiguration` с запросом на создание экземпляра. Он возвращает объект `Configuration` с данными конфигурации HBase, прочитанными из файлов `hbase-site.xml` и `hbase-default.xml` из пути к классам. В дальнейшем полученный экземпляр `Configuration` используется для создания экземпляров `HBaseAdmin` и `HTable` — двух классов из пакета `org.apache.hadoop.hbase.client`. Класс `HBaseAdmin` используется для администрирования кластера HBase, а конкретно — для добавления и удаления таблиц. Класс `HTable` используется для обращения к конкретной таблице. Экземпляр `Configuration` сообщает этим классам информацию о кластере, с которым должен работать код.

Чтобы создать таблицу, мы сначала создаем экземпляр `HBaseAdmin`, а затем приказываем ему создать таблицу с именем `test` и одним семейством столбцов с именем `data`. В нашем примере используется схема таблицы по умолчанию. Для изменения схемы таблицы используются методы `org.apache.hadoop.hbase.HTableDescriptor` и `org.apache.hadoop.hbase.HColumnDescriptor`. Затем код проверяет, что таблица действительно была создана, и переходит к выполнению операций с только что созданной таблицей.

Для выполнения операций с таблицей необходимо создать экземпляр `org.apache.hadoop.hbase.client.HTable` с передачей экземпляра `Configuration` и имени таблицы, с которой будут выполняться операции. После создания `HTable` мы создаем экземпляр `org.apache.hadoop.hbase.client`. Вызов `Put` помещает значение `value1` в строку `row1` столбца `data:1`. Имя столбца состоит из двух частей: имя семейства столбцов в виде байтов (`getBytes` в приведенном коде), а затем квалификатор семейства столбцов, заданный в виде `Bytes.toBytes("1")`. Далее мы создаем экземпляр `org.apache.hadoop.hbase.client.Get`, читаем значение только что записанной ячейки, а затем используем `org.apache.hadoop.hbase.client.Scan` для сканирования только что созданной таблицы с выводом найденных данных.

Программа завершается отключением таблицы и ее последующим удалением. Прежде чем удалять таблицу, необходимо сначала отключить ее, то есть вывести из использования.

MapReduce

Классы HBase и пакета `org.apache.hadoop.hbase.mapreduce` упрощают использование HBase в качестве источника и(или) приемника данных в заданиях MapReduce. Класс `TableInputFormat` записывает результат выполненной свертки в HBase. Класс `RowCounter` (см. листинг 13.2) из пакета HBase `mapreduce` выполняет задачу отображения для подсчета строк таблицы с использованием `TableInputFormat`.

Листинг 13.2. Приложение MapReduce для подсчета строк данных в таблице HBase

```
public class RowCounter {  
    /** Имя "программы". */  
    static final String NAME = "rowcounter";  
  
    static class RowCounterMapper  
        extends TableMapper<ImmutableBytesWritable, Result> {  
        /** Счетчик строк. */  
        public static enum Counters {ROWS}  
  
        @Override  
        public void map(ImmutableBytesWritable row, Result values,  
                        Context context)  
            throws IOException {  
            for (KeyValue value: values.list()) {  
                if (value.getValue().length > 0) {  
                    context.getCounter(Counters.ROWS).increment(1);  
                    break;  
                }  
            }  
        }  
    }  
  
    public static Job createSubmittableJob(Configuration conf, String[] args)  
        throws IOException {  
        String tableName = args[0];  
        Job job = new Job(conf, NAME + "_" + tableName);  
        job.setJarByClass(RowCounter.class);  
        // Столбцы разделяются пробелами  
        StringBuilder sb = new StringBuilder();  
        final int columnoffset = 1;  
        for (int i = columnoffset; i < args.length; i++) {  
            if (i > columnoffset) {  
                sb.append(" ");  
            }  
            sb.append(args[i]);  
        }  
        Scan scan = new Scan();  
        scan.setFilter(new FirstKeyOnlyFilter());  
        if (sb.length() > 0) {  
            for (String columnName : sb.toString().split(" ")) {  
                String [] fields = columnName.split(":");  
                if(fields.length == 1) {  
                    scan.addFamily(Bytes.toBytes(fields[0]));  
                }  
            }  
        }  
    }  
}
```

продолжение ↗

Листинг 13.2 (продолжение)

```

        } else {
            scan.addColumn(Bytes.toBytes(fields[0]), Bytes.toBytes(fields[1]));
        }
    }
// Второй аргумент - имя таблицы.
job.setOutputFormatClass(NullOutputFormat.class);
TableMapReduceUtil.initTableMapperJob(tableName, scan,
    RowCounterMapper.class, ImmutableBytesWritable.class, Result.class, job);
job.setNumReduceTasks(0);
return job;
}

public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();
    String[] otherArgs = new
        GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 1) {
        System.err.println("ERROR: Wrong number of parameters: " + args.length);
        System.err.println("Usage: RowCounter <tablename>
                           [<column1> <column2>...]");
        System.exit(-1);
    }
    Job job = createSubmittableJob(conf, otherArgs);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

Для разбора аргументов командной строки используется класс `GenericOptionsParser`, рассмотренный в разделе «`GenericOptionsParser`, `Tool` и `ToolRunner`». Внутренний класс `RowCounterMapper` реализует абстракцию HBase `TableMapper` — специализацию `org.apache.hadoop.mapreduce.Mapper`, которая задает входные типы, передаваемые `TableInputFormat`. Метод `createSubmittableJob()` разбирает аргументы, добавленные в конфигурацию из командной строки, и задает таблицу и столбцы, с которыми будет работать `RowCounter`. Имена столбцов используются для настройки экземпляра `org.apache.hadoop.hbase.client.Scan` — объекта сканирования, который будет передаваться `TableInputFormat` и использоваться для ограничения данных, видимых для `Mapper`. Обратите внимание на определение фильтра (экземпляр `org.apache.hadoop.hbase.filter.FirstKeyOnlyFilter`) для объекта сканирования. Фильтр приказывает серверу использовать ускоренную обработку при выполнении на стороне сервера, ограничиваясь простой проверкой наличия данных в строке перед возвращением; он ускоряет подсчет строк. Метод `createSubmittableJob()` также вызывает вспомогательный метод

`TableMapReduceUtil.initTableMapJob()`, который помимо прочего назначает входной формат `TableInputFormat`. Реализация `map` проста — она ограничивается проверкой пустых значений. Если значение пусто, строка не учитывается в подсчете, а если нет — значение `Counters.ROWS` увеличивается на 1.

Avro, REST и Thrift

HBase поставляется с интерфейсами Avro, REST и Thrift. Эти интерфейсы полезны при взаимодействии с приложениями, написанными на языке, отличном от Java. Во всех случаях на сервере Java работает экземпляр клиента HBase, который обслуживает запросы приложений Avro, REST и Thrift из кластера HBase. Дополнительная работа по обработке запросов и ответов означает, что эти интерфейсы работают медленнее, чем прямое использование клиента Java.

REST

Экземпляр `stargate` (имя REST-сервиса HBase) запускается следующей командой:

```
% hbase-daemon.sh start rest
```

Команда запускает экземпляр сервера (по умолчанию на порте 8080) и переводит его в фоновый режим. Операции сервера регистрируются в журналах, находящихся в каталоге HBase `logs`.

Клиент может запросить ответ в формате JSON, Google protobufs или XML (в зависимости от настройки заголовка HTTP `Accept`). За документацией и примерами клиентских запросов REST обращайтесь к вики REST.

Сервер REST останавливается командой

```
% hbase-daemon.sh stop rest
```

Thrift

Сервер Thrift, обслуживающий клиентов Thrift, запускается командой

```
% hbase-daemon.sh start thrift
```

Команда запускает экземпляр сервера (по умолчанию на порте 9090) и переводит его в фоновый режим. Операции сервера регистрируются в журналах, находящихся в каталоге HBase `logs`. В документации HBase Thrift¹ сказано, что версия Thrift использует генерирование классов. Код HBase Thrift IDL находится в каталоге `src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift` в исходном коде HBase.

¹ <http://hbase.apache.org/docs/current/api/org/apache/hadoop/hbase/thrift/package-summary.html>

Сервер Thrift останавливается командой

```
% hbase-daemon.sh stop thrift
```

Avro

Сервер Avro запускается и останавливается так же, как и серверы Thrift или REST. По умолчанию сервер Avro использует порт 9090 (тот же, что и сервер Thrift, но обычно оба сервера одновременно не запускаются).

Пример

HDFS и MapReduce — мощные инструменты для выполнения пакетных операций с большими наборами данных, но они не предоставляют средств для эффективного чтения или сохранения отдельных записей. В следующем примере мы посмотрим, как HBase помогает заполнить этот пробел.

Набор метеорологических данных, описанный в предыдущих главах, содержит результаты наблюдений по десяткам тысяч станций за 100 лет, и объем данных постоянно растет. Мы построим простой веб-интерфейс, при помощи которого пользователь сможет запрашивать данные разных станций и просматривать историю температурных наблюдений в хронологическом порядке. Будем считать, что объем данных очень велик, количество наблюдений исчисляется миллиардами, а скорость поступлений обновлений значительна — скажем, от сотен до тысяч обновлений в секунду для всего диапазона метеорологических станций. Также допустим, что веб-приложение должно отображать данные новейших наблюдений примерно через секунду после их получения.

Первое требование исключает применение простой РСУБД и делает HBase подходящим кандидатом. Второе требование исключает простое решение на базе HDFS. Задание MapReduce может построить исходные индексы, обеспечивающие произвольный доступ ко всем данным наблюдений, но с обновлением индекса по мере поступления обновлений HDFS и MapReduce уже не справятся.

Схемы

В нашем примере будут использоваться две таблицы:

stations

Таблица содержит данные метеостанций (ключ `stationid`). Семейство столбцов `info` содержит информацию о метеостанциях в формате «ключ-значение». Ключами словаря являются столбцы `info:name`, `info:location`

и `info:description`. Эта таблица является статической, а семейство `info` по своей структуре напоминает таблицу типичной РСУБД.

`observations`

Таблица содержит результаты наблюдений температур; составной ключ образуется из `stationid` и инвертированной временной метки. В таблице определяется семейство столбцов `data` из одного столбца `airtemp`, в котором хранится значение температуры.

При выборе схемы мы руководствовались самым эффективным способом чтения из HBase. Строки и столбцы хранятся в лексикографическом порядке по возрастанию. Средства вторичного индексирования и поиска по регулярным выражениям существуют, но за них приходится расплачиваться быстродействием. Очень важно понимать, какой способ обращения к данным будет самым эффективным, чтобы выбрать самую эффективную организацию их хранения.

Для таблицы `stations` выбор `stationid` в качестве ключа очевиден, потому что мы всегда обращаемся к информации конкретной станции по идентификатору. Однако таблица `observations` использует составной ключ с добавлением временной метки наблюдения. С таким выбором ключа все наблюдения конкретной станции группируются вместе, а при инверсии временной метки (`Long.MAX_VALUE - epoch`) и хранении ее в двоичном виде наблюдения по каждой станции будут упорядочены так, что самые свежие данные будут находиться на первом месте.

Таблицы создаются следующими командами:

```
hbase(main):036:0> create 'stations', {NAME => 'info', VERSIONS => 1}
0 row(s) in 0.1304 seconds
hbase(main):037:0> create 'observations', {NAME => 'data', VERSIONS => 1}
0 row(s) in 0.1332 seconds
```

В обоих случаях нас интересует только новейшая версия ячейки, поэтому атрибуту `VERSIONS` присваивается значение 1 (по умолчанию 3).

Загрузка данных

Количество станций относительно невелико, поэтому их статические данные легко вставляются в любом из доступных интерфейсов.

Однако будем считать, что количество данных наблюдений исчисляется миллиардами. Обычно импортование такого рода является исключительно сложной и продолжительной операцией, но распределенная модель MapReduce и HBase позволяет нам в полной мере использовать преимущества кластера. Скопируйте входные данные в HDFS, а затем запустите задание MapReduce, которое прочитает входные данные и запишет их в HBase.

В листинге 13.3 приведен пример задания MapReduce, импортирующего данные наблюдений в HBase из входного файла, который использовался в примерах предыдущих глав.

Листинг 13.3. Приложение MapReduce для импортирования температурных данных из HDFS в таблицу HBase

```
public class HBaseTemperatureImporter extends Configured implements Tool {

    // Внутренний класс для map
    static class HBaseTemperatureMapper<K, V> extends MapReduceBase implements
        Mapper<LongWritable, Text, K, V> {
        private NcdcRecordParser parser = new NcdcRecordParser();
        private HTable table;

        public void map(LongWritable key, Text value,
                       OutputCollector<K, V> output, Reporter reporter)
            throws IOException {
            parser.parse(value.toString());
            if (parser.isValidTemperature()) {
                byte[] rowKey =
                    RowKeyConverter.makeObservationRowKey(parser.getStationId(),
                        parser.getObservationDate().getTime());
                Put p = new Put(rowKey);
                p.add(HBaseTemperatureCli.DATA_COLUMNFAMILY,
                      HBaseTemperatureCli.AIRTEMP_QUALIFIER,
                      Bytes.toBytes(parser.getAirTemperature()));
                table.put(p);
            }
        }

        public void configure(JobConf jc) {
            super.configure(jc);
            // Клиент таблицы HBase создается заранее и сохраняется
            // (вместо создания при каждом вызове map).
            try {
                this.table = new HTable(new HBaseConfiguration(jc), "observations");
            } catch (IOException e) {
                throw new RuntimeException("Failed HTable construction", e);
            }
        }

        @Override
        public void close() throws IOException {
            super.close();
            table.close();
        }
    }
}
```

```

public int run(String[] args) throws IOException {
    if (args.length != 1) {
        System.err.println("Usage: HBaseTemperatureImporter <input>");
        return -1;
    }
    JobConf jc = new JobConf(getConf(), getClass());
    FileInputFormat.addInputPath(jc, new Path(args[0]));
    jc.setMapperClass(HBaseTemperatureMapper.class);
    jc.setNumReduceTasks(0);
    jc.setOutputFormat(NullOutputFormat.class);
    JobClient.runJob(jc);
    return 0;
}

public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new HBaseConfiguration(),
        new HBaseTemperatureImporter(), args);
    System.exit(exitCode);
}
}

```

`HBaseTemperatureImporter` содержит внутренний класс `HBaseTemperatureMapper`, который является аналогом класса `MaxTemperatureMapper` из главы 5. Внешний класс реализует `Tool` и создает условия для запуска внутреннего класса `HBaseTemperatureMapper`. `HBaseTemperatureMapper` получает те же данные, что и `MaxTemperatureMapper`, и выполняет тот же разбор данных (с использованием класса `NcdcRecordParser`, представленного в главе 5) для проверки температур. Однако вместо добавления действительных температур в приемник вывода, как это делал класс `MaxTemperatureMapper`, он добавляет действительные температуры в столбец `data:airtemp` таблицы HBase `observations`. (Мы используем для `data` и `airtemp` статические константы, импортированные из класса `HBaseTemperatureCli` — см. далее.) В методе `configure()` однократно создается экземпляр `HTable`, который в дальнейшем используется во взаимодействии с HBase. В завершение мы вызываем `close()` для экземпляра `HTable`, чтобы выгрузить все буферы записи, которые не были очищены ранее.

Используемый ключ создается в методе `makeObservationRowKey()` класса `RowKeyConverter` из идентификатора станции и времени наблюдения:

```

public class RowKeyConverter {

    private static final int STATION_ID_LENGTH = 12;

    /**
     * @return Ключ строки в формате: <station_id> <reverse_order_epoch>
     */

```

продолжение ↗

```

public static byte[] makeObservationRowKey(String stationId,
    long observationTime) {
    byte[] row = new byte[STATION_ID_LENGTH + Bytes.SIZEOF_LONG];
    Bytes.putBytes(row, 0, Bytes.toBytes(stationId), 0, STATION_ID_LENGTH);
    long reverseOrderEpoch = Long.MAX_VALUE - observationTime;
    Bytes.putLong(row, STATION_ID_LENGTH, reverseOrderEpoch);
    return row;
}

}

```

В преобразовании используется тот факт, что идентификатор станции представляет собой строку фиксированной длины. Класс `Bytes`, используемый в `makeObservationRowKey()`, взят из вспомогательного пакета HBase, и включает в себя методы для преобразования между байтовыми массивами и стандартными типами Java и Hadoop. В методе `makeObservationRowKey()` метод `Bytes.putLong()` заполняет массив байтов ключа, а константа `Bytes.SIZEOF_LONG` используется для определения размера.

Программа запускается следующей командой:

```
% hbase HBaseTemperatureImporter input/ncdc/all
```

Замечания по оптимизации

- Заранее продумайте ситуацию точечной нагрузки на таблицу при импортировании, когда все клиенты дружно работают с одним из регионов таблицы (а следовательно, с одним узлом), затем переходят к следующему и т. д. — вместо равномерного распределения нагрузки по всем регионам. Обычно такие ситуации возникают из-за сочетания отсортированных входных данных с особенностями работы механизма разбиения. Возможно, проблема решится рандомизацией порядка ключей строк перед вставкой. В нашем примере с учетом распределения значений `stationid` и способом разбиения `TextInputFormat` нагрузка должна быть распределена достаточно равномерно¹.
- Получайте только один экземпляр `HTable` на задачу. Создание экземпляра `HTable` требует определенных затрат; выполнение этой операции при каждой вставке может отрицательно повлиять на производительность, поэтому мы и создаем экземпляр `HTable` на шаге `configure()`.

¹ Новая таблица содержит всего один регион, и изначально все обновления будут относиться к этому одному региону до тех пор, пока не произойдет разбиение. Это произойдет даже в том случае, если ключи имеют случайное распределение. Данная ситуация означает, что передача данных поначалу будет происходить медленно, пока количество регионов не станет достаточным для того, чтобы в передаче могли участвовать все узлы кластера. Не путайте это явление с тем, которое описано в тексте.

- По умолчанию каждый вызов `HTable.put(put)` выполняет фактическую вставку без буферизации. Чтобы включить режим буферизации `HTable`, вызовите `HTable.setAutoFlush(false)` и задайте размер настраиваемого буфера записи. Когда буфер записи заполняется несохраненными вставками, происходит выгрузка. Однако следует помнить, что в конце выполнения каждой задачи необходимо вызвать `HTable.flushCommits()` (или `HTable.close()`, что приведет к вызову `HTable.flushCommits()`). Это можно сделать в переопределении метода `close()` функции отображения.
- В HBase имеются классы `TableInputFormat` и `TableOutputFormat`, которые упрощают программирование заданий MapReduce, использующих HBase в качестве источника и приемника данных (см. листинг 13.2). В одной из возможных реализаций предыдущего примера можно было бы использовать `MaxTemperatureMapper` из главы 5 в неизменном виде, но добавить задачу свертки, которая получает вывод `MaxTemperatureMapper` и передает его HBase через `TableOutputFormat`.

Веб-запросы

В реализации веб-приложения мы будем напрямую использовать HBase Java API. При этом становится особенно очевидно, насколько важен выбор схемы и формата хранения данных.

Простейший запрос получает статическую информацию о метеостанциях. Запросы такого рода легко реализуются в традиционных базах данных, но HBase предоставляет дополнительную гибкость и возможности управления. При использовании семейства `info` как словаря (ключи — имена столбцов, значения — данные столбцов) код будет выглядеть примерно так:

```
public Map<String, String> getStationInfo(HTable table, String stationId)
    throws IOException {
    Get get = new Get(Bytes.toBytes(stationId));
    get.addColumn(INFO_COLUMNFAMILY);
    Result res = table.get(get);
    if (res == null) {
        return null;
    }
    Map<String, String> resultMap = new HashMap<String, String>();
    resultMap.put("name", getValue(res, INFO_COLUMNFAMILY, NAME_QUALIFIER));
    resultMap.put("location", getValue(res, INFO_COLUMNFAMILY,
        LOCATION_QUALIFIER));
    resultMap.put("description", getValue(res, INFO_COLUMNFAMILY,
        DESCRIPTION_QUALIFIER));
```

продолжение ↗

```

        return resultMap;
    }

private static String getValue(Result res, byte [] cf, byte [] qualifier) {
    byte [] value = res.getValue(cf, qualifier);
    return value == null? ":" : Bytes.toString(value);
}

```

В этом примере `getStationInfo()` получает экземпляр `HTable` и идентификатор метеостанции. Чтобы получить информацию о станции, мы передаем `HTable.get()` экземпляр `Get`, настроенный для получения значений всех столбцов строки, заданной идентификатором станции в семействе столбцов `INFO_COLUMNFAMILY`.

Результаты `get()` возвращаются в виде объекта `Result`. Он содержит строку данных; чтобы получить значения столбцов, следует указать нужную ячейку. Метод `getStationInfo()` преобразует данные `Result` в более удобный контейнер `Map` с ключами и значениями `String`.

Как видите, вспомогательные функции при использовании HBase действительно необходимы. На базе HBase строится все больше абстракций, упрощающих подобные низкоуровневые взаимодействия, но вы должны понимать, как работает базовый механизм и на что влияет выбор способа хранения данных.

Одно из преимуществ HBase перед реляционными базами данных заключается в том, что столбцы не обязательно задавать заранее. Таким образом, если каждая метеостанция сейчас обладает как минимум тремя основными атрибутами, но вместе с ними существуют сотни дополнительных атрибутов, их вставка может быть выполнена без модификации схемы. Конечно, код чтения и записи в приложении придется изменить. Например, в нашем примере измененный код может перебирать содержимое `Result` вместо того, чтобы извлекать каждое значение напрямую.

Для выборки результатов наблюдений в нашем веб-приложении будут использоваться сканеры HBase¹.

Итак, имеется результат в формате `Map<ObservationTime, ObservedTemp>`. Мы используем контейнер `NavigableMap<Long, Integer>`, потому что он отсортирован, а метод `descendingMap()` позволяет получить доступ к результатам, упорядоченным как по возрастанию, так и по убыванию. Код приведен в листинге 13.4.

Листинг 13.4. Методы для выборки диапазона метеорологических данных из таблицы HBase

```

public NavigableMap<Long, Integer> getStationObservations(HTable table,
    String stationId, long maxStamp, int maxCount) throws IOException {
    byte[] startRow = RowKeyConverter.makeObservationRowKey(stationId, maxStamp);

```

¹ См. далее врезку «Сканеры». — Примеч. перев.

```

NavigableMap<Long, Integer> resultMap = new TreeMap<Long, Integer>();
Scan scan = new Scan(startRow);
scan.addColumn(DATA_COLUMNFAMILY, AIRTEMP_QUALIFIER);
ResultScanner scanner = table.getScanner(scan);
Result res = null;
int count = 0;
try {
    while ((res = scanner.next()) != null && count++ < maxCount) {
        byte[] row = res.getRow();
        byte[] value = res.getValue(DATA_COLUMNFAMILY, AIRTEMP_QUALIFIER);
        Long stamp = Long.MAX_VALUE -
            Bytes.toLong(row, row.length - Bytes.SIZEOF_LONG, Bytes.SIZEOF_LONG);
        Integer temp = Bytes.toInt(value);
        resultMap.put(stamp, temp);
    }
} finally {
    scanner.close();
}
return resultMap;
}
/***
 * Получение последних десяти наблюдений.
 */
public NavigableMap<Long, Integer> getStationObservations(HTable table,
    String stationId) throws IOException {
    return getStationObservations(table, stationId, Long.MAX_VALUE, 10);
}

```

Метод `getStationObservations()` получает идентификатор станции и диапазон, определяемый `maxStamp` и максимальным количеством строк (`maxCount`). Обратите внимание: возвращаемый объект `NavigableMap` отсортирован по убыванию времени. Чтобы прочитать данные, отсортированные по возрастанию времени, используйте `NavigableMap.descendingMap()`.

СКАНЕРЫ

Сканеры HBase похожи на курсоры традиционных баз данных или итераторы Java, однако они — в отличие от последних — должны закрываться после использования. Сканеры возвращают строки в определенном порядке. Чтобы получить сканер для таблицы HBase, пользователь вызывает `HTable.getScanner(scan)`, где параметр `scan` содержит настроенный экземпляр объекта `Scan`.

В экземпляре Scan можно передать строку, в которой должно начинаться или заканчиваться сканирование, столбцы, возвращаемые в результат, и (не обязательно) фильтр, выполняемый на стороне сервера. Интерфейс ResultScanner, возвращаемый при вызове HTable.getScanner(), выглядит следующим образом:

```
public interface ResultScanner extends Closeable, Iterable<Result> {  
    public Result next() throws IOException;  
    public Result [] next(int nbRows) throws IOException;  
    public void close();  
}
```

Вы можете запросить результаты следующей строки или нескольких строк. Каждый вызов next() требует обмена данными с сервером регионов, поэтому одновременная загрузка группы записей обеспечивает существенную экономию ресурсов.

Возможно, в приведенном примере преимущество хранения данных в виде `Long.MAX_VALUE - stamp` не столь очевидно. Оно проявляется более наглядно, когда вам потребуется получить новые наблюдения для заданного смещения и ограничения, как это часто требуется в веб-приложениях. Если бы наблюдения хранились с обычными временными метками, то мы могли бы эффективно получать только самые старые наблюдения для заданного смещения и ограничения. Для получения самых новых данных нам пришлось бы получить их все, а потом отделить нужные в конце. Одна из главных причин для перехода с РСУБД на HBase — поддержка подобных сценариев «первоочередной выдачи данных».

HBase и РСУБД

HBase и другие столбцово-ориентированные базы данных часто сравнивают с более традиционными и популярными реляционными базами данных или РСУБД. Несмотря на радикальные различия в реализации и предназначении, тот факт, что они являются потенциальными решениями одной задачи, означает, что при всех колоссальных различиях такое сравнение все же корректно.

Как упоминалось ранее, HBase представляет собой распределенную столбцово-ориентированную систему хранения данных. Она предоставляет возможность, недоступную для Hadoop: поддержку произвольных операций чтения и записи на базе HDFS. При проектировании системы изначально ставились такие цели, как

масштабируемость в обоих направлениях: вертикальном (количество строк данных миллиарды) и горизонтальном (количество столбцов — миллионы), а также возможность автоматического горизонтального разбиения и репликации по тысячам узлов. Схемы таблиц отражают физическую организацию хранения данных, создавая систему для эффективной сериализации, хранения и выборки структур данных. Задача разработчика приложения — правильно организовать хранение и выборку.

Строго говоря, для причисления к категории РСУБД база данных должна удовлетворять *12 правилам Кодда*. Типичная РСУБД — строчно-ориентированная база данных с фиксированной схемой, свойствами ACID¹ и полнофункциональным ядром запросов SQL. Особое внимание уделяется сильной согласованности, целостности на уровне ссылок, абстрагированию от физического уровня и поддержке сложных запросов с использованием языка SQL. Разработчик может легко создавать вторичные индексы, выполнять сложные внешние и внутренние соединения, подсчитывать, суммировать, сортировать и группировать данные по разным таблицам, строкам и столбцам.

В большинстве приложений, работающих с данными малого и среднего объема, решения РСУБД с открытым кодом (такие, как MySQL и PostgreSQL) остаются вне конкуренции по простоте использования, гибкости, зрелости и широте возможностей. Однако если вам потребуется провести масштабирование по размеру базы данных и(или) параллелизму чтения/записи, вскоре выясняется, что удобства РСУБД сопряжены с огромными потерями производительности, а механизм распределения данных чрезвычайно затруднен. При масштабировании РСУБД обычно приходится нарушать правила Кодда, ослаблять ограничения ACID, отказываться от традиционной практики администрирования баз данных — и, по сути, терять большинство полезных свойств, из-за которых с реляционными базами данных так удобно работать.

Масштабирование успешного сервиса

Ниже приведено краткое описание типичной истории масштабирования РСУБД успешно развивающегося сервиса:

Исходный запуск

Переход от локальной рабочей станции к совместно используемому экземпляру MySQL с хорошо определенной схемой.

¹ Атомарность, согласованность, изолированность, надежность (Atomicity, Consistency, Isolation, Durability). — Примеч. перев.

Сервис становится более популярным; слишком много обращений по чтению к базе данных

Добавление Memcached для кэширования частых запросов. Операции чтения уже не соответствуют критериям ACID в полной мере; срок действия кэшированных данных должен быть ограничен.

Популярность сервиса продолжает расти; слишком много обращений по записи к базе данных

Вертикальное масштабирование MySQL с покупкой мощного сервера с 16 ядрами, 128 Гбайт памяти и банком сверхбыстрых жестких дисков. Стоит дорого.

Новые возможности увеличивают сложность запросов; слишком много соединений данных

Денормализация данных для уменьшения количества соединений.

Растущая популярность превышает возможности сервера; все работает слишком медленно

Отказ от любых вычислений на стороне сервера.

Некоторые запросы по-прежнему обрабатываются слишком медленно

Периодическая опережающая материализация самых сложных запросов, попытки отказаться от соединения в большинстве случаев.

С чтением все в порядке, но запись происходит все медленнее

Отказ от вторичных индексов и триггеров.

На этой стадии не существует однозначных решений проблем масштабирования. В любом случае необходимо начать с горизонтального масштабирования. Вы пытаетесь самостоятельно организовать некое подобие разбиения для самых больших таблиц или же ищете коммерческие решения.

Многие приложения, организации и веб-сайты успешно реализовали масштабируемые, отказоустойчивые распределенные системы данных на базе РСУБД — скорее всего, используя многие из перечисленных стратегий. Но в итоге получается нечто такое, что не может считаться полноценной РСУБД из-за отказа от функций и удобств ради компромиссов и сложностей. Любая форма нисходящей репликации или внешнего кэширования вводит слабые зависимости в денормализованные данные. Неэффективность соединений и вторичных индексов означает, что почти все запросы превращаются в операции поиска по первичному ключу. Распределенные транзакции становятся сущим кошмаром. Возникает невероятно сложная сетевая топология с отдельным кластером для кэширования. Даже в этой системе после всех компромиссов приходится беспокоиться о возможных сбоях первичного сервера и пугающей возможности 10-кратного увеличения объема данных и нагрузки в ближайшие несколько месяцев.

HBase

На сцену выходит система HBase, обладающая следующими характеристиками:

Отсутствие реальных индексов

Строки данных хранятся последовательно, как и столбцы в каждой строке. Соответственно, исчезают проблемы с разрастанием индексов, а производительность вставки не зависит от размера таблицы.

Автоматическое разбиение

В процессе роста таблицы автоматически разбиваются на регионы и распределяются по всем доступным узлам.

Автоматическое линейное масштабирование с вводом новых узлов

Администратор добавляет узел, настраивает ссылку на существующий кластер и запускает сервер регионов. Происходит автоматическая перебалансировка регионов с равномерным распределением нагрузки.

Стандартное оборудование

Кластеры строятся из узлов стоимостью \$1000–5000, а не \$50 000. РСУБД с интенсивным вводом/выводом требуют более дорогостоящего оборудования.

Отказоустойчивость

Большое количество узлов означает, что каждый узел относительно незнаком. Простой отдельных узлов не создают значительных проблем.

Пакетная обработка

Интеграция с MapReduce позволяет выполнять полностью параллельные распределенные задания с учетом локальности данных.

Если вам не дают заснуть тревожные мысли о базе данных (проблемы работоспособности, масштаба или скорости), стоит серьезно рассмотреть возможность перехода с РСУБД на HBase. Используйте решение, изначально ориентированное на масштабирование — вместо решений, основанных на сокращении функциональности и затратах на то, что когда-то работало. При использовании HBase программное обеспечение бесплатно, оборудование дешево, а распределение данных происходит легко и естественно.

Пример из практики: HBase в Streamy.com

Streamy.com — агрегатор новостей в реальном времени и платформа социальных сетей. Из-за широты функциональности мы начали со сложной реализации,

построенной на основе PostgreSQL. Это был замечательный продукт с хорошим сообществом и красивой кодовой базой. Мы сделали все возможное, чтобы система сохраняла производительность в процессе масштабирования — вплоть до модификации исходного кода PostgreSQL под наши потребности. Изначально пользуясь всеми преимуществами РСУБД, мы обнаружили, что со временем нам пришлось постепенно от них отказываться. Попутно все участники нашей группы превратились в администраторов БД.

Нам удалось решить многие проблемы, с которыми мы столкнулись, но две из них в конечном итоге заставили нас искать другое решение за пределами мира РСУБД.

Streamy обходит тысячи RSS-каналов и агрегирует сотни миллионов элементов данных. Кроме необходимости организовать хранение этих данных, один из наших сложных запросов читает список всех элементов из набора источников. Теоретически один запрос может затрагивать несколько тысяч источников и все их элементы.

Очень большие таблицы элементов

Сначала мы использовали одну таблицу элементов данных, но из-за большого количества вторичных индексов операции вставки и обновления выполнялись очень медленно. Мы начали делить элементы на несколько таблиц, объединенных связями «один к одному», отделять статические поля от динамических, группировать поля по способам обращения и денормализовать все подряд. Даже после этих изменений одно обновление требовало замены целой записи, так что отслеживание статистики по элементам плохо масштабировалось. Замена записей и необходимость обновления индексов были неотъемлемыми свойствами используемой РСУБД. Мы провели разбиение своих таблиц, что было не особо сложно сделать из-за естественного деления по времени, но сложность быстро вышла из-под контроля. Требовалось другое решение!

Очень большие слияния с сортировкой

Операции слияния с сортировкой со списками, упорядоченными по времени, типичны для многих приложений Web 2.0. Запрос SQL может выглядеть примерно так:

```
SELECT id, stamp, type FROM streams
  WHERE type IN ('type1', 'type2', 'type3', 'type4', ..., 'typeN')
    ORDER BY stamp DESC LIMIT 10 OFFSET 0;
```

Предполагая, что `id` является первичным ключом для `streams`, а `stamp` и `type` имеют вторичные индексы, планировщик запросов РСУБД интерпретирует этот запрос следующим образом:

```
MERGE (
    SELECT id, stamp, type FROM streams
    WHERE type = 'type1' ORDER BY stamp DESC,
    ...
    SELECT id, stamp, type FROM streams
    WHERE type = 'typeN' ORDER BY stamp DESC
) ORDER BY stamp DESC LIMIT 10 OFFSET 0;
```

Проблема в том, что нас интересуют только старшие 10 значений, но планировщик запросов материализует всю операцию слияния, а проверяет ограничение в конце. Простая пирамидальная сортировка по каждому типу позволит досрочно завершить обработку после того, как будет известна старшая «десятка». В нашем случае каждый тип может содержать десятки тысяч идентификаторов, так что материализация всего списка и его сортировка будет чрезвычайно медленной и ненужной операцией. Мы даже написали специальный сценарий PL/Python, который выполнял пирамидальную сортировку с использованием серии запросов следующего вида:

```
SELECT id, stamp, type FROM streams
WHERE type = 'typeN'
ORDER BY stamp DESC LIMIT 1 OFFSET 0;
```

Практически во всех случаях наш сценарий превосходил по производительности «родную» реализацию SQL и стратегию планировщика запросов. В худшем для SQL случае написанная на Python процедура выполнялась на порядок быстрее. Оказалось, что мы постоянно пытаемся перехитрить планировщик запросов.

И в этом отношении нам также требовалось другое решение.

Переход на HBase

Система на базе РСУБД всегда была способна правильно реализовать наши требования; проблемы возникали с масштабированием. Когда вы в первую очередь думаете о масштабировании и производительности, а не о правильности, вы начинаете всюду, где только можно, искать всевозможные трюки и оптимизации для конкретных сценариев использования, привязанных к предметной области. А когда вы реализуете собственные решения своих проблем с данными, сложность и дополнительные затраты ресурсов РСУБД становятся на вашем пути. Абстрагирование от уровня хранения данных и требования ACID — огромное подспорье и роскошь, которые не всегда можно позволить себе в системах, построенных для масштабирования. HBase — распределенное столбцово-ориентированное хранилище отсортированных данных... и почти ничего более. Единственный аспект,

абстрагированный от пользователя, — распределение данных, а именно с ним как раз и не хочется соприкасаться. С другой стороны, бизнес-логика чрезвычайно сильно специализирована и оптимизирована. Свои проблемы мы лучше решим самостоятельно, доверив HBase масштабирование хранилища, а не логику. Мы почувствовали себя по-настоящему свободно, когда вместо масштабирования данных нам удалось сосредоточиться на приложениях и логике.

В настоящее время мы используем таблицы с сотнями миллионов строк и десятками тысяч столбцов, а перспектива хранения миллиардов строк с миллионами столбцов кажется скорее интересной, чем устрашающей.

14 ZooKeeper

До настоящего момента мы занимались изучением крупномасштабной обработки данных. Эта глава отличается от предыдущих: она посвящена построению распределенных приложений с использованием сервиса распределенной координации ZooKeeper.

Написать распределенное приложение непросто — прежде всего из-за возможности неполных сбоев. Если при передаче сообщения по сети между двумя узлами происходит сбой, отправитель не знает, было ли получено его сообщение. Может, оно добралось до получателя до сбоя сети, а может, и нет. А может, процесс получателя прекратил существование. Отправитель может узнать о случившемся только одним способом: заново связаться с получателем и запросить информацию. Подобная ситуация называется частичным сбоем: мы даже не знаем точно, произошел ли сбой.

ZooKeeper не может устранить частичные сбои, потому что они присущи распределенным системам. И конечно, он не скрывает частичные сбои¹. Вместо этого ZooKeeper предоставляет в ваше распоряжение средства построения распределенных приложений, способные безопасно обработать частичные сбои.

ZooKeeper также обладает следующими характеристиками:

¹ Это основная мысль статьи Дж. Уолдо (J.Waldo) «A Note on Distributed Computing» (1994), http://research.sun.com/techrep/1994/smli_tr-94-29.pdf: распределенное программирование принципиально отличается от локального, и различия просто невозможно замаскировать.

Простота

По сути, ZooKeeper представляет собой упрощенную файловую систему, предоставляющую ряд простых операций и несколько дополнительных абстракций (таких, как упорядочение и оповещения).

Выразительность

Примитивы ZooKeeper образуют богатый набор структурных элементов, которые могут использоваться для построения разнообразных координационных структур данных и протоколов: распределенные очереди, распределенные блокировки, выбор ведущего и т. д.

Высокая доступность

ZooKeeper работает на многих машинах и обеспечивает высокую доступность. ZooKeeper поможет избежать возникновения в системе простых единичных точек сбоев, что повышает надежность ваших приложений.

Упрощение слабосвязанных взаимодействий

Участники взаимодействий ZooKeeper не обязаны знать друг о друге. Например, ZooKeeper может использоваться как механизм организации встреч, чтобы процессы, не знающие о существовании друг друга (и не располагающие подробной информацией о сети), могли обнаружить и взаимодействовать друг с другом. Координируемые стороны даже не обязаны существовать в одно время — один процесс может оставить ZooKeeper сообщение, которое будет прочитано другим процессом уже после завершения первого.

Библиотека

ZooKeeper предоставляет общий репозиторий кода основных паттернов координации с открытым кодом. Программисты избавляются от необходимости самостоятельной реализации стандартных протоколов (которые часто бывает трудно правильно реализовать). Сообщество постепенно дополняет и совершенствует библиотеки ко всеобщей пользе.

ZooKeeper также обладает высокой производительностью. В Yahoo!, где была создана система, пропускная способность кластера ZooKeeper составляла свыше 10 000 операций в секунду для сгенерированной сотнями клиентов нагрузки, в которой в основном преобладала запись. Для нагрузок с преобладающим чтением (более типичный вариант) пропускная способность в несколько раз выше¹.

¹ Подробные результаты тестирования представлены в превосходной статье «ZooKeeper: Wait-free coordination for Internet-scale systems» Патрика Ханта (Patrick Hunt), Махадева Конара (Mahadev Konar), Флавио П. Джункеира (Flavio P. Junqueira) и Бенджамина Рида (Benjamin Reed)(USENIX Annual Technology Conference, 2010).

Установка и запуск ZooKeeper

Первые эксперименты с ZooKeeper проще всего проводить в автономном режиме с одним сервером ZooKeeper. Например, это можно сделать на машине разработки. Для работы ZooKeeper требуется Java 6; не забудьте установить пакет заранее. Для запуска ZooKeeper в системе Windows устанавливать Cygwin не обязательно — существуют Windows-версии сценариев ZooKeeper (Windows поддерживается только как платформа разработки, но не как платформа эксплуатации).

Загрузите стабильную версию ZooKeeper со страницы Apache ZooKeeper по адресу <http://zookeeper.apache.org/releases.html> и распакуйте архив в подходящий каталог:

```
% tar xzf zookeeper-x.y.z.tar.gz
```

В поставку ZooKeeper входят двоичные файлы для взаимодействия с сервисом. Каталог с этими файлами удобно включить в путь командной строки:

```
% export ZOOKEEPER_INSTALL=/home/tom/zookeeper-x.y.z
% export PATH=$PATH:$ZOOKEEPER_INSTALL/bin
```

Прежде чем запускать ZooKeeper, необходимо настроить конфигурационный файл. Он называется *zoo.cfg* и находится в подкаталоге *conf* (хотя его также можно поместить в каталог */etc/zookeeper* или в каталог, определяемой переменной окружения *ZOOCFGDIR*). Пример:

```
tickTime=2000
dataDir=/Users/tom/zookeeper
clientPort=2181
```

Файл *zoo.cfg* представляет собой стандартный файл свойств Java. Три свойства, определяемые в этом примере, составляют необходимый минимум для запуска ZooKeeper в автономном режиме. Свойство *tickTime* определяет базовую единицу времени в ZooKeeper (задается в миллисекундах); свойство *dataDir* определяет каталог локальной файловой системы, в котором ZooKeeper хранит информацию; свойство *clientPort* определяет порт, на котором ZooKeeper прослушивает клиентские подключения (стандартное значение — 2181). Задайте *dataDir* значение, соответствующее вашей файловой системе.

После определения конфигурации можно запустить локальный сервер ZooKeeper:

```
% zkServer.sh start
```

Чтобы проверить работоспособность ZooKeeper, отправьте команду *ruok* на клиентский порт через *nc* (*telnet* тоже подойдет):

```
% echo ruok | nc localhost 2181
imok
```

Другие команды для управления ZooKeeper перечислены в табл. 14.1.

Таблица 14.1. Команды ZooKeeper

Категория	Команда	Описание
Состояние сервера	ruok	Если сервер работает и не находится в состоянии ошибки, выводит сообщение «imok»
	conf	Выводит описание конфигурации сервера (из zoo.cfg)
	envi	Выводит информацию окружения сервера, включая версию ZooKeeper, версию Java и другие системные свойства
	srvr	Выводит статистику сервера, включая статистику задержки, количество z-узлов и режим работы сервера (автономный, ведущий или ведомый)
	stat	Выводит статистику сервера и подключенных клиентов
	srst	Сбрасывает статистику сервера
Клиентские подключения	isro	Сообщает, находится ли сервер в режиме только чтения (ro) или в режиме чтения/записи (rw)
	dump	Выводит список всех сеансов и эфемерных z-узлов для ансамбля. Для выполнения этой команды необходимо подключение к ведущему (см. srvr)
	cons	Выводит статистику подключений для клиентов сервера
Сторожа	crst	Сбрасывает статистику сервера
	wchs	Выводит сводную информацию о сторожах на сервере
	wchc	Перечисляет всех сторожей сервера по подключениям. Внимание: при большом количестве сторожей выполнение команды может отрицательно повлиять на быстродействие сервера
Мониторинг	wchp	Перечисляет всех сторожей сервера по путям z-узлов. Внимание: при большом количестве сторожей выполнение команды может отрицательно повлиять на быстродействие сервера
	mntr	Выводит статистику сервера в формате свойств Java; данные могут использоваться в качестве ввода для таких систем мониторинга, как Ganglia и Nagios

Кроме команды `mntr`, ZooKeeper предоставляет статистику через JMX. За более подробной информацией обращайтесь к документации ZooKeeper по адресу <http://zookeeper.apache.org/>. Некоторые инструменты мониторинга и решения содержатся в каталоге `src/contrib` дистрибутива.

Пример

Допустим, имеется группа серверов, предоставляющих клиентам некоторый сервис. Мы хотим, чтобы клиенты нашли один из серверов, который они могли бы использовать для обслуживания. Одной из проблем оказывается ведение списка серверов в группе.

Разумеется, список принадлежности не может храниться на одном узле сети, так как сбой этого узла приведет к отказу всей системы (список должен находиться в режиме высокой доступности). Представьте, что у нас появился надежный механизм хранения списка. Остается другая проблема — как исключить сервер из списка в случае сбоя? За исключение сбойных серверов должен отвечать некий процесс, однако на самих серверах он находиться не может, потому что серверы не работают!

То Наше описание представляет не пассивную распределенную структуру данных, а активную, которая может изменять состояние при возникновении внешних событий. ZooKeeper предоставляет такой сервис; посмотрим, как построить на его основе приложение для ведения списка принадлежности к группе.

Реализация списка принадлежности в ZooKeeper

ZooKeeper можно рассматривать как реализацию файловой системы с высокой доступностью. В ZooKeeper нет файлов и каталогов, но имеется единая концепция *z-узла* (*znode*), который действует и как контейнер данных (по аналогии с файлом), и как контейнер для других *z-узлов* (по аналогии с каталогом). *Z-узлы* образуют иерархическое пространство имен; естественный механизм построения списка принадлежности — создание родительского *z-узла* с именем группы и *дочерних узлов* с именами участников группы (серверов). Иерархия изображена на рис. 14.1.

В нашем примере *z-узлы* не содержат данных, но в реальном приложении, скорее всего, в них будет храниться информация об участниках группы (например, имена хостов).

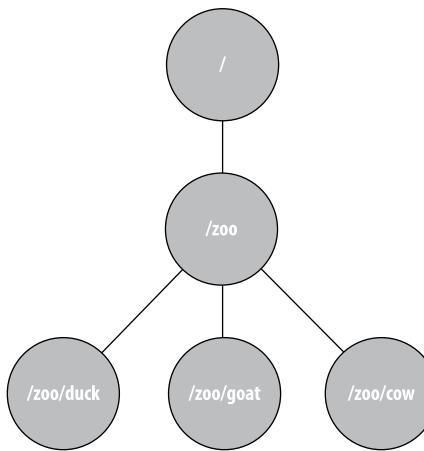


Рис. 14.1. Z-узлы ZooKeeper

Создание группы

Наше знакомство с ZooKeeper Java API начнется с написания программы создания z-узла группы (`/zoo` в листинге 14.1).

Листинг 14.1. Программа для создания в ZooKeeper z-узлов, представляющих группу

```
public class CreateGroup implements Watcher {

    private static final int SESSION_TIMEOUT = 5000;

    private ZooKeeper zk;
    private CountDownLatch connectedSignal = new CountDownLatch(1);

    public void connect(String hosts) throws IOException, InterruptedException {
        zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
        connectedSignal.await();
    }

    @Override
    public void process(WatchedEvent event) { // Watcher interface
        if (event.getState() == KeeperState.SyncConnected) {
            connectedSignal.countDown();
        }
    }
}
```

```
public void create(String groupName) throws KeeperException,  
    InterruptedException {  
    String path = "/" + groupName;  
    String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,  
        CreateMode.PERSISTENT);  
    System.out.println("Created " + createdPath);  
}  
  
public void close() throws InterruptedException {  
    zk.close();  
}  
  
public static void main(String[] args) throws Exception {  
    CreateGroup createGroup = new CreateGroup();  
    createGroup.connect(args[0]);  
    createGroup.create(args[1]);  
    createGroup.close();  
}
```

}

Метод `main()` создает экземпляр `CreateGroup` и вызывает его метод `connect()`. Метод создает новый экземпляр `ZooKeeper` — центрального класса клиентского API, обеспечивающего связь между клиентом и сервисом `ZooKeeper`. Конструктор получает три аргумента: адрес хоста (и не обязательно — порт, по умолчанию 2181) сервиса `ZooKeeper`¹; тайм-аут сеанса в миллисекундах (мы задаем его равным 5 секундам) — см. далее; и экземпляр объекта `Watcher`. Объект `Watcher` получает от `ZooKeeper` обратные вызовы, информирующие о различных событиях. В нашем сценарии `CreateGroup` реализует `Watcher`, поэтому мы передаем его конструктору `ZooKeeper`.

При создании экземпляра `ZooKeeper` создается программный поток для подключения к сервису `ZooKeeper`. Вызов конструктора немедленно возвращает управление, поэтому, прежде чем использовать объект `ZooKeeper`, важно дождаться установления связи. Мы используем класс `Java CountDownLatch` (из пакета `java.util.concurrent`) для блокировки выполнения до момента готовности экземпляра `ZooKeeper`. Здесь-то нам и пригодится `Watcher`. Интерфейс `Watcher` состоит из одного метода:

```
public void process(WatchedEvent event);
```

Когда клиент подключается к `ZooKeeper`, `Watcher` получает вызов своего метода `process()` с событием, сообщающим о подключении. При получении события

¹ Для реплицированного сервиса `ZooKeeper` параметр содержит разделенный запятыми список серверов (хост и не обязательный порт), входящих в ансамбль.

подключения (представленного перечислением `Watcher.Event.KeeperState` со значением `SyncConnected`) мы уменьшаем счетчик в `CountDownLatch`, используя его метод `countDown()`. Объект создается со значением 1, представляющим количество событий, которые должны произойти перед освобождением всех ожидающих потоков. После однократного вызова `countDown()` счетчик достигает нуля, и метод `await()` возвращает управление.

Метод `connect()` вернул управление, и следующим вызываемым методом `CreateGroup` будет метод `create()`. В этом методе мы создаем новый z-узел ZooKeeper, используя метод `create()` экземпляра ZooKeeper. В аргументах передаются путь (представленный строкой), содержащее z-узла (байтовый массив, `null` в нашем случае), список ACL (полностью открытый, позволяющий любому клиенту выполнять чтение или запись с z-узлом) и тип создаваемого z-узла.

Z-узлы делятся на эфемерные и постоянные. Эфемерный z-узел будет удален сервисом ZooKeeper при отключении создавшего его клиента — либо отдавшего команду на отключение, либо завершающего выполнение по какой-либо причине. Вместе с тем постоянный z-узел не удаляется при отключении клиента. Мы хотим, чтобы z-узел, представляющий группу, существовал за пределами жизненного цикла создавшей его программы, поэтому мы создаем постоянный z-узел.

Возвращаемое значение метода `create()` содержит путь, созданный ZooKeeper. Мы используем его для вывода сообщения об успешном создании пути. При рассмотрении последовательных z-узлов вы увидите, что путь, возвращаемый `create()`, может отличаться от пути, переданного методу.

Чтобы увидеть программу в действии, запустите ZooKeeper на локальной машине и введите следующие команды:

```
% export CLASSPATH=ch14/target/classes/:$ZOOKEEPER_INSTALL/*:$ZOOKEEPER_INSTALL
    /lib/*:\$ZOOKEEPER_INSTALL/conf
% java CreateGroup localhost zoo
Created /zoo
```

Присоединение к группе

Следующая часть приложения — программа для регистрации участников группы. Каждый новый участник запускает программу и присоединяется к группе. Программа, завершающая работу, должна быть удалена из группы; для этого мы создаем эфемерный z-узел, представляющий ее в пространстве имен ZooKeeper.

Программа `JoinGroup` реализует эту схему, а ее код приведен в листинге 14.2. Логика создания и подключения к экземпляру ZooKeeper была посредством

рефакторинга вынесена в базовый класс `ConnectionWatcher`, приведенный в листинге 14.3.

Листинг 14.2. Программа присоединения к группе

```
public class JoinGroup extends ConnectionWatcher {

    public void join(String groupName, String memberName) throws KeeperException,
        InterruptedException {
        String path = "/" + groupName + "/" + memberName;
        String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,
            CreateMode.EPHEMERAL);
        System.out.println("Created " + createdPath);
    }

    public static void main(String[] args) throws Exception {
        JoinGroup joinGroup = new JoinGroup();
        joinGroup.connect(args[0]);
        joinGroup.join(args[1], args[2]);

        // Продолжать существование до уничтожения процесса
        // или прерывания программного потока
        Thread.sleep(Long.MAX_VALUE);
    }
}
```

Листинг 14.3. Вспомогательный класс, ожидающий установления связи с ZooKeeper

```
public class ConnectionWatcher implements Watcher {

    private static final int SESSION_TIMEOUT = 5000;
    protected ZooKeeper zk;
    private CountDownLatch connectedSignal = new CountDownLatch(1);
    public void connect(String hosts) throws IOException, InterruptedException {
        zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
        connectedSignal.await();
    }

    @Override
    public void process(WatchedEvent event) {
        if (event.getState() == KeeperState.SyncConnected) {
            connectedSignal.countDown();
        }
    }
}
```

продолжение ⇨

Листинг 14.3 (продолжение)

```
public void close() throws InterruptedException {
    zk.close();
}
```

Код `JoinGroup` очень похож на код `CreateGroup`. Он создает эфемерный z-узел как дочерний для z-узла группы в методе `join()`, а затем переходит в ожидание до принудительного завершения процесса. Как будет показано позднее, при завершении `ZooKeeper` уничтожает эфемерный z-узел.

Вывод списка участников группы

Теперь нам нужна программа для вывода списка участников группы (листинг 14.4).

Листинг 14.4. Программа для вывода списка участников группы

```
public class ListGroup extends ConnectionWatcher {

    public void list(String groupName) throws KeeperException,
        InterruptedException {
        String path = "/" + groupName;

        try {
            List<String> children = zk.getChildren(path, false);
            if (children.isEmpty()) {
                System.out.printf("No members in group %s\n", groupName);
                System.exit(1);
            }
            for (String child : children) {
                System.out.println(child);
            }
        } catch (KeeperException.NoNodeException e) {
            System.out.printf("Group %s does not exist\n", groupName);
            System.exit(1);
        }
    }

    public static void main(String[] args) throws Exception {
        ListGroup listGroup = new ListGroup();
        listGroup.connect(args[0]);
        listGroup.list(args[1]);
        listGroup.close();
    }
}
```

В методе `list()` мы вызываем метод `getChildren()` с передачей пути к z-узлу и сторожевого флага, получаем список дочерних путей и выводим его. Установка сторожа для z-узла заставляет зарегистрированный объект `Watcher` срабатывать при изменении состояния z-узла. Хотя здесь эта возможность не используется, отслеживание дочерних z-узлов позволяет программе получать оповещения о присоединении или выходе из группы или удалении последней. Наша программа перехватывает исключение `KeeperException.NoNodeException`, которое инициируется в том случае, если z-узел группы не существует.

Посмотрим, как работает `ListGroup`. Как и ожидалось, группа `zoo` пуста, так как в нее еще не был добавлен ни один участник:

```
% java ListGroup localhost zoo  
No members in group zoo
```

Используем программу `JoinGroup` для добавления участников. Они запускаются как фоновые процессы, так как самостоятельно эти процессы не завершаются (из-за команды `sleep`):

```
% java JoinGroup localhost zoo duck &  
% java JoinGroup localhost zoo cow &  
% java JoinGroup localhost zoo goat &  
% goat_pid=$!
```

Последняя строка сохраняет идентификатор процесса Java, запустившего программу, которая добавляет участника `goat`. Он нам вскоре понадобится для уничтожения процесса после проверки участников:

```
% java ListGroup localhost zoo  
goat  
duck  
cow
```

Чтобы исключить участника из группы, мы уничтожаем его процесс:

```
% kill $goat_pid
```

Через несколько секунд процесс исчезает из группы, потому что сеанс ZooKeeper процесса был завершен (тайм-аут установлен равным 5 секундам), а соответствующий эфемерный узел был удален:

```
% java ListGroup localhost zoo  
duck  
cow
```

Итак, что же было сделано в этом примере? Мы построили список групп узлов, участвующих в распределенной системе. Узлы могут не располагать информацией друг о друге. Клиент, который хочет использовать узлы из списка (например, для

выполнения некоторой операции), может получить информацию о них — при этом сами узлы ничего не будут знать о существовании клиента.

Наконец, стоит заметить, что участие в группе не заменяет обработки сетевых ошибок при взаимодействии с узлом. Даже если узел является участником группы, при работе с ним может произойти сбой; такие сбои должны обрабатываться стандартными способами (повторная попытка, попытка подключения к другому участнику группы и т. д.).

Инструменты командной строки ZooKeeper

В поставку ZooKeeper включена программа командной строки для взаимодействия с пространством имен ZooKeeper. Ее можно использовать для получения списка z-узлов в иерархии /zoo:

```
% zkCli.sh localhost ls /zoo
Processing ls
WatchedEvent: Server state change. New state: SyncConnected
[duck, cow]
```

При выполнении без аргументов программа выводит инструкции по использованию.

Удаление группы

В завершение примера давайте посмотрим, как удалить группу. Класс ZooKeeper предоставляет метод `delete()`, которому передается путь и номер версии. ZooKeeper удаляет z-узел только в том случае, если указанный номер версии совпадает с номером версии удаляемого z-узла — механизм оптимистичной блокировки, позволяющий клиентам выявлять конфликты за право модификации z-узла. Впрочем, проверку версии можно обойти — если передать вместо версии `-1`, z-узел будет удален независимо от его версии.

В ZooKeeper нет операции рекурсивного удаления, поэтому дочерние z-узлы необходимо удалить перед родителями. Именно это делает класс `DeleteGroup`, который удаляет группу и всех ее участников (листинг 14.5).

Листинг 14.5. Программа для удаления группы и ее участников

```
public class DeleteGroup extends ConnectionWatcher {

    public void delete(String groupName) throws KeeperException,
        InterruptedException {
```

```
String path = "/" + groupName;

try {
    List<String> children = zk.getChildren(path, false);
    for (String child : children) {
        zk.delete(path + "/" + child, -1);
    }
    zk.delete(path, -1);
} catch (KeeperException.NoNodeException e) {
    System.out.printf("Group %s does not exist\n", groupName);
    System.exit(1);
}

public static void main(String[] args) throws Exception {
    DeleteGroup deleteGroup = new DeleteGroup();
    deleteGroup.connect(args[0]);
    deleteGroup.delete(args[1]);
    deleteGroup.close();
}
}
```

Наконец, мы можем удалить созданную ранее группу zoo:

```
% java DeleteGroup localhost zoo
% java ListGroup localhost zoo
Group zoo does not exist
```

Сервис ZooKeeper

ZooKeeper — высокопроизводительная система координации с высокой доступностью. В этом разделе рассматривается природа предоставляемого сервиса: модель, операции и реализация.

Модель данных

ZooKeeper поддерживает иерархическое дерево узлов, называемых z-узлами. В z-узлах хранятся данные, и с ними связываются списки ACL. Система ZooKeeper была спроектирована для координации (для которой обычно используются малые файлы данных), а не хранения больших объемов информации, поэтому объем данных, хранимых в любом z-узле, не может превышать 1 Мбайт.

Обращения к данным обладают свойством атомарности. Клиент, читающий данные, хранимые в узле, никогда не получает часть данных; либо все данные доставляются полностью, либо операция чтения завершается неудачей. Аналогично операция записи заменяет все данные, связанные с z-узлом. ZooKeeper гарантирует, что запись либо будет завершена успешно, либо произойдет неудача; частичная запись, при которой сохраняется только часть данных, записанных клиентом, невозможна. ZooKeeper не поддерживает операцию присоединения данных. Эти характеристики отличают ZooKeeper от системы HDFS, спроектированной для хранения больших объемов данных с потоковым доступом и поддержкой присоединения.

Для ссылки на z-узлы используются пути, которые в ZooKeeper представлены строками символов в кодировке Юникод, разделенными символом / (как и пути файловых систем Unix). Пути должны быть абсолютными, поэтому они всегда начинаются с символа /. Кроме того, пути каноничны, то есть каждый путь имеет единственное представление. Например, в Unix к файлу с путем /a/b можно обратиться через путь /a./b, потому что символ «.» обозначает текущий каталог. В ZooKeeper символ «.» не имеет особого смысла, и его присутствие в пути недопустимо (как и комбинации «..», обозначающей родительский каталог).

Компоненты пути строятся из символов Юникода с некоторыми ограничениями (перечисленными в справочной документации ZooKeeper). Стока «zookeeper» является зарезервированным словом и не может использоваться в качестве компонента пути. ZooKeeper использует поддерево /zookeeper для хранения управляемой информации — например, информации о квотах.

Обратите внимание: пути отличаются от URI, а в Java API они представляются классом `java.lang.String` вместо класса Hadoop `Path` (или класса `java.net.URI`).

Z-узлы обладают некоторыми свойствами, полезными для построения распределенных приложений. Эти свойства будут рассмотрены в следующих разделах.

Эфемерные z-узлы

Z-узлы делятся на два типа: эфемерные и постоянные. Тип z-узла задается в момент создания и не может изменяться позднее. Эфемерный z-узел удаляется ZooKeeper при завершении сеанса клиента, создавшего этот узел. Вместе с тем постоянный z-узел не привязывается к сеансу клиента и удаляется только по явному распоряжению клиента (не обязательно того, который его создал). Эфемерный z-узел не может иметь потомков, даже эфемерных. И хотя эфемерные узлы связаны с сеансом конкретного клиента, они видны всем клиентам (конечно, в зависимости от их политики ACL).

Эфемерные узлы идеально подходят для построения приложений, которым необходима информация о доступности некоторых распределенных ресурсов.

Приведенный ранее пример использует эфемерные z-узлы для реализации контроля принадлежности к группе, так что любой процесс может в любой момент получить информацию о составе группы.

Последовательные номера

Последовательным z-узлам ZooKeeper присваивает порядковый номер, который становится частью их имени. Если z-узел создается с установкой флага последовательного узла, то к имени присоединяется значение монотонно возрастающего счетчика (который ведется родительским z-узлом).

Например, если клиент создает последовательный z-узел с именем */a/b-*, то имя фактически созданного z-узла может иметь вид */a/b-3*¹. Если позднее будет создан другой последовательный z-узел с именем */a/b-*, ему присваивается имя с большим значением счетчика — например, */a/b-5*. В Java API фактический путь, назначенный последовательному z-узлу, передается клиенту в возвращаемом значении вызова `create()`.

Последовательные числа могут использоваться для глобального упорядочения событий в распределенных системах или для введения упорядочения клиентом. В разделе «Блокировка» на с. 641 будет показано, как использовать последовательные z-узлы для реализации блокировки совместно используемых ресурсов.

Сторожа

Сторожа позволяют клиентам получать оповещения об изменениях z-узлов. Они устанавливаются операциями сервиса ZooKeeper и срабатывают при выполнении других операций этого сервиса. Например, клиент может вызвать для z-узла операцию `exists`, одновременно установив сторожа. Если z-узел не существует, операция `exists` вернет `false`. Если позднее z-узел будет создан другим клиентом, сторож сработает, оповестив первого клиента о создании z-узла. О том, какие операции приводят к срабатыванию других операций, будет рассказано в следующем разделе.

Сторожа срабатывают только один раз². Чтобы получить повторное оповещение, клиент перерегистрирует сторожа. Если клиент из предыдущего примера захочет получать последующие оповещения о существовании z-узла (например, о его удалении), он должен снова вызвать операцию `exists` для установки нового сторожа.

В примере «Конфигурация» на с. 633 показано, как использовать сторожей для обновления конфигурации в кластере.

¹ Принято (хотя и не обязательно) завершать имена последовательных узлов дефисом, чтобы упростить их чтение и разбор в приложениях.

² Кроме обратных вызовов событий подключения, не требующих перерегистрации.

Операции

ZooKeeper поддерживает девять основных операций, перечисленных в табл. 14.2.

Таблица 14.2. Операции ZooKeeper

Операция	Описание
create	Создает z-узел (родительский z-узел уже должен существовать)
delete	Удаляет z-узел (который не должен иметь дочерних z-узлов)
exists	Проверяет существование z-узла и получает его метаданные
getACL, setACL	Получает/задает список ACL для z-узла
getChildren	Получает список дочерних узлов z-узла
getData, setData	Получает/задает данные, связанные с z-узлом
sync	Синхронизирует клиентское представление z-узла с ZooKeeper

Выполнение операций обновления в ZooKeeper зависит от проверки условия. Операция `delete` или `setData` должна задать номер версии обновляемого z-узла (который определяется по предыдущему вызову `exists`). Если номер версии не совпадает, обновление завершается неудачей. Обновления являются неблокирующими операциями, поэтому клиент, потерявший обновление (из-за того, что другой процесс успел обновить z-узел), может выбрать между повторной попыткой и выполнением другого действия, причем сделать это без блокирования других процессов.

Хотя ZooKeeper может рассматриваться как файловая система, некоторые примитивы файловых систем были опущены ради простоты. Так как файлы имеют небольшой размер, а их содержимое читается и записывается полностью, в операциях открытия, закрытия и поиска нет необходимости.



Операция `sync` не является аналогом `fsync()` в файловых системах POSIX. Как упоминалось ранее, запись в ZooKeeper выполняется атомарно, и успешная запись будет гарантированно сохранена в постоянном хранилище на большинстве серверов ZooKeeper. Однако операция чтения может отставать от актуального состояния сервиса ZooKeeper; операция `sync` позволяет клиенту обновить свою информацию до самого нового состояния. Эта тема более подробно рассматривается в разделе «Согласованность данных» на с. 628.

Пакетное обновление

Операция ZooKeeper `multi` объединяет несколько примитивных операций в пакет, который либо успешно выполняется, либо не выполняется как единое целое. Ситуация, в которой часть примитивных операций завершается успешно, а остальные отменяются, не возникает никогда.

Пакетное обновление чрезвычайно полезно для создания структур ZooKeeper, поддерживающих некоторый глобальный инвариант. Пример — ненаправленный граф: каждая вершина графа естественным образом представляется в ZooKeeper z-узлом, а добавление или удаление ребра требует обновления двух z-узлов, соответствующих вершинам графа, потому что они содержат ссылки друг на друга. При использовании примитивных операций ZooKeeper возможна ситуация, при которой другой клиент «увидит» граф в несогласованном состоянии, при котором одна вершина содержит ссылку на другую, а обратная ссылка отсутствует. Объединение обновлений двух z-узлов в одну составную операцию гарантирует атомарность обновления, так что между парами вершин никогда не появятся «висячие» ссылки.

API

Для клиентов ZooKeeper существует две основные языковые привязки: для Java и для C. Также существуют сторонние привязки для клиентов Perl, Python и REST. Для каждой привязки существует выбор между синхронным и асинхронным выполнением операций. С синхронным Java API вы уже знакомы. Сигнатура операции `exists`, которая возвращает либо объект `Stat`, инкапсулирующий метаданные z-узла, либо `null`, если z-узел не существует:

```
public Stat exists(String path, Watcher watcher) throws KeeperException,  
    InterruptedException
```

Асинхронный эквивалент, который также находится в классе ZooKeeper, выглядит так:

```
public void exists(String path, Watcher watcher, StatCallback cb, Object ctx)
```

В Java API все асинхронные методы возвращают `void`, так как результат операции передается через обратный вызов. Вызывающая сторона передает реализацию обратного вызова, метод которой будет вызван при получении ответа от ZooKeeper. В нашем случае это интерфейс `StatCallback`, который содержит следующий метод:

```
public void processResult(int rc, String path, Object ctx, Stat stat);
```

Аргумент `rc` содержит возвращаемый код, соответствующий кодам, определенным `KeeperException`. Ненулевой код представляет собой исключение; в этом случае

параметр `stat` содержит `null`. Аргументы `path` и `ctx` соответствуют эквивалентным аргументам, передаваемым клиентом методу `exists()`; они могут использоваться для идентификации запроса, на который отвечает данный обратный вызов. Параметр `ctx` может содержать произвольный объект, который используется клиентом, если путь не содержит достаточного контекста для однозначной идентификации запроса. Если он не нужен, ему может быть задано значение `null`.

Для языка С существует две общие библиотеки. Однопоточная библиотека `zookeeper_st` поддерживает только асинхронный API; она предназначена для платформ, на которых библиотека `pthread` недоступна или нестабильна. Большинство разработчиков использует многопоточную библиотеку `zookeeper_mt`, которая поддерживает как синхронный, так и асинхронный API. За подробностями о построении программ и использовании С API обращайтесь к файлу `README` в каталоге `src/c` дистрибутива ZooKeeper.

КАКОЙ API ИСПОЛЬЗОВАТЬ?

Оба API — синхронный и асинхронный — предоставляют одинаковую функциональность, так что выбор в основном зависит от стиля. Например, асинхронный API лучше подходит для модели программирования, управляемой событиями.

Асинхронный API позволяет организовать конвейерную обработку запросов, которая в некоторых ситуациях повышает производительность. Допустим, требуется прочитать большой набор z-узлов и независимо обработать их. При использовании синхронного API каждое чтение блокируется до возвращения управления, тогда как с асинхронным API вы можете очень быстро запустить все операции и обрабатывать ответы в отдельном программном потоке по мере их поступления.

Срабатывание сторожей

Для операций чтения `exists`, `getChildren` и `getData` могут устанавливаться сторожа, которые срабатывают по операциям записи: `create`, `delete` и `setData`. Операции ACL не участвуют в работе сторожей. При срабатывании сторожа генерируется событие, тип которого зависит как от сторожа, так и от операции, по которой он сработал:

- Сторож, установленный для операции `exists`, срабатывает при создании, удалении или обновлении данных отслеживаемого z-узла.

- Сторож, установленный для операции `getData`, срабатывает при удалении или обновлении данных отслеживаемого z-узла. При создании сторож не срабатывает, потому что для успешного выполнения `getData` z-узел уже должен существовать.
- Сторож, установленный для операции `getChildren`, срабатывает при создании или удалении z-узлов, являющихся дочерними по отношению к данному, или самого z-узла. Чтобы узнать, какой именно z-узел удаляется, достаточно проверить тип события: `NodeDeleted` означает, что удаляется сам z-узел, а `No-deChildrenChanged` — что удаляется дочерний z-узел.

Разные комбинации перечислены в табл. 14.3.

Таблица 14.3. Операции создания сторожей и соответствующие условия срабатывания

Создание сторожа	Условие срабатывания				<code>setData</code>
	создание z-узла	создание дочернего z-узла	удаление z-узла	удаление дочернего z-узла	
<code>exists</code>	<code>NodeCreated</code>		<code>NodeDeleted</code>		<code>Node-DataChanged</code>
<code>getData</code>			<code>NodeDeleted</code>		<code>Node-DataChanged</code>
<code>getChildren</code>		<code>Node-Children-Changed</code>	<code>NodeDeleted</code>	<code>Node-Children-Changed</code>	

В событие включается путь к z-узлу, так что для событий `NodeCreated` и `NodeDeleted` можно определить, какой z-узел был создан или удален, простой проверкой пути. Чтобы узнать, какие дочерние z-узлы изменились после события `NodeChildrenChanged`, необходимо снова вызвать `getChildren` для получения нового списка дочерних z-узлов. Аналогичным образом, для получения новых данных для события `NodeDataChanged` необходимо вызвать `getData`. В обоих случаях состояние z-узлов может измениться между получением события и выполнением операции чтения; учтите это при написании приложений.

Списки ACL

Z-узел создается со списком ACL, определяющим, кто может выполнять с ним некоторые операции.

Работа списков ACL основана на аутентификации — процедуре, в ходе которой клиент подтверждает свою подлинность. ZooKeeper поддерживает несколько схем аутентификации:

digest

Аутентификация выполняется на основе имени и пароля.

sasl

Аутентификация выполняется с использованием Kerberos.

ip

Аутентификация выполняется на основе IP-адреса клиента.

Клиент может пройти аутентификацию после создания сеанса ZooKeeper. Аутентификация не является обязательной, хотя список ACL z-узла может требовать аутентификации клиента — в этом случае клиент должен пройти аутентификацию для обращения к z-узлу. Пример использования схемы **digest** для выполнения аутентификации с именем пользователя и паролем:

```
zk.addAuthInfo("digest", "tom:secret".getBytes());
```

Содержимое списка ACL представляет собой комбинацию схемы аутентификации, идентификатор для этой схемы и набора разрешений. Например, чтобы разрешить клиенту с IP-адресом 10.0.0.1 доступ для чтения к z-узлу, следует установить в списке ACL для z-узла схему **ip**, идентификатор 10.0.0.1 и разрешение **READ**. В Java объект ACL создается следующим образом:

```
new ACL(Perms.READ,
new Id("ip", "10.0.0.1"));
```

Полный набор разрешений приведен в табл. 14.4. Обратите внимание: на операцию **exists** разрешения ACL не распространяются, поэтому любой клиент может вызывать **exists** и получить объект **Stat** для z-узла (или узнать, что z-узел не существует).

Таблица 14.4. Разрешения ACL

Разрешение ACL	Разрешенные операции
CREATE	create (для дочернего z-узла)
READ	getChildren getData
WRITE	setData
DELETE	delete (для дочернего z-узла)
ADMIN	setACL

В классе `ZooDefs.Ids` определяется ряд стандартных списков ACL, включая режим `OPEN_ACL_UNSAFE`, предоставляющий все разрешения (кроме `ADMIN`) всем пользователям.

В ZooKeeper также существует расширяемый механизм аутентификации, что позволяет при необходимости интегрировать сторонние системы аутентификации.

Реализация

ZooKeeper может работать в двух режимах. *Автономный режим*, в котором работает единственный сервер ZooKeeper, удобен для тестирования из-за своей простоты (он даже может встраиваться в модульные тесты), но не обеспечивает высокой доступности или защиты от сбоев. В процессе реальной эксплуатации ZooKeeper работает в *реплицированном режиме* в кластере, который называется *ансамблем*. ZooKeeper обеспечивает высокую доступность посредством репликации и может обслуживать клиентов при условии работоспособности большинства машин в ансамбле. Например, в ансамбле из 5 узлов любые две машины могут отказаться, а сервис продолжит работать, потому что большинство (три машины) остаются работоспособными. Учтите, что ансамбль из 6 узлов переживет сбой только двух машин, потому что при трех отказах оставшиеся три машины не составляют большинство. По этой причине в ансамбль обычно включается нечетное количество машин.

На концептуальном уровне работа ZooKeeper очень проста: системе нужно лишь следить за тем, чтобы каждое изменение дерева z-узлов реплицировалось на большинстве машин ансамбля. Если сбои произойдут менее чем на половине машин, то как минимум одна машина сохранит последнее состояние. Остальные оставшиеся реплики со временем синхронизируются по ней.

Тем не менее реализована эта простая идея нетривиально. ZooKeeper использует протокол Zab, работающий в две фазы, которые могут повторяться неопределенное количество раз:

Фаза 1: Выбор ведущего

Машины, входящие в ансамбль, проходят процесс выбора привилегированного участника, называемого *ведущим*. Остальные машины называются *ведомыми*. Эта фаза завершается после того, как большинство (или кворум) ведомых синхронизирует свое состояние с ведущим.

Фаза 2: Атомарная рассылка

Все запросы записи передаются ведущему, который рассыпает обновление среди ведомых. Когда большинство сохранит изменения, ведущий закрепляет обновление, а клиент получает подтверждение успешного обновления.

Протокол достижения консенсуса обладает атомарностью, поэтому изменение завершается либо успехом, либо неудачей. В целом процесс напоминает процесс двухфазного закрепления.

Если на ведущем происходит сбой, остальные машины снова проводят выборы ведущего и продолжают работать, как прежде, с новым ведущим. Если старый ведущий позднее восстановит работоспособность, он начинает работать как ведомый. Выборы ведущего проходят очень быстро — около 200 мс по одному из опубликованных результатов¹, и производительность от них не страдает.

Все машины ансамбля записывают обновления на диск перед тем, как обновлять свою копию дерева z-узлов, находящуюся в памяти. Запросы чтения могут обслуживаться любой машиной; так как они требуют лишь поиска по ключу в памяти, такие запросы обслуживаются очень быстро.

Согласованность данных

Понимание основ реализации ZooKeeper помогает понять гарантии согласованности данных, предоставляемые этим сервером. Термины «ведущий» и «ведомый» для машин ансамбля выбраны довольно удачно: они дают понять, что ведомый может отставать от ведущего на нескольких обновлениях. Это следует из того факта, что сохранение перед закреплением должно быть сохранено на большинстве машин, а не на всех машинах ансамбля. Представьте клиентов, подключенных к серверам ZooKeeper. Клиент может оказаться подключенным к ведущему, но он не может сознательно управлять этим, и даже не знает об этом факте² (см. рис. 14.2).

Каждому обновлению, вносимому в дерево z-узлов, присваивается глобально-универсальный идентификатор, называемый *zxid* (ZooKeeper transaction ID). Обновления упорядочены, поэтому если *zxid* *z1* меньше *z2*, то обновление *z1* произошло до *z2* — по мнению ZooKeeper, единственного авторитетного источника, определяющего порядок событий в распределенной системе.

Из архитектуры ZooKeeper следует ряд гарантий согласованности данных:

Сохранение последовательности

Обновления от любого конкретного клиента применяются в порядке их отправки. Таким образом, если клиент обновляет z-узел *z* значением *a*, а в

¹ По данным Yahoo! (<http://zookeeper.apache.org/doc/current/zookeeperOver.html>).

² ZooKeeper можно настроить так, чтобы ведущий не принимал клиентские подключения — в этом случае его единственной функцией становится координация обновлений. Для этого задайте свойству *leaderServes* значение *no*. Данный режим рекомендуется для ансамблей, содержащих более трех серверов.

последующей операции он обновляет его узел значением *b*, то ни один клиент уже не увидит *z* со значением *a* после того, как он увидел его со значением *b* (если к *z* не применяются другие обновления).

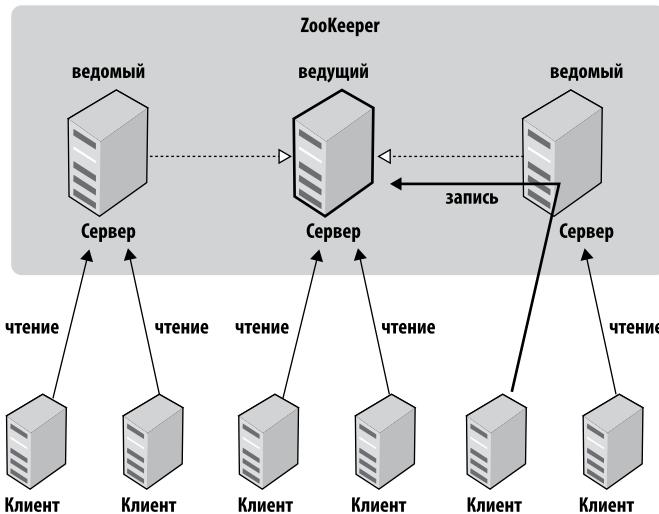


Рис. 14.2. Операции чтения обслуживаются ведомыми, а операции записи закрепляются ведущим

Атомарность

Обновления либо проходят полностью успешно, либо не проходят вообще. Если попытка обновления завершилась неудачей, ни один клиент ее не увидит.

Единый образ системы

Клиент видит одно представление системы независимо от того, к какому серверу он подключился. Если клиент подключается к новому серверу в ходе того же сеанса, он не увидит более старое состояние системы по сравнению с предыдущим сервером. Если на сервере происходит сбой и клиент пытается подключиться к другому серверу в ансамбле, то сервер, отстающий от сбояного, не будет принимать подключения от клиентов, пока не «догонит» его.

Устойчивость

Успешное обновление сохраняется, и оно уже не будет отменено. Это означает, что обновления переживают сбои серверов.

Своевременность

Отставание в восприятии системы любым клиентом ограничено и не превышает нескольких десятков секунд. Вместо того, чтобы предоставлять клиенту сильно устаревшие данные, сервер прерывает работу, заставляя клиента подключиться к серверу с более актуальной информацией.

По соображениям быстродействия запросы чтения обслуживаются из памяти сервера ZooKeeper и не участвуют в глобальном упорядочении операций записи. Это свойство может привести к видимой непоследовательности состояний ZooKeeper для клиентов, взаимодействующих с использованием внешних механизмов.

Сеансы

При настройке клиента ZooKeeper указывается список серверов ансамбля. При запуске клиент пытается подключиться к одному из серверов из списка. Если попытка подключения завершается неудачей, клиент опробует другой сервер из списка и так далее — либо ему удастся успешно подключиться к одному из них, либо происходит сбой, если все серверы ZooKeeper недоступны.

После подключения клиента к серверу ZooKeeper сервер создает для клиента новый сеанс. У сеанса имеется период тайм-аута, который определяется создавшим его приложением. Если сервер не получает запрос на протяжении периода тайм-аута, он может прервать сеанс. Такой сеанс нельзя открыть повторно, а все связанные с ним эфемерные узлы будут потеряны. Хотя прерывание сеансов по тайм-ауту является относительно редким событием, так как сеансы имеют большой срок жизни, приложение должно обрабатывать эту ситуацию (см. «Отказоустойчивое приложение ZooKeeper», с. 637).

Чтобы избежать прерывания сеанса, клиент отправляет периодические запросы в том случае, если простой в сеансе превысит определенный промежуток времени. (Клиентская библиотека ZooKeeper делает это автоматически, вам не нужно беспокоиться о поддержании сеанса в коде). Промежуток выбирается достаточно низким, чтобы клиент мог обнаружить сбой сервера (проявляющийся в тайм-ауте чтения) и подключиться к другому серверу в пределах периода тайм-аута сеанса.

Переход на другой сервер ZooKeeper осуществляется клиентом ZooKeeper автоматически. Очень важно, что сеансы (и связанные с ними эфемерные узлы) остаются действительными после того, как сбойный сервер будет заменен другим.

В ходе преодоления сбоя приложение будет получать оповещения об отключениях и подключениях. Оповещения сторожей не доставляются, пока клиент отключен, но будут доставлены при успешном повторном подключении. Если приложение попытается выполнить операцию в ходе подключения клиента к другому серверу,

попытка завершится неудачей. Это лишний раз подчеркивает важность обработки исключений потери соединения в реальных приложениях ZooKeeper (см. «Отказоустойчивое приложение ZooKeeper», с. 637).

Время

В ZooKeeper существует несколько временных параметров. Такт (tick) — фундаментальная единица времени в ZooKeeper, используемая серверами ансамбля для определения расписания их взаимодействий. Другие временные параметры определяются по продолжительности такта (или, по крайней мере, ограничиваются ею). Например, тайм-аут сеанса не может быть менее двух тактов или более 20. Если вы попробуете задать недопустимую продолжительность тайм-аута сеанса, она будет автоматически приведена к этим границам.

Стандартная продолжительность такта составляет 2 секунды (2000 миллисекунд). Таким образом, разрешенный тайм-аут сеанса лежит в пределах от 4 до 40 секунд. При выборе тайм-аута сеанса следует учитывать ряд факторов.

Низкое значение тайм-аута ускоряет выявление сбоев. В рассмотренном ранее примере с принадлежностью к группе тайм-аут сеанса определяет время исключения из группы неработоспособной машины. Однако слишком низкая величина тайм-аута тоже нежелательна, потому что в сильно загруженной сети пакеты могут доставляться с задержкой, которая приведет к непреднамеренному прерыванию сеансов.

В приложениях, создающих более сложные эфемерные состояния, лучше использовать большую продолжительность тайм-аута из-за более высоких затрат ресурсов на восстановление. Иногда приложение удается спроектировать таким образом, что оно может перезапуститься во время тайм-аута и избежать прерывания сеанса. Каждому сеансу сервер назначает уникальный идентификатор и пароль; если передать эти данные ZooKeeper при подключении, сеанс может быть восстановлен (при условии, что его срок действия не истек). Таким образом, приложение может обеспечить корректное завершение: идентификатор и пароль сеанса сохраняются перед перезапуском процесса, а затем загружаются в перезапущенном процессе с возобновлением сеанса.

Эту функцию следует рассматривать как оптимизацию, предотвращающую прерывание сеансов. Она не избавляет от необходимости обработки прерывания сеанса, которое может произойти при неожиданном сбое машины или даже в том случае, если приложение завершилось корректно, но не перезапустилось до истечения срока действия сеанса (независимо от причины).

Как правило, чем крупнее ансамбль ZooKeeper, тем больше должен быть тайм-аут сеанса. Тайм-ауты подключения, тайм-ауты чтения и частота периодических

сигналов — все эти параметры определяются во внутренней реализации как функция количества серверов в ансамбле, поэтому с ростом ансамбля эти интервалы уменьшаются. Если вы сталкиваетесь с частыми потерями подключений, рассмотрите возможность увеличения тайм-аута. Для отслеживания метрик ZooKeeper (например, статистики задержки запросов) можно использовать JMX.

Состояния

На протяжении своего жизненного цикла объект `ZooKeeper` переходит между различными состояниями (рис. 14.3). Вы можете в любой момент запросить его состояние методом `getState()`:

```
public States getState()
```

Перечисление `States` представляет различные состояния, в которых может находиться объект `ZooKeeper`. (В любой момент времени экземпляр `ZooKeeper` может находиться только в одном состоянии.)

Только что сконструированный экземпляр `ZooKeeper`, находящийся в процессе установления связи с сервисом `ZooKeeper`, находится в состоянии `CONNECTING`. После того, как связь будет установлена, он переходит в состояние `CONNECTED`.

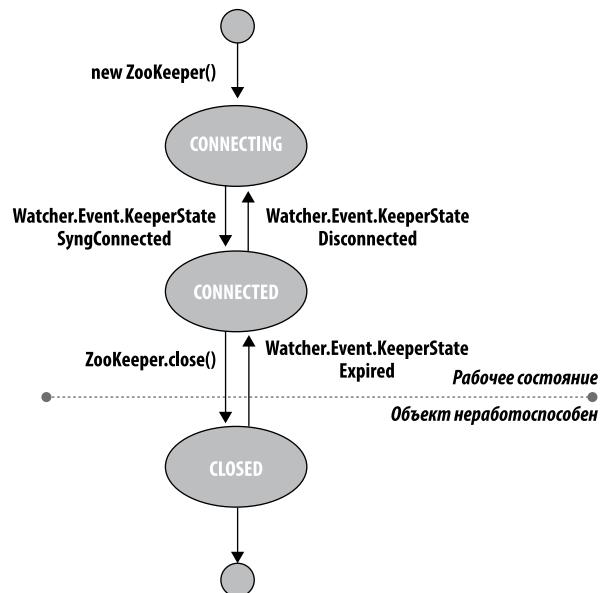


Рис. 14.3. Переходы между состояниями ZooKeeper

Клиент, использующий объект ZooKeeper, может получать оповещения о переходах между состояниями, зарегистрировав объект сторожа `Watcher`. При входе в состояние `CONNECTED` сторож получает событие `WatchedEvent`, у которого свойство `KeeperState` содержит значение `SyncConnected`.



Объект ZooKeeper `Watcher` выполняет две функции: он может использоваться для получения оповещений как об изменениях состояния ZooKeeper (как описано в этом разделе), так и об изменениях в состоянии z-узлов (см. «Сторож», с. 621). Сторож (по умолчанию), передаваемый конструктору ZooKeeper, используется для изменений состояния; для изменений в z-узлах требуется либо специальный экземпляр `Watcher` (передаваемый операции чтения), либо совместное использование сторожа по умолчанию (для формы операции чтения с флагом использования сторожа).

Экземпляр ZooKeeper может подключаться к сервису ZooKeeper и отключаться от него, перемещаясь между состояниями `CONNECTED` и `CONNECTING`. При отключении сторож получает событие `Disconnected`. Эти переходы между состояниями инициируются самим экземпляром ZooKeeper, который пытается автоматически восстановить подключение в случае его потери.

Экземпляр ZooKeeper также может перейти в третье состояние `CLOSED`, если был вызван метод `close()` или произошел тайм-аут сеанса, обозначаемый `KeeperState` с типом `Expired`. В состоянии `CLOSED` объект ZooKeeper не считается работоспособным (работоспособность проверяется вызовом метода `isAlive()` для `States`) и не может использоваться повторно. Чтобы снова подключиться к сервису ZooKeeper, клиент должен сконструировать новый экземпляр ZooKeeper.

Построение приложений с использованием ZooKeeper

Итак, мы довольно подробно рассмотрели ZooKeeper. Давайте посмотрим, как использовать эту систему для написания полезных приложений.

Конфигурация

Один из основных видов сервиса, необходимых распределенному приложению, — сервис управления конфигурацией, обеспечивающий совместное использование конфигурационных данных всеми машинами кластера. На простейшем уровне

ZooKeeper может действовать как хранилище конфигурационных данных с высокой доступностью, которое позволяет приложениям-участникам получать или обновлять конфигурационные файлы. С помощью сторожей ZooKeeper можно реализовать активный сервис конфигурации, в котором заинтересованные клиенты оповещаются об изменениях конфигурации.

В этом разделе мы напишем такой сервис. Мы сделаем пару допущений, упрощающих реализацию (их можно снять ценой небольшой дополнительной работы). Во-первых, сохраняться будут только строковые значения, а ключами будут пути к z-узлам. Во-вторых, в любой момент времени обновления выполняются только одним клиентом. В частности, эта модель соответствует концепции главного узла (такого, как узел имен в HDFS), который обновляет информацию, используемую его подчиненными узлами.

Код будет упакован в класс с именем `ActiveKeyValueStore`:

```
public class ActiveKeyValueStore extends ConnectionWatcher {  
  
    private static final Charset CHARSET = Charset.forName("UTF-8");  
  
    public void write(String path, String value) throws InterruptedException,  
        KeeperException {  
        Stat stat = zk.exists(path, false);  
        if (stat == null) {  
            zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,  
                CreateMode.PERSISTENT);  
        } else {  
            zk.setData(path, value.getBytes(CHARSET), -1);  
        }  
    }  
}
```

Контракт метода `write()` подразумевает запись ключа с заданным значением в ZooKeeper. Он скрывает различия между созданием нового и обновлением существующего z-узла; класс сначала проверяет существование z-узла операцией `exists`, а затем выполняет соответствующую операцию. Еще одна подробность, о которой стоит упомянуть, — необходимость преобразования строкового значения в массив байтов, для которого мы просто используем метод `getBytes()` с кодировкой UTF-8.

Чтобы продемонстрировать использование класса `ActiveKeyValueStore`, возьмем класс `ConfigUpdater`, который обновляет свойство конфигурации. Код этого класса приведен в листинге 14.6.

Листинг 14.6. Приложение, обновляющее свойство из хранилища ZooKeeper в случайные моменты времени

```
public class ConfigUpdater {  
  
    public static final String PATH = "/config";  
  
    private ActiveKeyValueStore store;  
    private Random random = new Random();  
  
    public ConfigUpdater(String hosts) throws IOException, InterruptedException {  
        store = new ActiveKeyValueStore();  
        store.connect(hosts);  
    }  
  
    public void run() throws InterruptedException, KeeperException {  
        while (true) {  
            String value = random.nextInt(100) + " ";  
            store.write(PATH, value);  
            System.out.printf("Set %s to %s\n", PATH, value);  
            TimeUnit.SECONDS.sleep(random.nextInt(10));  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        ConfigUpdater configUpdater = new ConfigUpdater(args[0]);  
        configUpdater.run();  
    }  
}
```

Код достаточно прост. `ConfigUpdater` содержит экземпляр `ActiveKeyValueStore`, который подключается к ZooKeeper в конструкторе `ConfigUpdater`. Метод `run()` выполняется в бесконечном цикле, обновляя z-узел `/config` случайным значением в случайный момент времени.

Теперь посмотрим, как происходит чтение свойства конфигурации `/config`. Сначала в `ActiveKeyValueStore` добавляется метод `read()`:

```
public String read(String path, Watcher watcher) throws InterruptedException,  
    KeeperException {  
    byte[] data = zk.getData(path, watcher, null/*stat*/);  
    return new String(data, CHARSET);  
}
```

Метод `getData()` получает путь, объект `Watcher` и объект `Stat`. Объект `Stat` заполняется методом `getData()` и используется для возвращения информации

вызывающей стороне. Таким образом вызывающая сторона может получить и данные, и метаданные z-узла, хотя в нашем случае вместо `Stat` передается `null`, так как метаданные нас не интересуют.

Потребитель сервиса `ConfigWatcher` (листинг 14.7) создает `ActiveKeyValueStore`, и после запуска вызывает метод `read()` хранилища (в своем методе `displayConfig()`) для передачи ссылки на самого себя как сторожа. Метод выводит исходное значение прочитанного свойства конфигурации.

Листинг 14.7. Приложение для отслеживания обновлений свойства в ZooKeeper и вывода их на консоль

```
public class ConfigWatcher implements Watcher {  
  
    private ActiveKeyValueStore store;  
  
    public ConfigWatcher(String hosts) throws IOException, InterruptedException {  
        store = new ActiveKeyValueStore();  
        store.connect(hosts);  
    }  
  
    public void displayConfig() throws InterruptedException, KeeperException {  
        String value = store.read(ConfigUpdater.PATH, this);  
        System.out.printf("Read %s as %s\n", ConfigUpdater.PATH, value);  
    }  
  
    @Override  
    public void process(WatchedEvent event) {  
        if (event.getType() == EventType.NodeDataChanged) {  
            try {  
                displayConfig();  
            } catch (InterruptedException e) {  
                System.err.println("Interrupted. Exiting.");  
                Thread.currentThread().interrupt();  
            } catch (KeeperException e) {  
                System.err.printf("KeeperException: %s. Exiting.\n", e);  
            }  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        ConfigWatcher configWatcher = new ConfigWatcher(args[0]);  
        configWatcher.displayConfig();  
  
        // Продолжать существование до уничтожения процесса  
    }  
}
```

```
// или прерывания программного потока  
Thread.sleep(Long.MAX_VALUE);  
}  
}
```

Когда `ConfigUpdater` обновляет узел, ZooKeeper заставляет сторожа инициировать событие типа `EventType.NodeDataChanged`. `ConfigWatcher` реагирует на это событие в своем методе `process()` чтением и выводом обновленной версии свойства.

Так как сторожа являются одноразовыми сигналами, мы сообщаем ZooKeeper о новом стороже при каждом вызове `read()` для `ActiveKeyValueStore`; это гарантирует, что мы увидим будущие обновления. Более того, не гарантировано даже получение каждого обновления, потому что между получением события сторожа и следующим чтением z-узел мог обновиться (возможно, неоднократно), и так как в этот период у клиента не было зарегистрированного сторожа, оповещаться он не будет. Для сервиса конфигурации это не проблема, потому что клиента интересует только последнее значение свойства, заменяющее все предыдущие значения. Тем не менее в общем случае следует помнить об этом потенциальном ограничении.

Посмотрим, как работает этот код. Запустите `ConfigUpdater` в одном терминальном окне:

```
% java ConfigUpdater localhost  
Set /config to 79  
Set /config to 14  
Set /config to 78
```

Затем немедленно запустите `ConfigWatcher` в другом окне:

```
% java ConfigWatcher localhost  
Read /config as 79  
Read /config as 14  
Read /config as 78
```

Отказоустойчивое приложение ZooKeeper

Первое из Заблуждений Распределенного Программирования гласит: «Сеть работает надежно»¹. Программы, рассмотренные нами до настоящего момента, предполагали надежную сеть, и при запуске в реальной сети в них могут произойти разные сбои. Рассмотрим несколько разновидностей сбоев и меры по их исправлению, чтобы наши программы сохраняли работоспособность.

¹ См. http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.

У каждой операции ZooKeeper в Java API в секции `throws` объявляются два типа исключений: `InterruptedException` и `KeeperException`.

InterruptedException

Исключение `InterruptedException` инициируется в случае прерывания операции. В Java существует стандартный механизм отмены блокирующих методов, который заключается в вызове `interrupt()` для потока, из которого был вызван блокирующий метод. Успешная отмена приводит к выдаче `InterruptedException`. ZooKeeper поддерживает этот стандарт, поэтому операции ZooKeeper можно отменять подобным образом. Классы и библиотеки, использующие ZooKeeper, обычно должны передавать (`propagate`) `InterruptedException`, чтобы их клиенты могли отменять операции¹.

Исключение `InterruptedException` сообщает не об ошибке, а об отмене операции, поэтому в примере с управлением конфигурацией будет уместна его передача, приводящая к завершению приложения.

KeeperException

Исключение `KeeperException` инициируется, когда сервер ZooKeeper сигнализирует об ошибке, или при возникновении проблем при взаимодействии с сервером. Для разных видов ошибок создаются разные субклассы `KeeperException` — как, например, `KeeperException.NoNodeException`. Исключение этого класса инициируется при попытке выполнения операции с несуществующим z-узлом.

У каждого субкласса `KeeperException` имеется соответствующий код с информацией о типе ошибки. Например, исключению `KeeperException.NoNodeException` соответствует код `KeeperException.Code.NONODE` (значение перечисления).

Существует два способа обработки `KeeperException`: либо перехват `KeeperException` и проверка кода для определения действий по исправлению, либо перехват эквивалентных субклассов `KeeperException` и выполнение соответствующего действия в каждом блоке `catch`.

Исключения `KeeperException` делятся на три категории.

Исключения состояния

Исключение состояния происходит при невозможности применения операции к дереву z-узлов. Исключения состояния обычно происходят из-за одновременного

¹ За подробностями обращайтесь к превосходной статье Брайана Гетца (Brian Goetz) «Dealing with `InterruptedException`».

изменения z-узла другим процессом. Например, попытка выполнения операции `setData` с номером версии завершится исключением `KeeperException.BadVersionException`, если z-узел был обновлен другим процессом, так как номера версий не совпадут. Программист обычно знает о возможности таких конфликтов и программирует их обработку.

Некоторые исключения состояния свидетельствуют об ошибке в программе — как, например, исключение `KeeperException.NoChildrenForEphemeralsException`, инициируемое при попытке создания дочернего z-узла у эфемерного z-узла.

Восстанавливаемые исключения

К категории восстанавливаемых относятся исключения, после которых приложение может восстановить работу в том же сеансе ZooKeeper. Признаком восстанавливаемого исключения является класс `KeeperException.ConnectionLossException`. ZooKeeper пытается восстановить подключение, в большинстве случаев попытка завершается удачей, а сеанс остается неповрежденным.

Однако ZooKeeper не может определить, была ли применена операция, завершившаяся выдачей `KeeperException.ConnectionLossException`. Это пример частичного сбоя (о которых говорилось в начале главы). Программист должен обработать эту неопределенность, а предпринимаемые действия зависят от приложения.

Здесь стоит провести черту между *идемпотентными* и *неидемпотентными* операциями. Идемпотентной называется операция, которая может быть многократно применена с одним результатом — как, например, запрос на чтение или безусловная команда `setData`. Такие операции можно просто повторить.

Неидемпотентную операцию нельзя повторить безусловно, так как эффект от ее многократного применения не идентичен эффекту от однократного применения. Программа должна определить, было ли применено обновление; для этого она декодирует информацию из пути или данных z-узла. О том, как обрабатывать сбои неидемпотентных операций, будет рассказано в разделе «Восстанавливаемые исключения» на с. 642, когда мы будем рассматривать реализацию сервиса блокировки.

Невосстанавливаемые исключения

В некоторых случаях сеанс ZooKeeper становится недействительным — например, из-за тайм-аута или из-за того, что сеанс был закрыт (в обоих случаях выдается исключение `KeeperException.SessionExpiredException`), или из-за неудачной аутентификации (`KeeperException.AuthFailedException`). В любом случае все эфемерные узлы, связанные с сеансом, будут потеряны, поэтому приложение перед повторным подключением к ZooKeeper должно заново построить свое состояние.

Надежный сервис конфигурации

Возвращаясь к методу `write()` в `ActiveKeyValueStore`, вспомним, что он состоит из операции `exists`, за которой следует операция `create` или `setData`:

```
public void write(String path, String value) throws InterruptedException,
    KeeperException {
    Stat stat = zk.exists(path, false);
    if (stat == null) {
        zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT);
    } else {
        zk.setData(path, value.getBytes(CHARSET), -1);
    }
}
```

В целом метод `write()` является идемпотентным, поэтому мы можем безусловно повторить попытку его выполнения. Ниже приведена обновленная версия метода `write()` с циклом повторных попыток. Метод пытается выполнить максимальное количество попыток (`MAX_RETRIES`) с задержкой `RETRY_PERIOD_SECONDS` между попытками:

```
public void write(String path, String value) throws InterruptedException,
    KeeperException {
    int retries = 0;
    while (true) {
        try {
            Stat stat = zk.exists(path, false);
            if (stat == null) {
                zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT);
            } else {
                zk.setData(path, value.getBytes(CHARSET), stat.getVersion());
            }
            return;
        } catch (KeeperException.SessionExpiredException e) {
            throw e;
        } catch (KeeperException e) {
            if (retries++ == MAX_RETRIES) {
                throw e;
            }
            // Задержка, затем повторная попытка
            TimeUnit.SECONDS.sleep(RETRY_PERIOD_SECONDS);
        }
    }
}
```

Мы принимаем особые меры, чтобы избежать повторной попытки с `KeeperException.SessionExpiredException`, потому что при истечении срока действия сеанса объект `ZooKeeper` переходит в состояние `CLOSED`, из которого он уже не сможет подключиться повторно (см. рис. 14.3). Мы просто перезапускаем исключение и позволяем вызывающей стороне создать новый экземпляр `ZooKeeper` для повторной попытки выполнения всего метода `write()`. Простой способ создания нового экземпляра заключается в создании нового объекта `ConfigUpdater` (переименованного в `ResilientConfigUpdater`) для восстановления из прерванного сеанса:

```
public static void main(String[] args) throws Exception {
    while (true) {
        try {
            ResilientConfigUpdater configUpdater =
                new ResilientConfigUpdater(args[0]);
            configUpdater.run();
        } catch (KeeperException.SessionExpiredException e) {
            // Начало нового сеанса
        } catch (KeeperException e) {
            // Повторная попытка уже сделана, завершить работу
            e.printStackTrace();
            break;
        }
    }
}
```

Это всего лишь одна из возможных стратегий реализации повторных попыток. Существуют и другие — например, экспоненциальная задержка, при которой промежуток между повторными попытками каждый раз умножается на константу. Пакет `org.apache.hadoop.io.retry` из Hadoop Core содержит набор средств для включения готовой логики повторных попыток в ваш код; он пригодится вам при построении приложений ZooKeeper.

Блокировка

Распределенная блокировка представляет собой механизм взаимного исключения доступа в группе процессов. В любой конкретный момент блокировку может удерживать только один процесс. Распределенные блокировки могут использоваться в больших распределенных системах для выбора ведущего, которым становится процесс, удерживающий блокировку в данный момент.



Не путайте выборы ведущего ZooKeeper с общим механизмом выбора ведущего, который может быть построен с использованием примитивов ZooKeeper (одна такая реализация поставляется с ZooKeeper). Механизм выбора ведущего ZooKeeper остается скрытым — в отличие от обобщенно-го сервиса выбора ведущего, описанного в этом разделе и предназначенно-го для использования распределенными системами, которым нужно согла-совать главный процесс.

Чтобы реализовать распределенную блокировку на основе ZooKeeper, мы используем последовательные z-узлы для упорядочения конкурирующих процессов. Идея проста: сначала назначается z-узел блокировки, который обычно описывает некую сущность (допустим, */leader*); клиенты, которые желают получить блокировку, создают последовательные эфемерные z-узлы, дочерние по отношению к z-узлу блокировки. В любой момент времени блокировку удерживает клиент с наименьшим последовательным номером. Например, если два клиента приблизительно одновременно создают z-узлы */leader/lock-1* и */leader/lock-2*, то блокировка будет принадлежать клиенту, создавшему */leader/lock-1*, поскольку его z-узел имеет меньший номер. Роль арбитра играет сервис ZooKeeper, назначающий номера последовательных z-узлов.

Блокировка снимается простым удалением z-узла */leader/lock-1*; также в случае прекращения работы клиента z-узел будет уничтожен как эфемерный. Блокировка в этом случае переходит к клиенту, создавшему */leader/lock-2*, потому что этот номер стал наименьшим. Клиент оповещается о получении блокировки созданием сторожа, срабатывающего при уничтожении z-узлов.

Псевдокод получения блокировки выглядит следующим образом:

1. Создать эфемерный последовательный z-узел с именем *lock-*, являющийся дочерним по отношению к z-узлу блокировки, и запомнить его фактический путь (возвращаемое значение операции *create*).
2. Получить дочерние узлы z-узла блокировки и установить сторожа.
3. Если путь z-узла, созданного на шаге 1, имеет наименьший номер из дочер-них z-узлов, полученных на шаге 2, блокировка получена (выход).
4. Ожидать оповещения от сторожа, установленного на шаге 2, и перейти к шагу 2.

Восстанавливаемые исключения

У алгоритма блокировки в его текущем виде есть один недостаток: он не обра-батывает случай сбоя операции *create* из-за потери подключения. Вспомните, что в этом случае мы не знаем, успешно ли была выполнена операция. Создание

последовательного z-узла является неидемпотентной операцией, поэтому простая повторная попытка не подходит — если операция создания завершилась успешно, появляется «бесхозный» z-узел, который не будет удален (по крайней мере, до завершения клиентского сеанса).

Проблема в том, что после повторного подключения клиент не может сказать, создал ли он какие-либо дочерние z-узлы. Если встроить в имя z-узла идентификатор, при потере подключения клиент сможет проверить, присутствует ли идентификатор в именах каких-либо дочерних z-узлов z-узла блокировки. Если идентификатор присутствует, значит, операция `create` прошла успешно и создавать новый дочерний z-узел не нужно. Если ни у одного дочернего z-узла в имени идентификатор не встречается, клиент может создать новый последовательный дочерний z-узел.

Клиентский идентификатор сеанса представляет собой длинное целое число, уникальное для ZooKeeper, а следовательно, идеально подходящее для идентификации клиента между событиями потери подключения. Идентификатор сеанса может быть получен вызовом метода `getSessionId()` класса Java `ZooKeeper`.

Невосстанавливаемые исключения

По истечении клиентского сеанса ZooKeeper эфемерный z-узел, созданный клиентом будет удален, что фактически приведет к потере блокировки или, по крайней мере, пропуску очереди клиента на ее получение. Приложение, использующее блокировку, должно понять, что блокировка потеряна, сбросить состояние и начать заново с создания нового объекта блокировки и попытки его получения. Обратите внимание: этим процессом управляет приложение, а не реализация блокировки, так как последняя не знает, как именно приложение должно выполнять сброс своего состояния.

Реализация

Правильная реализация распределенной блокировки — нетривиальная задача, требующая учета всех возможных сбоев. В поставку ZooKeeper включена качественная реализация блокировки на Java `WriteLock`, чрезвычайно простая в использовании клиентами.

Другие распределенные структуры данных и протоколы

На основе ZooKeeper могут быть реализованы и другие распределенные структуры данных и протоколы — барьеры, очереди и двухфазные закрепления. Интересно,

что все эти протоколы являются синхронными, хотя для их построения используются асинхронные примитивы ZooKeeper (такие, как оповещения).

На веб-сайте ZooKeeper приведен псевдокод реализации некоторых структур данных и протоколов. ZooKeeper поставляется с реализациями некоторых стандартных решений (включая блокировки, выбор ведущего и очереди); они находятся в каталоге *recipes* дистрибутива.

Проект Curator (<https://github.com/Netflix/curator>) также предоставляет обширный набор готовых решений ZooKeeper.

Практическое использование ZooKeeper

В условиях реальной эксплуатации ZooKeeper работает в реплицированном режиме. В этом разделе рассматриваются некоторые аспекты управления ансамблем серверов ZooKeeper. За более подробной информацией (поддерживаемые платформы, рекомендуемое оборудование, процедуры сопровождения, свойства конфигурации и т. д.) обращайтесь к руководству администратора ZooKeeper.

Надежность и производительность

Машины ZooKeeper должны размещаться таким образом, чтобы свести к минимуму последствия возможных сбоев машин и сети. На практике это означает, что серверы должны быть распределены между стойками, источниками питания и коммутаторами, чтобы сбой оборудования не привел к выходу из строя большинства серверов ансамбля.

Для приложений, требующих минимальной задержки (порядка миллисекунд), очень важно, чтобы все серверы работали в ансамбле в одном центре обработки данных. Однако некоторые сценарии использования не требуют минимальной задержки, что позволяет распределить серверы по нескольким центрам обработки данных (не менее двух на каждый центр) для обеспечения дополнительной устойчивости. Примеры такого рода — выборы ведущего и распределенная блокировка с низкой гранулярностью; в обоих случаях состояние изменяется относительно редко, так что дополнительные затраты в несколько десятков миллисекунд, обусловленные взаимодействием центров обработки данных, незначительны по сравнению с общим функционированием службы.

ZooKeeper — система с высокой доступностью, поэтому очень важно, чтобы ее операции выполнялись своевременно. По этой причине серверы ZooKeeper должны

работать на выделенных машинах. Наличие других приложений, конкурирующих за ресурсы, может существенно ухудшить быстродействие ZooKeeper.



В ZooKeeper существует концепция наблюдателя — своего рода ведомого без права голоса. Так как наблюдатели не участвуют в голосовании за консенсус во время запросов записи, их присутствие улучшает производительность чтения в кластере ZooKeeper без ущерба для производительности записи. Голосующие серверы размещаются в одном центре обработки данных, а наблюдатели — в другом.

Настройте ZooKeeper так, чтобы журнал транзакций и снимки состояния хранились на разных дисках. По умолчанию они хранятся в одном каталоге, заданном свойством `dataDir`, но, если изменить свойство `dataLogDir`, журнал транзакций будет записываться туда. Выделение серверу ZooKeeper отдельного устройства (не просто раздела) позволяет добиться максимальной скорости записи журнала на диск, которая выполняется последовательно без позиционирования. Так как все операции записи выполняются через ведущего, скорость записи не масштабируется добавлением новых серверов.

Выгрузка процесса на диск также отрицательно влияет на производительность. Чтобы избежать ее, следует задать размер кучи Java меньшим объема неиспользуемой физической памяти на машине. Файл `java.env` в каталоге конфигурации может использоваться для задания переменной окружения `JVMFLAGS`, определяющей размер кучи (а также других аргументов JVM).

Конфигурация

Каждому серверу ансамбля ZooKeeper присваивается числовой идентификатор из диапазона от 1 до 255, уникальный в пределах ансамбля. Номер сервера задается в виде простого текста в файле `myid` в каталоге, заданном свойством `dataDir`.

Впрочем, присваивание номеров серверам — лишь половина дела. Каждому серверу также необходимо сообщить идентификаторы других серверов ансамбля и их местонахождение в сети. Файл конфигурации ZooKeeper должен включать в себя строку для каждого сервера в формате

```
server.n=xhost:port:port
```

Значение `n` заменяется номером сервера. В строке задаются два порта: первый используется ведомыми для соединения с ведущим, а второй — для выбора ведущего. Пример конфигурации ансамбля ZooKeeper с репликацией на трех машинах:

```
tickTime=2000
dataDir=/disk1/zookeeper
dataLogDir=/disk2/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zookeeper1:2888:3888
server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
```

Серверы ведут прослушивание на трех портах: 2181 для клиентских подключений; 2888 для подключений ведомых, если данный сервер является ведущим; 3888 для других подключений серверов в фазе выбора ведущего. При запуске сервер ZooKeeper читает файл *myid*, а затем из файла конфигурации определяет порты, на которых он должен вести прослушивание, и сетевые адреса других серверов ансамбля.

Клиенты, подключающиеся к ансамблю ZooKeeper, должны указывать строку *zookeeper1:2181*, *zookeeper2:2181* и *zookeeper3:2181* в качестве строки хоста в конструкторе объекта ZooKeeper.

В реплицированном режиме существует два дополнительных обязательных свойства: *initLimit* и *syncLimit*. Оба значения задаются в единицах *tickTime*.

Свойство *initLimit* определяет время, выделяемое ведомым для подключения и синхронизации с ведущим. Если большинство ведомых не успевает синхронизироваться в течение этого времени, ведущий отказывается от своей роли, и его место занимает другой ведущий. Если это происходит слишком часто, значит, заданное значение слишком мало.

Свойство *syncLimit* определяет время, выделяемое ведомому для синхронизации с ведущим. Если ведомый не успевает синхронизироваться за этот период, он перезапускается. Клиенты, подключившиеся к этому ведомому, подключаются к другому.

Таковы минимальные настройки, необходимые для запуска кластера серверов ZooKeeper. Однако существуют и другие свойства конфигурации (прежде всего, направленные на оптимизацию производительности), документированные в руководстве администратора ZooKeeper.

15 Sqoop

Аарон Кимболл

Огромным достоинством платформы Hadoop является возможность работы с данными в нескольких разных форматах. HDFS обеспечивает надежное хранение журналов и других данных из множества разных источников, а программы MapReduce могут разбирать самые разнообразные форматы, извлекать нужную информацию и объединять наборы данных с формированием нужных результатов.

Но чтобы взаимодействовать с данными в репозиториях за пределами HDFS, программы MapReduce должны использовать внешние API для получения доступа к данным. Часто полезная информация в организации хранится в структурированных источниках — например, в реляционных системах управления базами данных (РСУБД). Apache Sqoop — программа с открытым кодом, позволяющая своим пользователям извлекать данные из структурированных источников в Hadoop для дальнейшей обработки. Обработка может выполняться программами MapReduce или другими инструментами более высокого уровня — такими, как Hive. (Sqoop можно использовать даже для перемещения информации из базы данных в HBase.) А когда появятся окончательные результаты аналитического конвейера, Sqoop может экспортить их обратно в хранилище данных для потребления другими клиентами.

В этой главе вы узнаете, как работает Sqoop и как эта система используется в конвейерах обработки данных.

Установка и запуск Sqoop

Sqoop можно загрузить в нескольких местах. Домашняя страница проекта находится по адресу <http://sqoop.apache.org/>. Здесь хранится весь исходный код и документация Sqoop.

На сайте доступны как официальные выпуски, так и исходный код версии, находящейся в разработке. Репозиторий содержит инструкции по компилированию проекта. Кроме того, дистрибутив CDH (Cloudera's Distribution Including Apache Hadoop) содержит пакет установки Sqoop вместе с совместимыми выпусками Hadoop и других инструментов (таких, как Hive).

Если вы загрузили версию Apache, она будет помещена в каталог вида */home/your-name/sqoop-x.y.z/*. Мы будем обозначать его *\$\$SQOOP_HOME*. Чтобы запустить Sqoop, следует выполнить сценарий *\$\$SQOOP_HOME/bin/sqoop*.

Если вы установили версию от Cloudera, пакет разместит сценарии Sqoop в стандартные каталоги (например, */usr/bin/sqoop*). Чтобы запустить Sqoop, введите команду *sqoop* в командной строке. (Независимо от того, как именно вы установили Sqoop, в дальнейшем этот сценарий мы будем обозначать просто *sqoop*.)

Запуск Sqoop без аргументов не делает ничего особенно интересного:

```
% sqoop
Try sqoop help for usage.
```

Sqoop представляет собой набор инструментов или команд. Если не выбрать команду, Sqoop попросту не знает, что нужно делать. *help* — одна из команд, которая выводит список доступных инструментов:

```
% sqoop help
usage: sqoop COMMAND [ARGS]
Available commands:
  codegen           Generate code to interact with database records
  create-hive-table Import a table definition into Hive
  eval              Evaluate a SQL statement and display the results
...
See 'sqoop help COMMAND' for information on a specific command.
```

При передаче имени команды в аргументе Sqoop выдает инструкции по использованию данной команды:

```
% sqoop help import
usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]
Common arguments:
  --connect <jdbc-uri>      Specify JDBC connect string
  --driver <class-name>     Manually specify JDBC driver class to use
  --hadoop-home <dir>       Override $HADOOP_HOME
  --help                  Print usage instructions
  -P                      Read password from console
  --password <password>    Set authentication password
  --username <username>    Set authentication username
  --verbose                Print more information while working
...
...
```

Также для запуска команд и инструментов Sqoop можно использовать специализированные сценарии. Имя сценария строится по схеме `sqoop-команда` — например, `sqoop-help`, `sqoop-import` и т. д. Приведенные команды эквивалентны командам `sqoop help` или `sqoop import`.

Коннекторы Sqoop

Инфраструктура расширения Sqoop позволяет импортировать (и экспортировать) данные из любой внешней системы хранения данных, поддерживающей массовую передачу данных. *Коннектор Sqoop* представляет собой модульный компонент, который использует эту инфраструктуру для выполнения операций импорта и экспорта. В поставку Sqoop включены коннекторы для многих популярных реляционных баз данных, включая MySQL, PostgreSQL, Oracle, SQL Server и DB2. Также имеется обобщенный коннектор JDBC для подключения к любой базе данных, поддерживающей протокол Java JDBC. Sqoop предоставляет оптимизированные коннекторы MySQL и PostgreSQL, использующие специализированные API для эффективного выполнения массовой пересылки данных (эта возможность более подробно рассматривается в разделе «Прямое импортирование», с. 657).

Помимо встроенных коннекторов Sqoop, существуют различные сторонние коннекторы для разных хранилищ данных, от корпоративного уровня (включая Netezza, Teradata и Oracle) до хранилищ NoSQL (таких, как Couchbase). Эти контейнеры необходимо загрузить отдельно и включить в существующую установку Sqoop по инструкциям, прилагаемым к коннектору.

Пример импортирования

После завершения установки Sqoop можно использовать для импортирования данных в Hadoop. В примерах этой главы будет использоваться MySQL — эта РСУБД проста в использовании и доступна для многих платформ.

Инструкции по установке и настройке MySQL приведены в документации по адресу <http://dev.mysql.com/doc/refman/5.1/en/>. Пользователи систем Linux на базе Debian (например, Ubuntu) могут ввести команду `sudo apt-get install mysql-client mysql-server`, а пользователи RedHat — команду `sudo yum install mysql mysql-server`.

После установки MySQL следует войти в систему и затем создать базу данных (листинг 15.1).

Листинг 15.1. Создание новой схемы базы данных MySQL

```
% mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 349
Server version: 5.1.37-1ubuntu5.4 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> CREATE DATABASE hadoopguide;
Query OK, 1 row affected (0.02 sec)

mysql> GRANT ALL PRIVILEGES ON hadoopguide.* TO '%'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON hadoopguide.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> quit;
Bye
```

В приглашении `Enter password:` следует ввести пароль привилегированного пользователя MySQL. Скорее всего, он совпадает с паролем системного привилегированного пользователя. Если вы работаете в Ubuntu или другой разновидности Linux, в которой прямой вход привилегированного пользователя невозможен, введите пароль, выбранный во время установки MySQL.

В этом сеансе мы создали новую схему базы данных с именем `hadoopguide`, которая будет использоваться в этой главе. Затем мы разрешаем любому локальному пользователю просматривать и изменять содержимое схемы `hadoopguide` и закрываем сеанс¹.

Снова войдите в базу данных (уже под своими учетными данными) и создайте таблицу для импортирования в HDFS (листинг 15.2).

Листинг 15.2. Заполнение базы данных

```
% mysql hadoopguide
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 352
```

¹ Конечно, при реальной эксплуатации к управлению доступом следует отнестись намного серьезнее, но этот пример приведен лишь для демонстрационных целей.

```
Server version: 5.1.37-1ubuntu5.4 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE TABLE widgets(id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
-> widget_name VARCHAR(64) NOT NULL,
-> price DECIMAL(10,2),
-> design_date DATE,
-> version INT,
-> design_comment VARCHAR(100));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO widgets VALUES (NULL, 'sprocket', 0.25, '2010-02-10',
-> 1, 'Connects two gizmos');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO widgets VALUES (NULL, 'gizmo', 4.00, '2009-11-30', 4,
-> NULL);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO widgets VALUES (NULL, 'gadget', 99.99, '1983-08-13',
-> 13, 'Our flagship product');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> quit;
```

В этом листинге создается новая таблица с именем `widgets`. Эта фиктивная таблица продуктов будет использоваться в дальнейших примерах этой главы. Таблица `widgets` содержит несколько полей, представляющих различные типы данных.

Используем Sqoop для импортирования этой таблицы в HDFS:

```
% sqoop import --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets -m 1
10/06/23 14:44:18 INFO tool.CodeGenTool: Beginning code generation
...
10/06/23 14:44:20 INFO mapred.JobClient: Running job: job_201006231439_0002
10/06/23 14:44:21 INFO mapred.JobClient: map 0% reduce 0%
10/06/23 14:44:32 INFO mapred.JobClient: map 100% reduce 0%
10/06/23 14:44:34 INFO mapred.JobClient: Job complete:
job_201006231439_0002
...
10/06/23 14:44:34 INFO mapreduce.ImportJobBase: Retrieved 3 records.
```

Команда Sqoop `import` запускает задание MapReduce, которое подключается к базе данных MySQL и читает таблицу. По умолчанию для ускорения процесса

импортирования используются четыре параллельных задачи отображения. Каждая задача записывает свои импортированные результаты в отдельный файл, но при этом все файлы находятся в одном каталоге. Так как в этом примере импортируются всего три строки данных, мы приказываем Sqoop использовать одну задачу отображения (`-m 1`), чтобы получить один файл в HDFS.

Содержимое файла можно просмотреть:

```
% hadoop fs -cat widgets/part-m-00000
1,sprocket,0.25,2010-02-10,1,Connects two gizmos
2,gizmo,4.00,2009-11-30,4,null
3,gadget,99.99,1983-08-13,13,Our flagship product
```



Строка подключения (`jdbc:mysql://localhost/hadoopguide`) из этого примера выполнит чтение из базы данных на локальной машине. Если вы используете распределенный кластер Hadoop, не включайте в строку подключения `localhost`, так как задачи отображения, не выполняемые на одной машине с базой данных, подключиться не смогут. Даже если Sqoop работает на одной машине с сервером базы данных, следует задать полное имя хоста.

По умолчанию Sqoop генерирует для импортированных данных текстовые файлы, разделенные запятыми. Разделители можно задать явно — как и символы, в которые заключаются поля, и экранирующие (escape) символы для использования символов-разделителей в содержимом полей. Аргументы командной строки, определяющие разделители, форматы файлов, сжатие и параметры управления процессом импортирования, описаны в руководстве пользователя Sqoop, входящем в дистрибутив Sqoop¹, а также в электронной документации (`sqoop help import` или `man sqoop-import` в CDH).

Текстовые и двоичные форматы

Sqoop позволяет импортировать данные в нескольких разных форматах. Текстовые файлы (формат по умолчанию) хорошо читаются, работают независимо от платформы и имеют простейшую структуру. Вместе с тем они не могут содержать двоичные поля (как столбцы базы данных типа `VARBINARY`); также возможны проблемы с различием `null` и строковых полей со значением "null" (хотя параметр `--null-string` позволяет управлять представлением значений `null`).

¹ Доступно на веб-сайте <http://sqoop.apache.org/>.

Для решения этих проблем можно воспользоваться форматом Sqoop на основе **SequenceFile** или Avro. Как файлы данных Avro, так и **SequenceFile** обеспечивают самое точное представление импортированных данных. Они также позволяют сжимать данные с сохранением возможностей MapReduce по параллельной обработке разных секций одного файла. Однако текущие версии Sqoop не могут загружать файлы Avro и **SequenceFile** в Hive (впрочем, файлы данных Avro можно загрузить в Hive вручную). Наконец, у **SequenceFile** есть еще один недостаток — они привязаны к Java, тогда как файлы данных Avro могут создаваться во многих языках.

Сгенерированный код

Кроме записи содержимого таблиц баз данных в HDFS Sqoop также генерирует исходный файл Java (*widgets.java*) и записывает его в текущий локальный каталог. (Чтобы увидеть этот файл после выполнения приведенной ранее команды `sqoop import`, выполните команду `ls widgets.java`.)

Как будет показано в разделе «Подробнее об импортировании» на с. 654, Sqoop может использовать сгенерированный код для десериализации данных таблицы из источника, прежде чем записывать их в HDFS.

Сгенерированный класс (*widgets*) вмещает одну запись из импортированной таблицы. Он может обработать такую запись в MapReduce или сохранить ее в формате **SequenceFile** в HDFS. (Файлы **SequenceFile**, записанные Sqoop в процессе импорта, сохраняют каждую импортированную строку в «значении» пары **SequenceFile** «ключ-значение», используя сгенерированный класс.)

Если вы работаете с записями, импортированными в **SequenceFile**, вам неизбежно придется использовать сгенерированные классы (для десериализации данных из хранилища **SequenceFile**). С записями на базе текстовых файлов можно работать без использования сгенерированного кода, но, как будет показано в разделе «Работа с импортированными данными» на с. 657, сгенерированный код Sqoop способен выполнить некоторые рутинные аспекты обработки данных за вас.

Другие системы сериализации

Новые версии Sqoop также поддерживают генерирование схем и сериализацию на базе Avro (см. «Avro», с. 161), что позволяет использовать Sqoop в проекте без интеграции со сгенерированным кодом.

Подробнее об импортировании

Как упоминалось ранее, для импортирования таблицы из базы данных Sqoop запускает задание MapReduce, которое извлекает строки данных из таблицы и записывает их в HDFS. Как MapReduce читает данные? В этом разделе объясняются некоторые внутренние механизмы Sqoop.

На рис. 15.1 представлена общая схема взаимодействия Sqoop с базой данных и Hadoop. Код Sqoop, как и код Hadoop, написан на Java. Java предоставляет API JDBC (Java Database Connectivity), который позволяет приложениям обращаться к данным, хранящимся в РСУБД, а также получать информацию об этих данных. Многие фирмы-разработчики баз данных предоставляют драйвер JDBC, который реализует JDBC API и содержит необходимый код для подключения к серверу базы данных.



Sqoop пытается выбрать загружаемый драйвер на основании URL-адреса в строке подключения, использованной для подключения к базе данных. Возможно, вам придется отдельно загрузить сам драйвер JDBC и установить его на клиенте Sqoop. В тех случаях, когда Sqoop не может выбрать подходящий драйвер JDBC, пользователь может указать, как именно драйвер JDBC должен загружаться в Sqoop. Эта возможность позволяет Sqoop работать с разнообразными платформами баз данных.

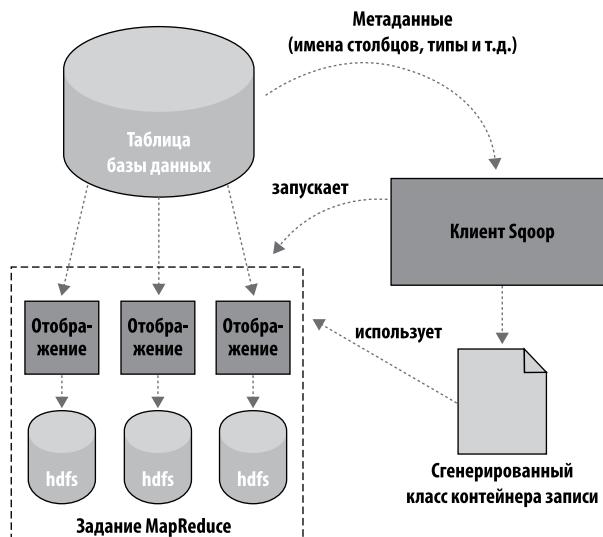


Рис. 15.1. Процесс импортирования в Sqoop

Прежде чем импортировать данные, Sqoop использует JDBC для анализа импортируемой таблицы. Sqoop получает список всех столбцов и их типов данных SQL. Типам SQL (`VARCHAR`, `INTEGER` и т. д.) ставятся в соответствие типы данных Java (`String`, `Integer` и т. д.), в которых будут храниться значения полей в приложениях MapReduce. На основании этой информации генератор кода Sqoop создает класс для хранения записи, извлеченной из таблицы. Так, класс `Widget` из предыдущего примера содержит следующие методы получения чтения столбцов извлеченной записи:

```
public Integer get_id();
public String get_widget_name();
public java.math.BigDecimal get_price();
public java.sql.Date get_design_date();
public Integer get_version();
public String get_design_comment();
```

Впрочем, в системе импортирования более важную роль играют методы сериализации, образующие интерфейс `DBWritable`, через который класс `Widget` взаимодействует с JDBC:

```
public void readFields(ResultSet __dbResults) throws SQLException;
public void write(PreparedStatement __dbStmt) throws SQLException;
```

Интерфейс JDBC `ResultSet` предоставляет курсор для получения записей из запроса; метод `readFields()` заполняет поля объекта `Widget` столбцами одной строки данных `ResultSet`. Метод `write()` используется Sqoop для вставки новых строк `Widget` в таблицу — этот процесс называется *экспортированием* (см. «Подробнее об экспортировании», с. 665).

Задание MapReduce, запущенное Sqoop, использует реализацию `InputFormat`, которая читает части таблицы базы данных через JDBC. Класс `DataDrivenDBInputFormat`, поставляемый с Hadoop, разбивает результаты запроса по нескольким задачам отображения.

Обычно чтение таблицы осуществляется простым запросом следующего вида:

```
SELECT col1,col2,col3,... FROM tableName
```

Но часто производительность импортирования можно улучшить, разделив запрос по нескольким узлам. На основании метаданных таблицы Sqoop выбирает подходящий столбец для разбиения таблицы (обычно первичный ключ, если он существует), определяет минимальное и максимальное значения столбца и использует их в сочетании с целевым количеством задач для определения запросов, которые должны выдаваться каждой задачей отображения.

Допустим, таблица `widgets` содержит 100 000 записей, а столбец `id` содержит значения от 0 до 99 999. При импортировании этой таблицы Sqoop определяет,

что `id` является первичным ключом. При запуске задания MapReduce объект `DataDrivenDBInputFormat`, используемый для импортирования, выдает команду вида `SELECT MIN(id), MAX(id) FROM widgets`. Полученные значения используются для интерполяции по всему диапазону данных. Если предположить, что пять задач отображения будут работать параллельно (параметр `-m 5`), эти задачи будут выполнять запросы вида `SELECT id, widget_name, ... FROM widgets WHERE id >= 0 AND id < 20000`, и т. д.

Выбор столбца разбиения чрезвычайно важен для эффективной организации параллельной обработки. Если значения столбца распределены неравномерно (Например, в диапазоне от 50 000 до 75 000 нет ни одного значения `id`), одни задачи будут практически бездействовать, а другим достанется повышенный объем работы. Пользователь может задать столбец разбиения при запуске задания импортирования, чтобы оптимизировать задание к фактическому распределению данных. Если задание импортирования выполняется как одна (последовательная) задача с параметром `-m 1`, разбиение не выполняется.

После генерирования кода десериализации и настройки `InputFormat` Sqoop отправляет задание в кластер MapReduce. Задачи отображения выполняют запросы и десериализуют строки данных из `ResultSet` в экземпляры сгенерированного класса, которые либо сохраняются напрямую в `SequenceFile`, либо преобразуются в текст с разделителями перед записью в HDFS.

Управление импортированием

Импортировать всю таблицу за один раз не обязательно — например, для импортирования можно выбрать подмножество столбцов таблицы. Пользователи также могут задать условие `WHERE`, ограничивающее импортируемые строки таблицы. например, если в прошлом месяце были импортированы данные с идентификаторами от 0 до 99 999, а в этом месяце в каталоге поставщика появились 1000 новых типов деталей, для импортирования можно задать условие `WHERE id >= 100000`; в этом случае задание импортирует все новые строки, добавленные в базу данных с момента предыдущего запуска. Пользовательские условия `WHERE` применяются перед разбиением и включаются в запросы, выполняемые каждой задачей.

Импортирование и согласованность данных

При импортировании данных в HDFS важно обеспечить доступ к целостному «снимку» исходных данных. Задачи отображения, параллельно читающие данные из базы, выполняются в разных процессах; следовательно, они не могут совместно

использовать одну транзакцию базы данных. Для обеспечения согласованности данных лучше всего заблокировать любые процессы, обновляющие существующие строки таблицы, на время импортирования.

Прямое импортирование

Архитектура Sqoop позволяет выбрать одну из нескольких стратегий импортирования. Для большинства баз данных используется решение на основе Sqoop, описанное ранее. Некоторые базы данных предоставляют специализированные средства для быстрого извлечения данных. Например, приложение MySQL *mysqldump* позволяет читать данные из таблицы быстрее, чем через канал JDBC. В документации Sqoop использование таких внешних инструментов называется *режимом прямого импортирования*. Режим прямого импортирования должен быть явно включен пользователем (аргумент `--direct`), так как он менее универсален, чем механизм JDBC. (Например, режим прямого импортирования MySQL не поддерживает большие объекты — столбцы CLOB и BLOB, потому что для загрузки этих столбцов в HDFS Sqoop приходится использовать API, специфический для JDBC).

Если база данных предоставляет такие средства, Sqoop может использовать их. Режим прямого импортирования из MySQL обычно намного эффективнее (в отношении количества задач отображения и необходимого времени) эквивалентного импортирования JDBC. Sqoop параллельно запускает несколько задач отображения. Эти задачи порождают экземпляры программы *mysqldump* и читают их результаты. Общая схема работает примерно так же, как распределенная реализация *mk-parallel-dump* из инструментария Maatkit. Sqoop также может выполнять прямое импортирование из PostgreSQL.

Даже когда режим прямого импортирования используется для обращения к содержимому базы данных, запросы метаданных все равно осуществляются через JDBC.

Работа с импортированными данными

После того, как данные будут импортированы в HDFS, они готовы к обработке программами MapReduce. Импортирование текстовых данных легко выполняется в сценариях Hadoop Streaming или заданиях MapReduce с используемой по умолчанию реализацией `TextInputFormat`.

Однако для использования отдельных полей импортированной записи необходимо обработать разделители полей (и все экранирующие символы), чтобы извлечь значения полей и преобразовать их к соответствующему типу данных. Например, идентификатор, хранимый в текстовом файле в виде строки "1", должен быть

преобразован в переменную `Integer` или `int` в языке Java. Сгенерированный класс таблицы, предоставляемый Sqoop, автоматизирует этот процесс, позволяя вам сосредоточиться на выполняемом задании MapReduce. Каждый автоматически сгенерированный класс содержит несколько перегруженных методов с именем `parse()`, которые работают с данными, представленными в виде `Text`, `CharSequence`, `char[]` или других распространенных типов.

Приложение MapReduce с именем `MaxWidgetId` (в коде примеров) находит строку данных с наибольшим идентификатором.

Класс может быть откомпилирован в файл JAR вместе с `Widget.java`. Для выполнения компиляции и Hadoop (`hadoop-core-version.jar`), и Sqoop (`sqoop-version.jar`) должны находиться в каталоге, входящем в путь к классам. Объединение файлов классов в файл JAR и их последующее выполнение выполняются следующей командой:

```
% jar cvvf widgets.jar *.class  
% HADOOP_CLASSPATH=/usr/lib/sqoop/sqoop-version.jar hadoop jar \  
> widgets.jar MaxWidgetId -libjars /usr/lib/sqoop/sqoop-version.jar
```

При выполнении в каталоге `maxwidgets` в HDFS создается файл с именем `part-r-00000`, содержащий следующий результат:

```
3,gadget,99.99,1983-08-13,13,Our flagship product
```

Стоит заметить, что в этом примере программы MapReduce объект `Widget` был передан из отображения в свертку; автоматически сгенерированный класс `Widget` реализует интерфейс `Writable`, предоставляемый Hadoop, который позволяет передавать объекты через механизм сериализации Hadoop, а также читать и записывать их в `SequenceFile`.

В примере `MaxWidgetId` используется новый MapReduce API. Приложения MapReduce, основанные на коде, сгенерированном Sqoop, могут использовать старый или новый API, хотя некоторые нетривиальные возможности (например, работа с большими объектами) удобнее использовать в новом API.

Импортование данных и Hive

Как упоминалось в главе 12, для многих типов анализа применение таких систем, как Hive, для выполнения реляционных операций радикально упрощает разработку аналитического конвейера. Оно особенно эффективно работает для данных, полученных из реляционного источника данных. Hive и Sqoop совместно образуют мощный инструмент для выполнения анализа.

Допустим, в системе имеется еще один журнал данных, сгенерированный по данным продаж Интернет-магазина. В нем хранятся идентификатор товара, количество, адрес доставки и дата заказа.

Фрагмент журнала может выглядеть так:

```
1,15,120 Any St.,Los Angeles,CA,90210,2010-08-01
3,4,120 Any St.,Los Angeles,CA,90210,2010-08-01
2,5,400 Some Pl.,Cupertino,CA,95014,2010-07-30
2,7,88 Mile Rd.,Manhattan,NY,10005,2010-07-18
```

Используя Hadoop для анализа журнала покупок, можно получить полезную информацию о продажах. Однако в сочетании с данными, извлеченными из реляционного источника (таблицы `widgets`), эта информация принесет еще больше пользы. В следующем примере мы определим, на какой почтовый индекс приходится наибольший объем продаж — это поможет придать более целенаправленный характер работе группы продаж. Однако для получений этой информации потребуются данные как из журнала продаж, так и из таблицы `widgets`.

Для работы этого примера таблица должна находиться в локальном файле с именем `sales.log`.

Начнем с загрузки данных в Hive:

```
hive> CREATE TABLE sales(widget_id INT, qty INT,
>   street STRING, city STRING, state STRING,
>   zip INT, sale_date STRING)
>   ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
OK
Time taken: 5.248 seconds
hive> LOAD DATA LOCAL INPATH "sales.log" INTO TABLE sales;
Copying data from file:/home/sales.log
Loading data to table sales
OK
Time taken: 0.188 seconds
```

Sqoop может сгенерировать таблицу Hive по таблице из существующего реляционного источника данных. Так как данные уже были импортированы в HDFS, мы можем сгенерировать определение таблицы Hive и затем загрузить данные, находящиеся в HDFS:

```
% sqoop create-hive-table --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets --fields-terminated-by ',' \
...
10/06/23 18:05:34 INFO hive.HiveImport: OK
10/06/23 18:05:34 INFO hive.HiveImport: Time taken: 3.22 seconds
10/06/23 18:05:35 INFO hive.HiveImport: Hive import complete.
% hive
hive> LOAD DATA INPATH "widgets" INTO TABLE widgets;
Loading data to table widgets
OK
Time taken: 3.265 seconds
```

При создании определения таблицы Hive для конкретного, уже импортированного набора данных необходимо указать разделители, использованные в этом наборе. В противном случае Sqoop позволит Hive использовать разделители по умолчанию (которые отличаются от разделителей по умолчанию Sqoop).



Система типов Hive беднее систем типов большинства систем SQL. Многие типы SQL не имеют прямых аналогов в Hive. Генерируя определение таблицы Hive для импортирования, Sqoop использует типы Hive, наиболее подходящие для хранения значений столбца. В отдельных случаях замена может привести к потере точности; тогда Sqoop выдает предупреждение следующего вида:

```
10/06/23 18:09:36 WARN hive.TableDefWriter:  
Column design_date had to be  
cast to a less precise type in Hive
```

Если вы знаете, что данные будут импортироваться из базы данных прямо в Hive, трехшаговый процесс — импортование данных в HDFS, создание таблицы Hive, загрузка данных из HDFS в Hive — можно сократить до одного шага. В процессе импортирования Sqoop может сгенерировать определение таблицы Hive и загрузить данные. Если бы данные еще были импортированы, мы могли бы выполнить следующую команду, которая создает таблицу `widgets` в Hive на основании копии из MySQL:

```
% sqoop import --connect jdbc:mysql://localhost/hadoopguide \  
> --table widgets -m 1 --hive-import
```



Команда `sqoop import` с аргументом `--hive-import` загружает данные прямо из базы данных в Hive; схема Hive вычисляется автоматически на основании схемы таблицы в источнике. Данная возможность позволяет приступить к работе с данными в Hive всего одной командой.

Независимо от выбранного способа импортирования теперь мы можем использовать набор данных `widgets` в сочетании с данными `sales` для определения почтовых индексов с наибольшими объемами продаж. Результат запроса будет сохранен в другой таблице на будущее:

```
hive> CREATE TABLE zip_profits (sales_vol DOUBLE, zip INT);  
OK  
hive> INSERT OVERWRITE TABLE zip_profits  
> SELECT SUM(w.price * s.qty) AS sales_vol, s.zip FROM SALES s  
> JOIN widgets w ON (s.widget_id = w.id) GROUP BY s.zip;  
...
```

```
3 Rows loaded to zip_profits
OK
hive> SELECT * FROM zip_profits ORDER BY sales_vol DESC;
...
OK
403.71 90210
28.0    10005
20.0    95014
```

Импортование больших объектов

Большинство баз данных позволяет сохранить в одном поле большой объем данных. В зависимости от типа (текстовые или двоичные) такие данные обычно представляются в таблице столбцом CLOB или BLOB. Часто база данных организует «особую» обработку таких данных. В частности, большинство таблиц имеет физическую структуру, изображенную на рис. 15.2. В процессе сканирования строк данных, отвечающих критериям определенного запроса, обычно требуется прочитать с диска все столбцы каждой строки. Если бы большие объекты хранились во «встроенным» формате, это привело бы к замедлению сканирования. Соответственно, большие объекты часто хранятся отдельно от своих строк, как показано на рис. 15.3. Для обращения к большому объекту часто требуется «открыть» его по ссылке, хранящейся в строке.

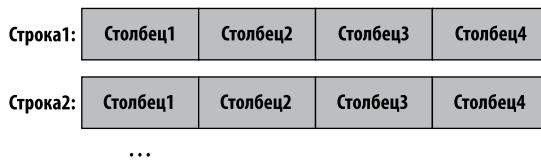


Рис. 15.2. Таблицы баз данных обычно представляются в виде массива строк, в которых все столбцы хранятся вплотную друг к другу

Трудности работы с большими объектами в базе данных наводят на мысль, что идеальным хранилищем для такой информации будет такая система, как Hadoop — намного лучше приспособленная к хранению и обработке больших сложных объектов данных. Sqoop позволяет извлекать большие объекты из таблиц и сохранять их в HDFS для дальнейшей обработки.

MapReduce, как и РСУБД, обычно *материализует* каждую запись перед тем, как передавать ее задаче отображения. При очень большом размере отдельных записей это может быть очень неэффективно.

Как показано ранее, записи, импортированные Sqoop, размещаются на диске в формате, очень близком к внутренней структуре базы данных: массив записей, в котором все поля расположены вплотную друг к другу. При выполнении программы MapReduce для импортированных записей каждая задача отображения должна полностью материализовать все поля каждой записи в выходном сплите. Если содержимое поля большого объекта актуально только для малого подмножества общего числа записей, использованных как входные данные программы MapReduce, полная материализация этих записей будет неэффективной. Более того, в зависимости от размера большого объекта полная материализация в памяти может оказаться невозможной.

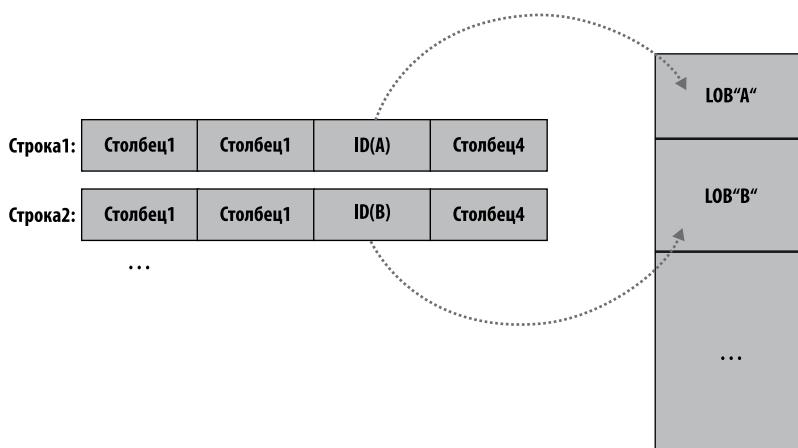


Рис. 15.3. Большие объекты обычно хранятся отдельно, а в строках находятся ссылки для обращения к ним

Для преодоления этих проблем Sqoop сохраняет большие объекты в отдельном файле, называемом **LobFile**. Формат **LobFile** позволяет хранить отдельные записи очень большого размера (благодаря использованию 64-разрядного адресного пространства). Каждая запись в **LobFile** содержит один большой объект. Формат **LobFile** позволяет клиентам сохранить ссылку без обращения к содержимому, для которого используется класс `java.io.InputStream` (для двоичных объектов) или `java.io.Reader` (для символьных объектов).

При импортировании записей «обычные» поля материализуются вместе в текстовый файл, в котором также сохраняется ссылка на **LobFile** с содержимым столбца **CLOB** или **BLOB**. Предположим, что таблица **widgets** содержит поле **BLOB** с именем **schematic**, в котором хранится принципиальная схема изделия.

Импортированная запись выглядит примерно так:

```
2,gizmo,4.00,2009-11-30,4,null,externallob(1f,lobfile0,100,5011714)
```

Текст `externalLob(...)` представляет ссылку на внешний большой объект, хранимый в формате `LobFile (1f)` в файле с именем `lobfile0`, с заданным смещением и длиной объекта внутри файла.

При работе с этой записью метод `Widget.get_schematic()` вернет объект типа `BlobRef`, ссылающийся на столбец `schematic`, но не вмещающий его фактическое содержимое. Метод `BlobRef.getDataStream()` открывает `LobFile` и возвращает `InputStream` для обращения к содержимому поля `schematic`.

Возможно, при выполнении задания MapReduce, обрабатывающего большое количество записей `Widget`, вам потребуется обратиться к полю `schematic` для небольшого подмножества записей. Эта система позволяет ограничиться затратами ресурсов на обращение только к необходимым большим объектам (размер каждой принципиальной схемы может составлять несколько мегабайт и более).

Классы `BlobRef` и `ClobRef` кэшируют ссылки на используемые файлы `LobFile` в задаче отображения. При обращении к полю `schematic` нескольких последовательных записей используется выравнивание файлового указателя по границе следующей записи.

Экспортирование

В Sqoop термином «импорт» обозначается перемещение данных из СУБД в HDFS. И наоборот, «экспорт» использует HDFS как источник данных, а удаленную базу данных — как приемник. Ранее в этой главе мы импортировали данные и использовали Hive для их анализа. Сейчас результаты этого анализа будут экспортированы в базу данных для потребления другими программами.

Прежде чем экспортировать таблицу из HDFS в базу данных, необходимо подготовить последнюю к получению данных, создав нужную таблицу. Хотя Sqoop может подобрать типы Java, подходящие для типов данных SQL, такое преобразование не работает в обратном направлении (например, существует несколько возможных определений столбцов SQL для хранения данных Java `String`: это может быть `CHAR(64)`, `VARCHAR(200)` или что-нибудь совершенно иное). А следовательно, нужно определить, какие типы будут наиболее подходящими.

Мы собираемся экспортировать таблицу `zip_profits` из Hive. Для этого необходимо создать в MySQL таблицу, которая содержит столбцы с соответствующими типами SQL, следующими в том же порядке:

```
% mysql hadoopguide
mysql> CREATE TABLE sales_by_zip (volume DECIMAL(8,2), zip INTEGER);
Query OK, 0 rows affected (0.01 sec)
```

Затем выполняется команда `export`:

```
% sqoop export --connect jdbc:mysql://localhost/hadoopguide -m 1 \
> --table sales_by_zip --export-dir /user/hive/warehouse/zip_profits \
> --input-fields-separated-by '\0001'
...
10/07/02 16:16:50 INFO mapreduce.ExportJobBase: Transferred 41 bytes in 10.8947
seconds (3.7633 bytes/sec)
10/07/02 16:16:50 INFO mapreduce.ExportJobBase: Exported 3 records.
```

Чтобы убедиться, что экспортирование прошло успешно, выполним выборку в MySQL:

```
% mysql hadoopguide -e 'SELECT * FROM sales_by_zip'
+-----+-----+
| volume | zip   |
+-----+-----+
| 28.00  | 10005 |
| 403.71 | 90210 |
| 20.00  | 95014 |
+-----+-----+
```

При создании таблицы `zip_profits` в Hive разделители явно не задавались, поэтому Hive использует свои разделители по умолчанию: Ctrl+A (Юникод 0x0001) между полями, символ новой строки в конце каждой записи. А когда мы использовали Hive для обращения к содержимому таблицы (в инструкции `SELECT`), Hive преобразовал данные в представление, разделенное символами табуляции для вывода на консоль. При прямом чтении таблиц из файлов необходимо сообщить Sqoop, какие разделители следует использовать. Sqoop по умолчанию предполагает, что записи разделены символом новой строки, но об использовании Ctrl+A как разделителя полей не знает. Для определения разделителей Sqoop поддерживает специальные последовательности, начинающиеся с символа обратной косой черты (\).

В синтаксисе из приведенного примера специальная последовательность заключена в апострофы, чтобы она была обработана оболочкой без лишних преобразований. Без апострофов начальную косую черту, возможно, придется экранировать (например, `--input-fields-separated-by \\0001`). Специальные последовательности, поддерживаемые Sqoop, перечислены в табл. 15.1.

Таблица 15.1. Специальные последовательности для определения непечатаемых символов как разделителей полей и записей

Последовательность	Описание
\b	Backspace
\n	Символ новой строки

Последовательность	Описание
\r	Возврат курсора
\t	Символ табуляции
\'	Апостроф
\"	Кавычка
\\\	Обратная косая черта
\0	NUL
\0ooo	Восьмеричное представление символа Юникода (ooo — восьмеричное значение).
\0xhhh	Шестнадцатеричное представление символа Юникода (hhh — шестнадцатеричное значение).

Подробнее об экспортировании

Архитектура экспортирования Sqoop основана на тех же принципах, что и архитектура импортирования (рис. 15.4). Перед выполнением экспортирования Sqoop выбирает стратегию на основании строки подключения к базе данных. Для большинства систем используется JDBC. Далее Sqoop генерирует класс Java по определению целевой таблицы. Сгенерированный класс умеет разбирать записи из текстовых файлов и вставлять значения соответствующих типов в таблицу (кроме способности читать столбцы из `ResultSet`). Затем запускается задание MapReduce, которое читает исходные файлы данных из HDFS, разбирает записи с использованием сгенерированного класса и выполняет выбранную стратегию экспортирования.

Стратегия экспортирования на основе JDBC строит пакет инструкций `INSERT`, каждая из которых добавляет в целевую таблицу несколько записей. Вставка многих записей в одной инструкции в большинстве баз данных осуществляется намного эффективнее, чем выполнение многих односторонних инструкций `INSERT`. Для чтения из HDFS и взаимодействия с базой данных используются разные программные потоки, чтобы операции ввода/вывода, в которых задействованы разные системы, перекрывались как можно меньше.

Для MySQL Sqoop может применить стратегию прямого режима с использованием `mysqlimport`. Каждая задача отображения порождает процесс `mysqlimport`, с которым она взаимодействует через именованный канал FIFO в локальной файловой системе. Затем данные передаются `mysqlimport` через канал FIFO, а оттуда поступают в базу данных.

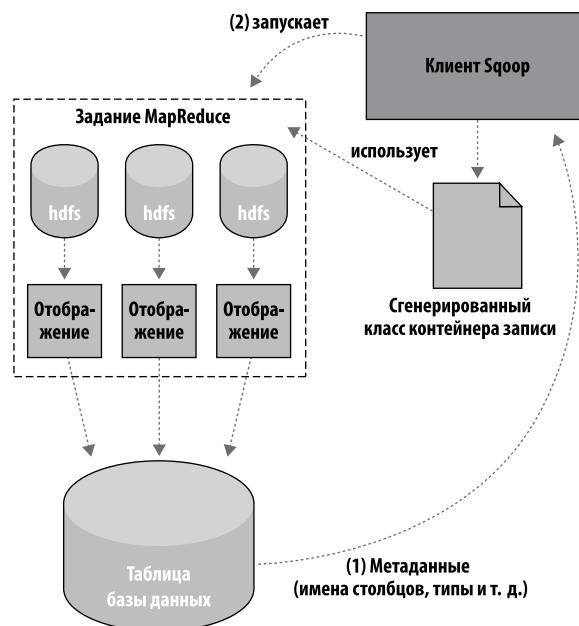


Рис. 15.4. Экспортирование выполняется параллельно с использованием MapReduce

Тогда как большинство заданий MapReduce, читающих данные из HDFS, выбирает степень параллелизма (количество задач отображения) в зависимости от количества и размера обрабатываемых файлов, система экспортирования Sqoop позволяет пользователям явно задавать количество задач. Производительность экспортирования зависит от количества параллельных операций записи в базу данных, поэтому Sqoop использует класс `CombineFileInputFormat` для группировки входных файлов в меньшем количестве задач отображения.

Экспортирование и транзакционность

Вследствие параллельной природы процесса экспортрование часто не является атомарной операцией. Sqoop порождает несколько задач для параллельного экспортования сегментов данных. Эти задачи могут завершаться в разное время; таким образом, даже при том, что внутри задач используются транзакции, результаты одной задачи могут стать видимыми раньше результатов другой задачи. Более того, для хранения транзакций базы данных часто используют буферы фиксированного размера. В результате одна транзакция не всегда содержит полный набор операций, выполненных задачей. Sqoop закрепляет результаты через

каждые несколько тысяч записей, чтобы избежать возможной нехватки памяти. Промежуточные результаты становятся видны в процессе экспортирования. Приложения, использующие результаты экспортирования, не должны запускаться до его завершения; в противном случае они могут увидеть частичные результаты.

Для решения этой проблемы Sqoop может экспортировать данные во временную накопительную таблицу, а в конце задания (если экспортование прошло успешно) переместить накопленные данные в приемную таблицу в одну транзакцию. Накопительная таблица задается параметром `--staging-table`; она должна уже существовать и иметь такую же схему, как и таблица-приемник. Кроме того, накопительная таблица должна быть пустой, если только в команду не включен параметр `--clear-staging-table`.

Экспортирование и SequenceFile

В рассмотренном примере исходные данные читаются из таблицы Hive, которая хранится в HDFS в виде текстового файла с разделителями. Sqoop также может экспортировать текстовые файлы с разделителями, которые не являются таблицами Hive — например, текстовые файлы с выходными данными задания MapReduce.

Также возможно экспортование записей, хранимых в формате `SequenceFile`, в выходную таблицу, хотя при этом действуют некоторые ограничения. `SequenceFile` может содержать произвольные типы записей. Команда экспортования Sqoop читает объекты из `SequenceFile` и отправляет их `OutputCollector` для передачи объекту экспортования `OutputFormat`. Чтобы эта схема работала, запись должна храниться в части «значение» формата пар «ключ-значение» `SequenceFile` и субклассировать абстрактный класс `org.apache.sqoop.lib.SqoopRecord` (как все классы, генерируемые Sqoop).

Если вы используете программу `sqoop-codegen` для генерирования реализации `SqoopRecord` на основе целевой таблицы экспортования, вы можете написать программу MapReduce, которая заполняет экземпляры класса и записывает их в `SequenceFile`. Программа `sqoop-export` сможет экспорттировать такие объекты `SequenceFile` в таблицу. Данные также могут оказаться в экземплярах `SqoopRecord` в `SequenceFile`, если данные были импортированы из таблицы базы данных в HDFS, там каким-то образом изменены, а затем результат был сохранен в `SequenceFile` с записями того же типа данных.

В этом случае Sqoop для чтения данных из `SequenceFile` повторно использует существующее определение класса вместо того, чтобы генерировать новый (временный) класс контейнера записи, как это делается при преобразовании текстовых записей в строки базы данных. Чтобы запретить генерирование кода и использовать

существующий класс записи и файл JAR, передайте Sqoop аргументы `--class-name` и `--jar-file`. В этом случае при экспортации записей Sqoop использует заданный класс, загруженный из указанного файла JAR.

В следующем примере мы снова импортируем таблицу `widgets` в формате `SequenceFile`, а затем экспортим ее обратно в базу данных как другую таблицу:

```
% sqoop import --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets -m 1 --class-name WidgetHolder --as-sequencefile \
> --target-dir widget_sequence_files --bindir .
...
10/07/05 17:09:13 INFO mapreduce.ImportJobBase: Retrieved 3 records.

% mysql hadoopguide
mysql> CREATE TABLE widgets2(id INT, widget_name VARCHAR(100),
   -> price DOUBLE, designed DATE, version INT, notes VARCHAR(200));
Query OK, 0 rows affected (0.03 sec)

mysql> exit;

% sqoop export --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets2 -m 1 --class-name WidgetHolder \
> --jar-file widgets.jar --export-dir widget_sequence_files
...
10/07/05 17:26:44 INFO mapreduce.ExportJobBase: Exported 3 records.
```

Во время импортирования мы выбираем формат `SequenceFile` и указываем, что файл JAR должен находиться в текущем каталоге (аргумент `--bindir`) для его последующего повторного использования. В противном случае файл будет размещен во временном каталоге. Затем создается приемная таблица для экспортации, которая имеет несколько отличающуюся схему (хотя и совместимую с исходными данными). Затем проводится экспорт, которое использует сгенерированный код для чтения данных из `SequenceFile` и их записи в базу данных.

Том Уайт
Hadoop: Подробное руководство

Перевел с английского Е. Матвеев

Заведующий редакцией	<i>А. Кривцов</i>
Руководитель проекта	<i>А. Кривцов</i>
Ведущий редактор	<i>Ю. Сергиенко</i>
Художественный редактор	<i>Л. Адуевская</i>
Корректор	<i>И. Тимофеева</i>
Верстка	<i>Л. Родионова</i>

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2: 95 3005 — литература учебная.
Подписано в печать 06.06.13. Формат 70x100/16. Усл. п. л. 54,180. Тираж 1000. Заказ
Отпечатано по технологии СтР в ООО «Полиграфический комплекс «ЛЕНИЗДАТ».
194044, Санкт-Петербург, ул. Менделеевская, 9. Телефон / факс (812) 495-56-10.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

-  Зарегистрируйтесь на нашем сайте в качестве партнера по адресу www.piter.com/ePartners
-  Получите свой персональный уникальный номер партнера
-  Выбирайте книги на сайте www.piter.com, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт www.piter.com)

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершил клиент, прийдя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка
<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

**Подробно о Партнерской программе
ИД «Питер» читайте на сайте
WWW.PITER.COM**



КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: www.piter.com
- по электронной почте: postbook@piter.com
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, 000 «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщают по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.



**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным
направлениям: история и публицистика, экономика и финансы, менеджмент
и маркетинг, компьютерные технологии, медицина и психология.**

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электроразводная», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Сузdalские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

 Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

 Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225
