5 Иллюстрация хранения данных на уровне пакетной обработки

В этой главе...

- Применение распределенной файловой системы HDFS.
- Высокоуровневая абстракция для манипулирования массивами данных в библиотеке Pail.

В предыдущей главе были рассмотрены требования к хранению главного массива данных, а также показано, что этим требованиям вполне удовлетворяет распределенная файловая система. Но в то же время в ней пояснялось, что непосредственное обращение к API такой файловой системы носит слишком низкоуровневый характер для всех видов операций, которые требуется выполнить над главным массивом данных. А в этой главе будет показано, как пользоваться конкретной распределенной файловой системой HDFS на практике и как автоматизировать решаемые задачи с помощью программного интерфейса более высокого уровня.

Как и во всех остальных иллюстративных главах данной книги, мы уделим основное внимание конкретным инструментальным средствам и рассмотрим особенности применения высокоуровневых понятий, представленных в предыдущей главе. Наша цель, как всегда, — не сравнивать и противопоставлять все возможные инструментальные средства, но подкрепить конкретными примерами применение высокоуровневых понятий на практике.

5.1. Применение распределенной файловой системы HDFS

Основы работы распределенной файловой системы HDFS были рассмотрены в предыдущей главе. Они еще раз приводятся ниже для краткого напоминания.

- Файлы разделяются на блоки, распределяемые по многим узлам в кластере.
- Блоки реплицируются по многим узлам, чтобы обеспечить постоянную доступность данных при выходе машин из строя.
- В узле имен отслеживаются блоки из каждого файла, а также места их хранения.

Знакомство с системой Hadoop

Настройка системы Hadoop может оказаться непростой задачей. В системе Hadoop имеются многочисленные параметры конфигурации, которые должны быть настроены на оптимальный режим работы оборудования. Чтобы не увязнуть в подробностях, рекомендуем загрузить предварительно сконфигурированную виртуальную машину для первого знакомства с Hadoop. Эта виртуальная машина ускорит процесс изучения системы HDFS и каркаса MapReduce, а также даст возможность лучше понять, когда следует устанавливать кластер.

На момент написания данной книги все поставщики Hadoop (компании Cloudera, Hortonworks и MapR) предоставляли общедоступный учебный материал для изучения данной системы. Рекомендуем получить доступ к Hadoop, чтобы проработать примеры, приведенные в этой и последующих главах.

Рассмотрим применение интерфейса API системы HDFS для манипулирования файлами и папками. Допустим, что требуется сохранить все данные регистрации на сервере. Ниже приведены некоторые примеры подобных данных.

```
$ cat logins-2012-10-25.txt
alex         192.168.12.125  Thu Oct 25 22:33 - 22:46 (00:12)
bob       192.168.8.251      Thu Oct 25 21:04 - 21:28 (00:24)
charlie      192.168.12.82   Thu Oct 25 21:02 - 23:14 (02:12)
doug         192.168.8.13    Thu Oct 25 20:30 - 21:03 (00:33)
...
```

Чтобы сохранить эти данные в системе HDFS, можно создать каталог для хранения массива данных и выгрузки файла следующим образом:

```
$ hadoop fs -mkdir /logins
$ hadoop fs -put logins-2012-10-25.txt /logins
```

Команды **hadoop fs** выполняются в оболочке Hadoop для непосредственного взаимодействия с системой HDFS. Их полный перечень приведен по адресу: **http://hadoop.apache.org/**

При автоматической выгрузке файл разбивается на блоки, которые распределяются по узлам данных

Вывести содержимое каталога списком можно следующим образом:

```
$ hadoop fs -ls -R /logins
-rw-r--r-- 3 hdfs hadoop 175802352 2012-10-26 01:38
   /logins/logins-2012-10-25.txt
```

Команда `ls` подобна одноименной команде в ОС Unix

А проверить содержимое файла можно так:

```
$ hadoop fs -cat /logins/logins-2012-10-25.txt
alex      192.168.12.125  Thu Oct 25 22:33 - 22:46 (00:12)
bob    192.168.8.251   Thu Oct 25 21:04 - 21:28 (00:24)
...
```

Как упоминалось ранее, файл автоматически разбивается на блоки и распределяется по узлам данных, когда он выгружается. Блоки и их местоположение можно распознать по следующей команде:

```
$ hadoop fsck /logins/logins-2012-10-25.txt -files -blocks -locations

/logins/logins-2012-10-25.txt 175802352 bytes, 2 block(s):
OK
0. blk_-1821909382043065392_1523 len=134217728
repl=3 [10.100.0.249:50010, 10.100.1.4:50010, 10.100.0.252:50010]
1. blk_2733341693279525583_1524 len=41584624
repl=3 [10.100.0.255:50010, 10.100.1.2:50010, 10.100.1.5:50010]
```

Файл хранится в двух блоках

IP-адреса и номера портов в узлах данных, где располагается каждый блок

5.1.1. Недостатки небольших файлов

Компоненты HDFS и MapReduce системы Hadoop тесно связаны, образуя инфраструктуру для хранения и обработки больших объемов данных. Каркас MapReduce будет подробнее рассмотрен в последующих главах, а до тех пор следует сказать, что вычислительные возможности системы Hadoop заметно сокращаются, когда данные хранятся во множестве небольших файлов, находящихся в системе HDFS. В частности, выполнение задания MapReduce может замедлиться на порядок величины, если оно обрабатывает данные объемом 10 Гбайт из множества небольших файлов, а не из нескольких больших файлов.

Дело в том, что задание MapReduce запускает целый ряд задач, каждая из которых предназначена для обработки одного блока из входного массива данных. И каждая задача несет определенные издержки на планирование и согласование своего выполнения, что влечет за собой постоянные затраты, поскольку для обработки каждого небольшого файла требуется отдельная задача. Такая особенность выполнения заданий MapReduce требует объединения данных из небольших файлов в массиве данных. С этой целью можно было бы написать код, в котором применяется интерфейс HDFS API, или же сформировать специальное задание MapReduce, но для реализации обоих подходов потребуются основательные знания внутреннего механизма работы MapReduce и немало труда.

5.1.2. Переход на более высокий уровень абстракции

Следует особо подчеркнуть, что в данной книге рассматриваются решения, которые являются не только масштабируемыми, отказоустойчивыми и производительными, но также *изящными*. Отчасти изящество решения определяется тем, что оно должно быть в состоянии лаконично выразить производимые вычисления.

Что же касается манипулирования главным массивом данных, то в предыдущей главе были представлены следующие две важные операции.

- Присоединение данных к массиву.
- Вертикальное разделение массива данных и принятие мер против нарушения уже существующего разделения данных.

Эти требования дополняются следующим характерным для HDFS требованием: небольшие файлы должны эффективно объединяться в большие файлы. Как было показано в предыдущей главе, решение подобных задач непосредственно в файлах и папках оказывается трудоемким и чреватым ошибками. Поэтому для более изящного их решения мы воспользуемся специальной библиотекой. В отличие от кода, в котором применяется интерфейс HDFS API, в приведенном ниже листинге демонстрируется применение библиотеки Pail.

Листинг 5.1. Абстракции задач сопровождения в системе HDFS

```
import java.io.IOException;
import backtype.hadoop.pail.Pail;

public class PailMove {
  public static void mergeData(String masterDir, String updateDir)
        throws IOException
  {
    Pail target = new Pail(masterDir);
    Pail source = new Pail(updateDir);
    target.absorb(source);
    target.consolidate();
  }
}
```

"Ведра" служат оболочками вокруг папок в системе HDFS

С помощью библиотеки Pail присоединения превращаются в однострочные операции

Небольшие файлы данных в "ведре" можно также объединить в одном вызове функции

Библиотека Pail позволяет присоединять папки в одной строке кода и объединять вместе небольшие файлы. Если при объединении данные в целевой папке оказываются в разных форматах файлов, библиотека Pail автоматически приведет новые данные к нужному формату. Если же в целевой папке употребляется другая схема вертикального разделения, библиотека Pail сгенерирует соответствующее исключение. А самое главное, что такая абстракция на более высоком уровне, как в Pail, позволяет обрабатывать данные непосредственно, а не пользоваться низкоуровневыми контейнерами вроде файлов и каталогов.

КРАТКОЕ ПОДВЕДЕНИЕ ИТОГОВ

Прежде чем рассматривать возможности библиотеки Pail дальше, уместно сделать небольшое отступление, чтобы восстановить более общую картину. Напомним, что главный массив данных является источником истины в лямбда-архитектуре, а следовательно, большие, постоянно разрастающиеся массивы

данных должны непременно обрабатываться на уровне пакетной обработки. Кроме того, для ответов на конкретные запросы должны быть простые и эффективные средства преобразования данных в пакетные представления. Материал этой главы носит более технический характер, чем в предыдущих главах, но всегда следует иметь в виду порядок интеграции компонентов в лямбда-архитектуре.

5.2. Хранение данных на уровне пакетной обработки с помощью библиотеки Pail

Библиотека Pail обеспечивает тонкое абстрагирование файлов и папок от библиотеки dfs-datastores (`http://github.com/nathanmarz/dfs-datastores`). Такое абстрагирование значительно упрощает управление коллекцией записей для пакетной обработки. Как подразумевает название библиотеки Pail, в ней применяются так называемые *"ведра"* — папки, в которых хранятся метаданные о массиве данных. Применяя эти метаданные, библиотека Pail позволяет безопасно действовать на уровне пакетной обработки, не беспокоясь о нарушении его целостности. Назначение библиотеки Pail — сделать важные операции (присоединения данных к массиву, вертикального разделения и объединения) безопасными, простыми и производительными.

Внутренне Pail является обычной библиотекой Java, где применяются интерфейсы Hadoop API. Она обеспечивает взаимодействие с файловой системой на низком уровне, предоставляя API, изолирующий пользователя этой библиотеки от сложностей внутреннего механизма Hadoop. Самое главное для библиотеки Pail — позволить ее пользователю сосредоточить основное внимание на самих данных, а не заботиться о том, как их хранить и сопровождать.

О значении библиотеки Pail

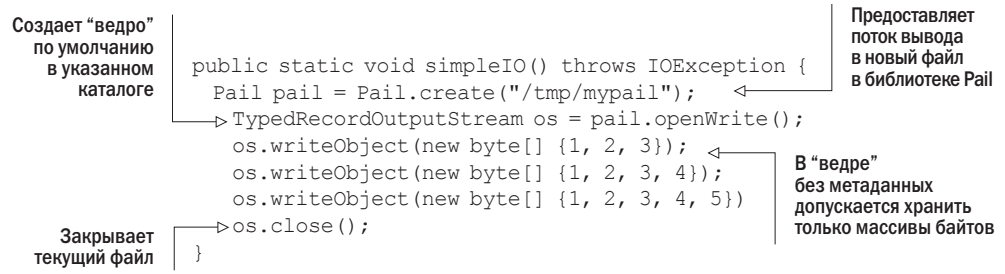
Как и многие другие инструментальные средства, рассматриваемые в данной книге, библиотека Pail была написана Натаном Марцем в процессе разработки лямбда-архитектуры. Мы представляем здесь эти технологии не для того, чтобы пропагандировать их, а для того, чтобы обсуждать контекст их происхождения и решаемые ими проблемы. А поскольку библиотека Pail была разработана Натаном Марцем, то она идеально соответствует рассмотренным до сих пор требованиям, предъявляемым к главному массиву данных, и эти требования естественно вытекают из основных принципов формирования запросов как функции всех данных. Впрочем, вы вольны пользоваться другими библиотеками или разрабатывать свои, а наша цель — предложить особый путь связывания принципов построения систем больших данных с имеющимися инструментальными средствами.

Ранее уже были рассмотрены характеристики системы HDFS, благодаря которым она является вполне жизнеспособным вариантом выбора для хранения главного массива данных на уровне пакетной обработки. Применяя библиотеку Pail, имейте в виду, что она сохраняет преимущества системы HDFS и в то же время оптимизирует операции над данными. Итак, рассмотрев основные операции Pail, подведем краткий итог общим ценностям данной библиотеки. В частности, рассмотрим действие этой библиотеки на примерах создания и записи данных в "ведро".

5.2.1. Основные операции в Pail

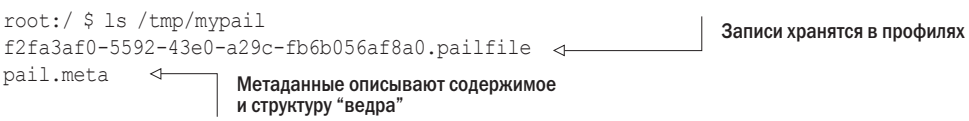
Понять принцип действия библиотеки Pail лучше всего на примере написания и выполнения кода на компьютере. С этой целью вам придется загрузить исходный код библиотеки dfs-datastores из хранилища данных GitHub и построить ее. Если в вашем распоряжении нет кластера Hadoop или виртуальной машины, в рассматриваемых далее примерах ваша локальная система будет трактоваться как HDFS. В таком случае вы сможете увидеть результаты выполнения вводимых команд, исследуя соответствующие каталоги в своей файловой системе.

Итак, начнем с создания нового "ведра" и сохранения в нем некоторых данных:




```
public static void simpleIO() throws IOException {
  Pail pail = Pail.create("/tmp/mypail");
  TypedRecordOutputStream os = pail.openWrite();
    os.writeObject(new byte[] {1, 2, 3});
    os.writeObject(new byte[] {1, 2, 3, 4});
    os.writeObject(new byte[] {1, 2, 3, 4, 5})
  os.close();
}
```

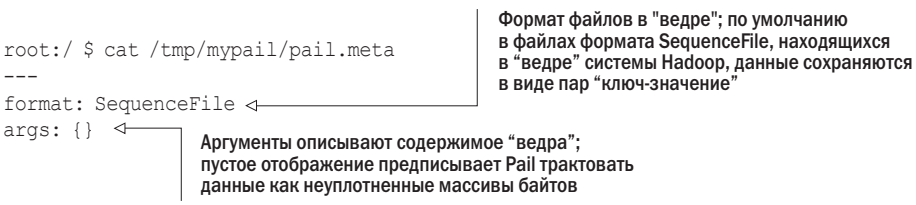
Проверяя свою файловую систему, вы непременно обнаружите, что в ней была создана папка `/tmp/mypail`, содержащая следующие файлы:




```
root:/ $ ls /tmp/mypail
f2fa3af0-5592-43e0-a29c-fb6b056af8a0.pailfile
pail.meta
```

"Ведерный" файл содержит сохраненные вами записи. Этот файл создается автоматически, и поэтому все созданные вами записи появляются сразу, т.е. приложение, читающее записи из "ведра", не заметит файл до тех пор, пока он не закроется в потоке записи. Кроме того, в "ведерных" файлах употребляются глобально однозначные имена, чтобы они назывались иначе в файловой системе. Эти однозначные имена позволяют параллельно и без всяких конфликтов записывать данные из многих источников в одно и то же "ведро".

Другой файл в данном каталоге содержит метаданные "ведра". Эти метаданные описывают тип данных, а также порядок их хранения в "ведре". В рассматриваемом здесь примере метаданные вообще не были указаны при построении "ведра", и поэтому в следующем файле содержатся настройки по умолчанию:




```
root:/ $ cat /tmp/mypail/pail.meta
---
format: SequenceFile
args: {}
```

Далее в этой главе будет рассмотрен другой файл `pail.meta`, содержащий более основательные данные, но общая его структура остается прежней. А теперь выясним, как средствами библиотеки Pail сохранять в "ведрах" реальные объекты, а не просто двоичные записи.

5.2.2. Сериализация объектов в "ведрах"

Чтобы сохранить объекты в "ведре", необходимо снабдить библиотеку Pail инструкциями по сериализации объектов в двоичные данные и их десериализации в обратном порядке. Вернемся к примеру с данными регистрации на сервере, чтобы показать, как это делается. В приведенном ниже листинге демонстрируется упрощенный код класса, представляющего процесс регистрации.

Листинг 5.2. Класс для регистрации без всяких излишеств

```
public class Login {
  public String userName;
  public long loginUnixTime;

  public Login(String _user, long _login) {
    userName = _user;
    loginUnixTime = _login;
  }
}
```

Чтобы сохранить объекты типа `Login` в "ведре", необходимо создать класс, реализующий интерфейс `PailStructure`. В приведенном ниже листинге определяется класс `LoginPailStructure`, описывающий порядок выполнения сериализации.

Листинг 5.3. Реализация интерфейса `PailStructure`

```
public class LoginPailStructure implements PailStructure<Login>{

  public Class getType() {
    return Login.class;
  }

  public byte[] serialize(Login login) {
    ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(byteOut);
    byte[] userBytes = login.userName.getBytes();
    try {
      dataOut.writeInt(userBytes.length);
      dataOut.write(userBytes);
      dataOut.writeLong(login.loginUnixTime);
      dataOut.close();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return byteOut.toByteArray();
  }

  public Login deserialize(byte[] serialized) {
    DataInputStream dataIn =
           new DataInputStream(new ByteArrayInputStream(serialized));
    try {
      byte[] userBytes = new byte[dataIn.readInt()];
      dataIn.read(userBytes);
      return new Login(new String(userBytes), dataIn.readLong());
```

"Ведро" со структурой для хранения только объектов типа `Login`

Объекты типа `Login` должны быть сериализированы при сохранении в "ведерных" файлах

Объекты типа `Login` восстанавливаются при последующем чтении из "ведерных" файлов

```
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }

  public List<String> getTarget(Login object) {
    return Collections.EMPTY_LIST;
  }

  public boolean isValidTarget(String... dirs) {
    return true;
  }
}
```

В методе `getTarget()` определяется схема вертикального разделения, хотя она не используется в данном примере

В методе `isValidTarget()` определяется соответствие заданного пути схеме вертикального разделения, но и он не используется в данном примере

Если передать объект типа `LoginPailStructure` функции `create()` из библиотеки Pail, как показано ниже, то инструкции для сериализации будут использованы в результирующем "ведре". После этого достаточно передать ему объекты типа `Login`, чтобы они были автоматически сериализированы средствами Pail.

```
public static void writeLogins() throws IOException {
  Pail<Login> loginPail = Pail.create("/tmp/logins",
                    new LoginPailStructure());
  TypedRecordOutputStream out = loginPail.openWrite();
  out.writeObject(new Login("alex", 1352679231));
  out.writeObject(new Login("bob", 1352674216));
  out.close();
}
```

Создает "ведро" с новой структурой

Аналогично при чтении данных записи автоматически десериализируются средствами Pail. Ниже показано, как перебрать все только что записанные объекты. Как только данные будут сохранены в "ведре", их можно благополучно обработать с помощью встроенных в Pail операций.

```
public static void readLogins() throws IOException {
  Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
  for(Login l : loginPail) {
    System.out.println(l.userName + " " + l.loginUnixTime);
  }
}
```

Для типа объектов в "ведре" поддерживается интерфейс `Iterable`

5.2.3. Выполнение пакетных операций средствами Pail

В библиотеку Pail встроена поддержка целого ряда типичных операций. Именно в этих операциях проявляются преимущества управления записями не вручную, а средствами Pail. Все операции реализованы с помощью каркаса MapReduce и поэтому допускают масштабирование независимо от объема данных в "ведре", будь то в гига- или терабайтах. Мы еще вернемся к более подробному рассмотрению каркаса MapReduce в последующих главах, а пока лишь заметим, что операции автоматически распараллеливаются и выполняются на всех рабочих машинах в кластере.

В предыдущем разделе обсуждалось, насколько важны операции присоединения и объединения. Как и следовало ожидать, оба эти вида операций поддерживаются в библиотеке Pail. Особенно изящна операция присоединения. Она проверяет, насколько допустимо присоединять "ведра" друг к другу. В частности, она не позволит присоединить "ведро", содержащее символьные строки, к "ведру", содержащему целые числа. Если в "ведрах" хранятся записи одинакового типа, но в разных форматах файлов, то операция присоединения приведет данные в соответствие с форматом целевого "ведра". Это означает, что для данного "ведра" будет соблюден выбранный компромисс между затратами на хранение и обработку данных.

По умолчанию операция объединения осуществляет слияние небольших файлов в новые файлы, размеры которых как можно более точно соответствуют 128 Мбайт, т.е. стандартному размеру блока в системе HDFS. Эта операция распараллеливается средствами MapReduce.

Допустим, что в рассматриваемом здесь примере дополнительные данные регистрации находятся в отдельном "ведре" и требуется соединить их с исходным "ведром". Обе операции присоединения и объединения выполняются в следующем фрагменте кода:

```
public static void appendData() throws IOException {
  Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
  Pail<Login> updatePail = new Pail<Login>("/tmp/updates");
  loginPail.absorb(updatePail);
  loginPail.consolidate();
}
```

Из всего изложенного выше можно сделать следующий главный вывод: встроенные функции позволяют сосредоточить основное внимание на обработке данных, не заботясь о правильности манипулирования файлами.

5.2.4. Вертикальное разделение средствами Pail

Как упоминалось ранее, вертикальное разделение данных в системе HDFS можно произвести, используя несколько папок. Если попытаться управлять вертикальным разделением вручную, то можно легко забыть, что два массива данных разделяются по-разному, соединив их по ошибке. Аналогично не составит большого труда случайно нарушить структуру разделения при объединении данных. Правда, библиотека Pail обладает достаточно развитой логикой, чтобы соблюсти структуру "ведра" и уберечь от подобного рода ошибок.

Чтобы создать структуру разделенных каталогов для "ведра", необходимо реализовать два дополнительных метода из интерфейса `PailStructure`. Эти методы перечислены ниже.

- Метод `getTarget()`. Получает запись, определяет структуру каталогов, где эта запись должна храниться, и возвращает путь в виде списка объектов типа `String`.
- Метод `isValidTarget()`. Получает массив объектов типа `String`, составляет путь к каталогу и определяет, согласуется ли он со схемой вертикального разделения.

Эти методы используются в библиотеке Pail для соблюдения структуры каталогов и автоматического распределения записей по соответствующим подкаталогам. В приведенном ниже листинге демонстрируется вертикальное разделение объектов типа `Login` таким образом, чтобы сгруппировать записи по дате регистрации.

Листинг 5.4. Схема вертикального разделения записей типа `Login`

```
public class PartitionedLoginPailStructure extends LoginPailStructure {
  SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");

  public List<String> getTarget(Login object)
    ArrayList<String> directoryPath = new ArrayList<String>();
    Date date = new Date(object.loginUnixTime * 1000L);
    directoryPath.add(formatter.format(date));
    return directoryPath;
  }

  public boolean isValidTarget(String... strings) {
    if(strings.length != 1) return false;
    try {
      return (formatter.parse(strings[0]) != null);
    }
    catch(ParseException e) {
      return false;
    }
  }
}
```

Данные регистрации вертикально разделяются по папкам в соответствии с датой регистрации

Отметка времени объекта типа `Login` преобразуется в понятную форму

Метод `isValidTarget()` проверяет, имеет ли структура каталогов единичную глубину, а имя папки — дату

Исходя из новой структуры "ведра", библиотека Pail выбирает подходящий подкаталог всякий раз, когда записывается новый объект типа `Login`:

```
public static void partitionData() throws IOException {
  Pail<Login> pail = Pail.create("/tmp/partitioned_logins",
                 new PartitionedLoginPailStructure());
  TypedRecordOutputStream os = pail.openWrite();
  os.writeObject(new Login("chris", 1352702020));
  os.writeObject(new Login("david", 1352788472));
  os.close();
}
```

1352702020 — отметка времени 2012-11-11, 2:33:40 PST

1352788472 — отметка времени 2012-11-12, 22:34:32 PST

Проверив этот новый "ведерный" каталог, можно убедиться, что данные были разделены правильно:

```
root:/ $ ls -R /tmp/partitioned_logins
2012-11-11 2012-11-12 pail.meta

/tmp/partitioned_logins/2012-11-11:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile

/tmp/partitioned_logins/2012-11-12:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile
```

В "ведре" были созданы папки для разных дат регистрации

5.2.5. Форматы и уплотнение "ведерных" файлов

В структуре каталогов "ведра" данные сохраняются во многих файлах. Чтобы контролировать порядок сохранения записей в этих файлах, достаточно указать

используемый формат файлов. Это дает возможность выбрать компромисс между величиной пространства для хранения данных и производительностью чтения записей из "ведра". Как пояснялось ранее в главе, это основной контроль, необходимый для удовлетворения потребностей приложения в настройке.

Несмотря на возможность реализовать собственный формат файлов, в библиотеке Pail по умолчанию используются файлы формата `SequenceFile` на платформе Hadoop. Этот широко употребляемый формат позволяет параллельно обрабатывать отдельные файлы средствами MapReduce. Он также имеет встроенную поддержку уплотнения записей в файлах.

В качестве примера ниже показано, как создать "ведро", в котором используется формат `SequenceFile` и алгоритм блочного уплотнения gzip.

```
public static void createCompressedPail() throws IOException {
  Map<String, Object> options = new HashMap<String, Object>();
  options.put(SequenceFileFormat.CODEC_ARG,
              SequenceFileFormat.CODEC_ARG_GZIP);
  options.put(SequenceFileFormat.TYPE_ARG,
              SequenceFileFormat.TYPE_ARG_BLOCK);
  LoginPailStructure struct = new LoginPailStructure();
  Pail compressed = Pail.create("/tmp/compressed",
                    new PailSpec("SequenceFile", options, struct));
}
```

Содержимое "ведра" будет уплотнено по алгоритму gzip

Блоки записи будут сжаты вместе, а не отдельно по строкам

Создает новое "ведро" для хранения объектов типа `Login` в нужном формате

Все эти свойства можно затем наблюдать в метаданных "ведра", как показано ниже.

```
root:/ $ cat /tmp/compressed/pail.meta
---
format: SequenceFile
structure: manning.LoginPailStructure
args:
  compressionCodec: gzip
  compressionType: block
```

Полное имя класса `LoginPailStructure`

Параметры уплотнения "ведерных" файлов

Всякий раз, когда записи вводятся в это "ведро", они автоматически уплотняются. Это "ведро" занимает намного меньше места, но требует бо́льших затрат вычислительных ресурсов ЦП на чтение и запись данных.

5.2.6. Преимущества библиотеки Pail

Потратив время на изучение внутреннего механизма работы библиотеки Pail, очень важно уяснить преимущества, которые она дает по сравнению с исходными возможностями системы HDFS. В табл. 5.1 приведены выгоды, которые приносит библиотека Pail по отношению к упоминавшемуся ранее перечню требований к хранению данных в главном массиве на уровне пакетной обработки.

На этом краткий экскурс в возможности библиотеки Pail завершается. Она служит удобной и эффективной абстракцией для взаимодействия с данными на уровне пакетной обработки, избавляя от необходимости вникать в подробности функционирования базовой файловой системы.

Таблица 5.1. Преимущества библиотеки Pail для хранения данных в главном массиве

Операция	Требование	Описание
Запись	Эффективность присоединения новых данных	Библиотека Pail предоставляет первоклассный интерфейс для присоединения данных и уберегает от выполнения недопустимых операций, на что неспособен исходный интерфейс HDFS API
	Масштабируемость хранилища	Узел имен содержит все пространство имен HDFS в оперативной памяти, что может оказаться обременительным, если в файловой системе находится большое количество небольших файлов. Операция объединения в Pail сокращает общее количество блоков HDFS и облегчает бремя, накладываемое на узел имен
Чтение	Поддержка параллельной обработки	Количество задач в задании MapReduce определяется количеством блоков в массиве данных. Объединение содержимого “ведра” позволяет сократить количество требуемых задач и повысить эффективность обработки данных
	Возможность вертикального разделения данных	Выводимые данные, записываемые в “ведре”, автоматически разделяются, причем каждый факт хранится в соответствующем каталоге. Подобная структура каталогов строго соблюдается для всех операций в Pail
И то и другое	Коррекция затрат на хранение и обработку	В библиотеку Pail встроена поддержка приведения данных в формат, указанный в структуре “ведра”. Такое приведение происходит автоматически при выполнении операций над содержимым “ведра”
	Соблюдение неизменяемости	Библиотека Pail служит лишь тонкой оболочкой вокруг файлов и папок, и поэтому она позволяет соблюсти свойство неизменяемости установкой соответствующих разрешений, как это делается непосредственно в системе HDFS

5.3. Хранение главного массива данных для приложения SuperWebAnalytics.com

Как было показано в предыдущей главе, высокоуровневые принципы для хранения данных в приложении SuperWebAnalytics.com довольно просты. Они состоят в том, чтобы воспользоваться распределенной файловой системой и вертикальным разделением, сохраняя свойства и ребра граф-схемы в разных подкаталогах. А теперь воспользуемся уже известными инструментальными средствами, чтобы воплотить эти принципы в жизнь.

Напомним исполняемую схему, разработанную средствами Apache Thrift для приложения SuperWebAnalytics.com. Ниже приведен фрагмент этой схемы.

Все факты в массиве данных представлены отметкой времени базовым блоком данных

```
struct Data {
  1: required Pedigree pedigree;
  2: required DataUnit dataunit;
}
```

Базовый блок данных описывает ребра и свойства граф-схемы массива данных

```
union DataUnit {
  1: PersonProperty person_property;
```

```
  2: PageProperty page_property;
  3: EquivEdge equiv;
  4: PageViewEdge page_view;
}

union PersonPropertyValue {
  1: string full_name;
  2: GenderType gender;
  3: Location location;
}
```

Значение свойства может относиться к разным типам

На рис. 5.1 показано, каким образом эта схема воплощается в папках.

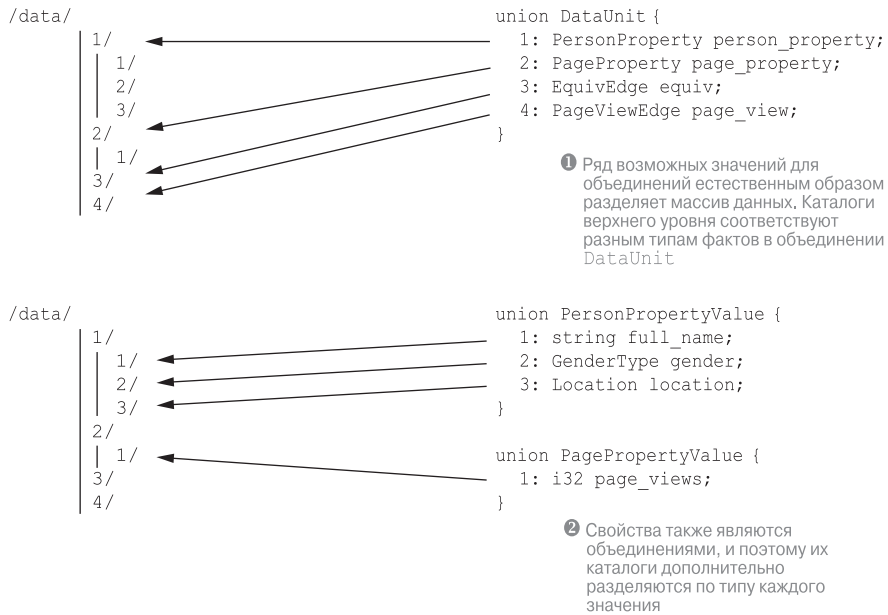


Рис. 5.1. Объединения в граф-схеме обеспечивают естественную схему для вертикального разделения массива данных

Чтобы воспользоваться системой HDFS и библиотекой Pail для разработки приложения SuperWebAnalytics.com, необходимо определить структурированное "ведро", предназначенное для хранения объектов типа `Data` и также соблюдающее упомянутую выше схему вертикального разделения. Код реализации такой возможности непростой, и поэтому представим его поэтапно.

1. Прежде всего создается абстрактная структура "ведра" для хранения объектов Apache Thrift. Сериализация этих объектов не зависит от типа сохраняемых данных, а код получается более ясным благодаря разделению логики.
2. Далее структура "ведра" получается из абстрактного класса для хранения объектов типа `Data` в приложении SuperWebAnalytics.com.
3. И наконец, определяется дополнительный подкласс, реализующий требующуюся схему вертикального разделения.

Читая этот раздел, не обращайте особого внимания на код. Самое главное, что этот код пригоден для реализации любой граф-схемы. Даже если граф-схема будет развиваться дальше, этот код все равно останется работоспособным.

5.3.1. Структурированное "ведро" для хранения объектов Apache Thrift

Создать структуру "ведра" для хранения объектов Apache Thrift необычайно просто, поскольку основные хлопоты Apache Thrift берет на себя. В приведенном ниже листинге демонстрируется применение утилит Apache Thrift для сериализации и десериализации данных.

Листинг 5.5. Обобщенная абстрактная структура "ведра" для сериализации объектов Apache Thrift

Благодаря обобщениям в Java структура "ведра" оказывается пригодной для хранения любых объектов Apache Thrift

```
public abstract class ThriftPailStructure<T extends Comparable>
    implements PailStructure<T>
{
  private transient TSerializer ser;
  private transient TDeserializer des;
```

TSerializer и Tdeserializer являются утилитами Apache Thrift для сериализации объектов в двоичные массивы и обратно

Утилиты Apache Thrift строятся только по требованию

```
  private TSerializer getSerializer() {
    if(ser==null) ser = new TSerializer();
    return ser;
  }

  private TDeserializer getDeserializer() {
    if(des==null) des = new TDeserializer();
    return des;
  }

  public byte[] serialize(T obj) {
    try {
      return getSerializer().serialize((TBase)obj);
    } catch (TException e) {
      throw new RuntimeException(e);
    }
  }
```

Объект приводится к типу объекта Apache Thrift для сериализации

Новый объект Apache Thrift создается перед десериализацией

```
  public T deserialize(byte[] record) {
    T ret = createThriftObject();
    try {
      getDeserializer().deserialize((TBase)ret, record);
    } catch (TException e) {
      throw new RuntimeException(e);
    }
    return ret;
  }

  protected abstract T createThriftObject();
}
```

Конструктор объектов Apache Thrift должен быть реализован в порожденном классе

5.3.2. Основное "ведро" для приложения SuperWebAnalytics.com

Далее можно определить основной класс для хранения объектов типа `Data` в приложении SuperWebAnalytics.com. С этой целью создается конкретный подкласс, производный от класса `ThriftPailStructure`, как показано в приведенном ниже листинге.

Листинг 5.6. Конкретная реализация для объектов типа `Data`

```
public class DataPailStructure extends ThriftPailStructure<Data> {
  public Class getType() {
    return Data.class;
  }

  protected Data createThriftObject() {
    return new Data();
  }

  public List<String> getTarget(Data object) {
    return Collections.EMPTY_LIST;
  }

  public boolean isValidTarget(String... dirs) {
    return true;
  }
}
```

Обозначает, что в "ведре" хранятся объекты типа `Data`

Требуется в классе `ThriftPailStructure` для создания объекта и с целью сериализации

В структуре этого "ведра" вертикальное разделение не применяется

5.3.3. Расчлененное "ведро" для вертикального разделения массива данных

И наконец, нужно создать структуру "ведра", реализующего стратегию вертикального разделения данных по граф-схеме. Это самая трудная стадия описываемого здесь процесса. Все приведенные ниже фрагменты кода извлечены из класса `SplitDataPailStructure`, выполняющего эту задачу.

На самом верхнем уровне в коде из класса `SplitDataPailStructure` проверяется класс `DataUnit` с целью создать отображение идентификаторов Apache Thrift на классы для обработки данных соответствующего типа. Подобное отображение для приложения SuperWebAnalytics.com приведено на рис. 5.2.

Рис. 5.2. Отображение полей для класса `DataUnit` из приложения SuperWebAnalytics.com в классе `SplitDataPailStructure`

В приведенном ниже листинге демонстрируется код, формирующий отображение полей. Он пригоден для любой граф-схемы, а не только для той, что рассматривается в данном примере.

Листинг 5.7. Код, формирующий отображение полей для граф-схемы

```
public class SplitDataPailStructure extends DataPailStructure {
  public static HashMap<Short, FieldStructure> validFieldMap =
       new HashMap<Short, FieldStructure>();

  static {
    for(DataUnit._Fields k: DataUnit.metaDataMap.keySet()) {
      FieldValueMetaData md = DataUnit.metaDataMap.get(k).valueMetaData;
      FieldStructure fieldStruct;
      if(md instanceof StructMetaData &&
        ((StructMetaData) md).structClass
          .getName().endsWith("Property"))
      {
        fieldStruct = new PropertyStructure(
                    ((StructMetaData) md).structClass);
      } else {
        fieldStruct = new EdgeStructure();
      }
      validFieldMap.put(k.getThriftFieldId(), fieldStruct);
    }
  }

  // остальная часть класса опущена
}
```

FieldStructure — это интерфейс как для ребер, так и для свойств граф-схемы

Код Apache Thrift для проверки и обхода объекта типа DataUnit

Свойства определяются по имени класса проверяемого объекта

Если имя класса не оканчивается на слово "Property", то оно обозначает ребро

Как упоминается в комментариях к приведенному выше коду, `FieldStructure` — это интерфейс, общий для классов `PropertyStructure` и `EdgeStructure`. Ниже приведено определение этого интерфейса.

```
protected static interface FieldStructure {
public boolean isValidTarget(String[] dirs);
public void fillTarget(List<String> ret, Object val);
}
```

В дальнейшем мы рассмотрим классы `PropertyStructure` и `EdgeStructure` более подробно. А до тех пор ниже показано, каким образом с помощью этого интерфейса осуществляется вертикальное разделение таблицы.

```
// Приведенные ниже методы взяты из класса SplitDataPailStructure

public List<String> getTarget(Data object) {
  List<String> ret = new ArrayList<String>();
  DataUnit du = object.get_dataunit();
  short id = du.getSetField().getThriftFieldId();
  ret.add("" + id);
  validFieldMap.get(id).fillTarget(ret, du.getFieldValue());
  return ret;
}

public boolean isValidTarget(String[] dirs) {
  if(dirs.length==0) return false;
  try {
    short id = Short.parseShort(dirs[0]);
    FieldStructure s = validFieldMap.get(id);
    if(s==null)
      return false;
```

Каталог верхнего уровня определяется при проверке класса DataUnit

Любое дальнейшее разделение передается интерфейсу FieldStructure

При проверке достоверности сначала выясняется, находится ли идентификатор поля из класса DataUnit в отображении полей

```
      else
        return s.isValidTarget(dirs);
    } catch(NumberFormatException e) {
      return false;
    }
  }
```

Любые дополнительные проверки передаются интерфейсу `FieldStructure`

Класс `SplitDataPailStructure` отвечает за каталог верхнего уровня при вертикальном разделении. А ответственность за любые дополнительные подкаталоги он передает классам, реализующим интерфейс `FieldStructure`. Следовательно, как только классы `EdgeStructure` и `PropertyStructure` будут определены, дело можно считать сделанным.

Ребра являются структурами, а следовательно, они не подлежат дальнейшему разделению. Благодаря этому класс `EdgeStructure` определяется довольно просто, как показано ниже.

```
protected static class EdgeStructure implements FieldStructure {
  public boolean isValidTarget(String[] dirs) { return true; }
  public void fillTarget(List<String> ret, Object val) { }
}
```

Но свойства являются объединениями подобно классу `DataUnit`. Поэтому в реализующем их коде выполняется проверка с целью создать множество достоверных идентификаторов полей Apache Thrift для класса заданного свойства. Ради полноты изложения в приведенном ниже листинге класс свойства представлен полностью, но особое внимание следует обратить на построение множества и его применение для выполнения контракта на интерфейс `FieldStructure`.

Листинг 5.8. Класс `PropertyStructure`

```
protected static class PropertyStructure implements FieldStructure {
  private TFieldIdEnum valueId;
  private HashSet<Short> validIds;

  public PropertyStructure(Class prop) {
    try {
      Map<TFieldIdEnum, FieldMetaData> propMeta = getMetadataMap(prop);
      Class valClass = Class.forName(prop.getName() + "Value");
      valueId = getIdForClass(propMeta, valClass);

      validIds = new HashSet<Short>();
      Map<TFieldIdEnum, FieldMetaData> valMeta = getMetadataMap(valClass);
      for(TFieldIdEnum valId: valMeta.keySet()) {
        validIds.add(valId.getThriftFieldId());
      }
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }

  public boolean isValidTarget(String[] dirs) {
    if(dirs.length < 2) return false;
    try { 1((check))
      short s = Short.parseShort(dirs[1]);
      return validIds.contains(s);
```

Множество идентификаторов Apache Thrift для типов значений свойств

Свойство является структурой Apache Thrift, содержащей поле со значением свойства; служит идентификатором этого поля

Выполняет синтаксический анализ метаданных Apache Thrift, чтобы получить идентификатор поля со значением свойства

Выполняет синтаксический анализ метаданных Apache Thrift, чтобы получить все достоверные идентификаторы полей со значением свойства

Вертикальное разделение значения свойства на глубину не меньше двух

```
  } catch(NumberFormatException e) {
    return false;
  }
}

public void fillTarget(List<String> ret, Object val) {
  ret.add("" + ((TUnion) ((TBase)val)
     .getFieldValue(valueId))
     .getSetField()
     .getThriftFieldId());
}

private static Map<TFieldIdEnum, FieldMetaData>
      getMetadataMap(Class c)
{
  try {
    Object o = c.newInstance();
    return (Map) c.getField("metaDataMap").get(o);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}

private static TFieldIdEnum getIdForClass(
      Map<TFieldIdEnum, FieldMetaData> meta, Class toFind)
{
  for(TFieldIdEnum k: meta.keySet()) {
    FieldValueMetaData md = meta.get(k).valueMetaData;
    if(md instanceof StructMetaData) {
      if(toFind.equals(((StructMetaData) md).structClass)) {
        return k;
      }
    }
  }
}

throw new RuntimeException("Could not find " + toFind.toString() +
     " in " + meta.toString());
}
```

Использует идентификаторы Apache Thrift, чтобы составить путь к каталогу для текущего факта

GetMetadataMap() и **getIdForClass()** являются вспомогательными методами для проверки объектов Apache Thrift

Проанализировав приведенный выше код, сделайте перерыв — вы его заслужили. Этот код примечателен тем, что он требует одноразовых затрат. Как только будет определена структура "ведра" для главного массива данных, дальнейшее взаимодействие на уровне пакетной обработки не составит особого труда. Более того, этот код можно применить в любом проекте, где составлена граф-схема средствами Apache Thrift.

Резюме

В этой главе было показано, что для хранения массива данных в системе HDFS требуется решить типичные задачи присоединения новых данных к главному массиву, вертикального разделения данных на многие папки и объединения небольших файлов. У вас была возможность убедиться, что выполнение этих задач

непосредственным обращением к интерфейсу HDFS API — дело нелегкое и чреватое ошибками, обусловленными человеческим фактором.

Далее в этой главе была представлена библиотека Pail, обеспечивающая требующийся уровень абстракции, избавляя вас от необходимости разбираться в форматах файлов и структуре каталогов системы HDFS и тем самым упрощая задачи надежного вертикального разделения и выполнения типичных операций над массивом данных. Для подобной абстракции в конечном счете требуется совсем немного строк кода. Вертикальное разделение происходит автоматически, а задачи вроде присоединения и объединения упрощаются до однострочных операций. Это означает, что вы можете уделить основное внимание обработке записей, не вдаваясь в подробности их хранения.

С помощью системы HDFS и библиотеки Pail в этой главе был представлен способ хранения главного массива данных, удовлетворяющий всем требованиям и довольно изящный в употреблении. Независимо от того, воспользуетесь ли вы этими инструментальными средствами или нет, мы показали на конкретном примере, насколько изящной может быть эта часть архитектуры, и надеемся, что этот пример вдохновит и нацелит вас на достижение хотя бы такого же уровня изящества.

В следующей главе речь пойдет об эффективном использовании хранения записей для выполнения следующего важного шага в реализации лямбда-архитектуры: вычисления пакетных представлений.