

第一部分 大数据与 Spark 概览

第一章 Apache Spark 是什么

Apache Spark 是一个统一的分布式内存计算引擎，包括一组用于在计算机集群上进行并行数据处理的函数库。截止目前，Spark 已经成为大数据开发人员以及数据科学家的必备工具。Spark 支持多种广泛使用的编程语言(Python、Java、Scala 和 R)，包括用于各种计算任务的库：SQL 结构化查询到流计算、机器学习。Spark 可以运行在笔记本电脑上，也可以运行在数千台服务器的集群上。

图 1-1 说明了提供给最终用户的所有组件和库。

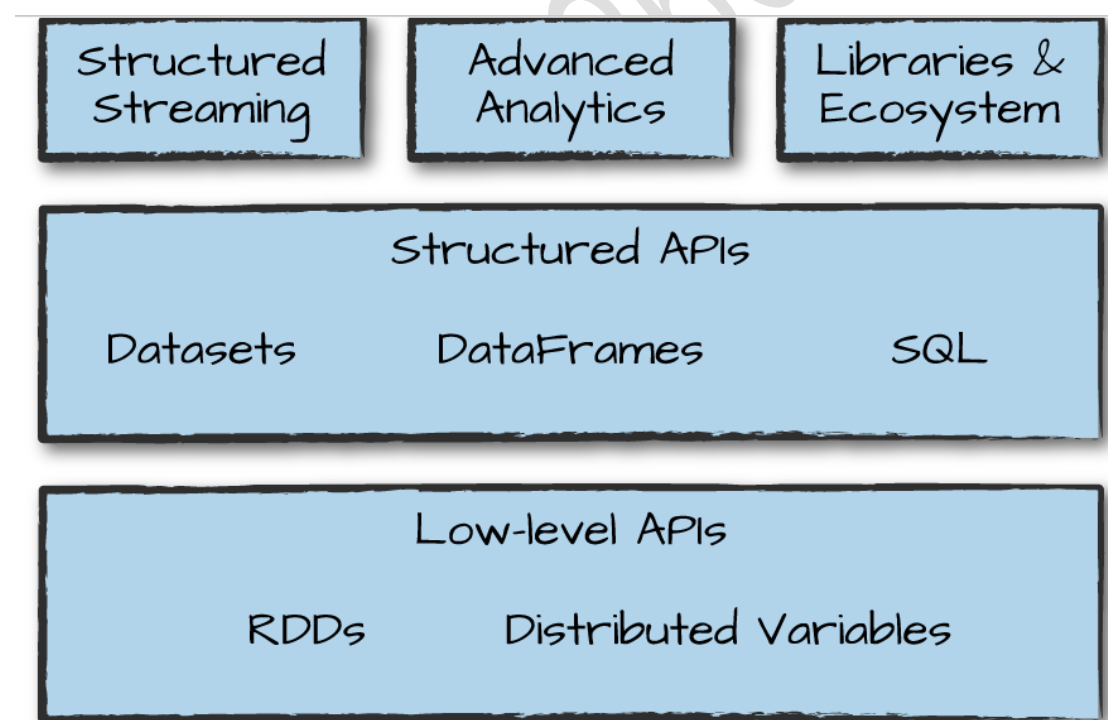


Figure 1-1. Spark's toolkit

你会注意到这些类别大致对应于本书的不同部分。这并不奇怪;我们的目标是让您了解 Spark 的各个方面，同时也说明了，Spark 是由许多不同的组件组成的。

考虑到您正在阅读这本书，您可能已经了解了一些关于 Apache Spark 的知识和它能做什么。尽管如此，在这一章中，我们想简单介绍一下 Spark 背后的主要理念以及 Spark 产生的环境 (为什么每个人都突然对并行数据处理感到兴奋?) 和它的历史。我们还将概述运行 Spark 的前几个步骤。

1.1. Apache Spark 的哲学

让我们对 Apache Spark 的各个关键部分进行分别描述。

统一

Spark 的关键驱动目标是为编写大数据应用程序提供一个统一的平台。统一是什么意思？Spark 的设计目的是支持广泛的数据分析任务，从简单的数据加载和 SQL 查询到机器学习和流计算，这些都通过相同的计算引擎和一致的 api 集合实现。这一目标背后的主要见解是，现实世界的数据分析任务——无论它们是工具中的交互式分析，还是用于生产应用程序的传统软件开发——都倾向于结合许多不同的处理引擎类型和库。

Spark 的统一特性使得这些任务的编写更加简单和高效。首先，Spark 提供了一致的、可组合的 api，您可以使用这些 api 从较小的部分或现有的库中构建应用程序。它还使您可以轻松地基于这些 API 编写自己的分析库。然而，可组合的 api 是不够的：Spark 的 api 也被设计为通过优化在用户程序中组合的不同库和函数来实现高性能。例如，如果您使用 SQL 查询加载数据，然后使用 Spark 的 ML 库评估机器学习模型，那么 Spark 计算引擎可以将这些步骤合并到一个步骤中，来扫描数据。通用 api 和高性能执行的结合，无论您如何组合它们，都使 Spark 成为交互式和生产应用程序的强大平台。

Spark 专注于定义一个统一的平台，这与其他软件领域的统一平台的想法是一致的。例如，当进行建模时，数据科学家受益于一组统一的库 (例如，Python 或 R)，而 web 开发人员则受益于诸如 Node.js 这样的统一框架。在 Spark 之前，没有一个开源系统试图提供这种类型的统一引擎来进行并行数据处理，这意味着用户应用程序中需要整合多个 api 来完成

一项任务。因此，Spark 很快成为了这种类型开发的标准。随着时间的推移，Spark 继续扩展其内置的 api，以覆盖更多的工作任务。与此同时，该项目的开发人员继续完善其统一引擎的主题。特别是，本书的一个重点将是在 Spark 2.0 中提供的 “structured api” (DataFrames、dataset 和 SQL)，structured api 提供了在用户应用程序下进行更强大的优化。

计算引擎

Spark 在打造一个统一平台的同时，它小心地将其范围限制在计算引擎上。（**有所为有所不为**）意思是，Spark 处理从存储系统加载数据并在其上执行计算，但最终数据并不永久存储在 Spark 中。Spark 可以和多种存储系统结合使用，如 Kafka、HBase、Hive、HDFS 以及关系型数据库。这样做的原因是，大多数数据已经存在于现有存储系统中。数据移动成本非常昂贵，所以 Spark 关注于对数据进行计算，不管数据在哪里。在面向用户的 api 中，Spark 努力使这些存储系统看起来非常相似，因此应用程序不必担心数据的位置。

Spark 对计算的关注使得它有别于早期的大数据软件平台，比如 Apache Hadoop。Hadoop 包括一个存储系统(Hadoop 文件系统 HDFS，用低成本的商品服务器集群做存储)和一个紧密集成的计算引擎(MapReduce)。Hadoop 这种设计在某些场景下会出现难以抉择的问题，如：如果只使用计算引擎 MapReduce，而不使用存储 HDFS，此时无法割裂两者，只能同时安装。

尽管 Spark 在 Hadoop 存储上运行得很好，但今天它在没有 Hadoop 的环境中也广泛使用。如 Spark+kafka 联合起来，进行流处理。

函数库

Spark 的最终组件是它的库，它以统一引擎的设计为基础，为公共数据分析任务提供统一的 API。Spark 既支持使用内置的标准库，也支持由开源社区发布为第三方包的大量外部库。如今，Spark 的标准库实际上是 Spark 开源项目的主要部分: Spark 核心计算引擎自发布以来几乎没有变化，但是，函数库已经提供了越来越多的功能类型。Spark 包括用于 SQL 和结构化数据的库(Spark SQL)、机器学习(MLlib)、流处理(Spark 流和新的结构化流处理 Structured Streaming)和图形分析(GraphX)。除了这些库之外，还有数百个开放源代码的外部库，从各种存储系统的连接器到机器学习算法。在 spark-packages.org 上有一个外部库索引。

1.2. Spark 产生的背景:大数据问题

为什么我们首先需要一个新的引擎和编程模型来进行数据分析？与计算机的许多趋势一样，这是由于计算机应用程序和硬件构成的经济因素发生了变化。

随着处理器速度的提升，计算机的运行速度也在增加：因此，应用程序每年也会自动变得更快，而不需要对代码进行任何修改。随着时间的推移，这种趋势导致了一个庞大而成熟的应用程序生态系统，其中大部分应用程序都设计为只在单个处理器上运行。这些应用程序采用了改进的处理器速度的趋势，以便随着时间的推移扩展到更大的计算和更大的数据量。

不幸的是，这种硬件的趋势在 2005 年左右停止了：由于在散热方面的严格限制，硬件开发人员停止让单个处理器的速度更快，转而使用相同的速度增加更多的并行 CPU 内核。这种变化意味着需要修改应用程序以增加并行性，以便更快地运行，这为新的编程模型(如 Apache Spark)创造了舞台。

与此同时，存储成本也在下降，可以获得的数据量在增加，如随着互联网的发展，视频数据、图像数据随处可见。最终结果是，搜集了大数据量的内容，处理这些内容需要大的、并行度高的计算引擎，通常需要运行在集群之上。

此外，过去 50 年开发的软件不能自动伸缩，传统的数据处理程序的编程模型也不能满足新的编程模型的需求。

Apache Spark 就是为当前这个环境而生的。

1.3. Spark 历史(见实验楼课程)

1.4. Spark 的现在和未来

Spark 已经存在了许多年，仍然是当前最流行的大数据计算框架，使用 Spark 的公司和项目都在不断增加。Spark 本身也在不断改进，新功能不断增加，例如，2016 年引入了一种新的高级流媒体引擎，即 Structured Streaming 结构化流处理。

Spark 将继续成为在可预见的未来进行大数据分析的公司的基石，尤其是考虑到该项目仍在快速发展。任何需要解决大数据问题的数据科学家或工程师都可能需要在他们的机器上安装一个 Spark。

1.5. 运行 Spark

本课程包含大量的与 spark 相关的代码，并且很重要的一点是，您需要在学习过程中运行它。在大多数情况下，您需要交互式地运行代码，以便可以进行试验。

您可以使用 Python、Java、Scala、R 或 SQL 与 Spark 进行交互。Spark 本身是用 Scala 编写的，并在 Java 虚拟机(JVM)上运行，因此在您的笔记本或集群上运行 Spark，您所需

要的只是安装 Java 环境。如果您想要使用 Python API，您还需要一个 Python 解释器(版本 2.7 或更高版本)。如果你想使用 R，你需要在你的机器上安装 R 语言环境。

有两种选择，我们建议开始使用 Spark: 在您的笔记本电脑上下载并安装 Apache Spark。或者在 Databricks Community Edition 中运行基于 web 的版本，这是一个学习 Spark 的免费云环境，其中包含了本书中的代码。我们接下来解释这两个选项。

1.5.1. 下载 Spark 到本地

如果您想在本地下载并运行 Spark，第一步是确保您的机器上安装了 Java(可用 Java)，以及 Python 版本，如果您想使用 Python 的话。接着，打开 Spark 官方网站 <http://spark.apache.org/downloads.html>，选择安装包的版本和类型“Pre-built for Hadoop 2.7 and later”，然后点击下载链接。此时会下载一压缩包文件，需要解压它。这本书的大部分是用 Spark 2.2 编写的，所以下载版本 2.2 或以后应该是一个很好的起点。

1.5.1.1. 为 Hadoop 集群下载对应的 Spark。

Spark 可以在本地运行，不需要任何分布式存储系统，比如 Apache Hadoop。但是，如果您想将您的笔记本上的 Spark 版本连接到 Hadoop 集群，请确保您下载了该 Hadoop 版本对应的 Spark 版本。我们在后面的章节中讨论了 Spark 是如何在集群和 Hadoop 文件系统上运行的，但是现在我们建议在您的笔记本上运行 Spark。

1.5.1.2. 从源代码构建 Spark

我们不会在书中介绍这一点，但是您也可以从源代码构建和配置 Spark。您可以在 ApacheSpark 下载页面上选择一个源代码包，以获取源代码，并按照 README 文件中的说明进行构建。

在您下载了 Spark 之后，您将希望打开一个命令行提示符并提取该包。在我们的例子中，我们安装了 Spark 2.2。下面是一个代码片段，您可以在任何 unix 风格的命令行上运行，以解压缩从 Spark 下载的文件并进入解压后的目录：

```
cd ~/Downloads  
  
tar -xf spark-2.2.0-bin-hadoop2.7.tgz  
  
cd spark-2.2.0-bin-hadoop2.7.tgz
```

注意，Spark 在项目中有大量的目录和文件。不要被吓倒！这些目录中的大多数只有在您阅读源代码时才有意义。下一节将讨论最重要的方向——让我们启动 Spark 的控制台以进行交互使用。

1.5.2. 启动 Spark 交互式控制台

您可以在 Spark 中为几种不同的编程语言启动交互式 shell。本书的大部分内容是用 Python、Scala 和 SQL 编写的；因此，这些是我们推荐的出发点。

1.5.2.1. 启动 Python 控制台

为了启动 Python 控制台，您需要安装 Python 2 或 3。从 Spark 的主目录，运行以下代码：

```
./bin/pyspark
```

完成之后，输入“spark”并按 Enter 键。您将看到打印的 SparkSession 对象，我们将在第 2 章中介绍它。

1.5.2.2. 启动 Scala 控制台

要启动 Scala 控制台，您需要运行以下命令：

```
./bin/spark-shell
```

完成之后，输入“spark”并按 Enter 键。与在 Python 控制台中看到的一样，您将看到 SparkSession 对象，我们将在第 2 章中介绍它。

1.5.2.3. 启动 SQL 控制台

本书的部分内容将涵盖大量的 Spark SQL。对于那些，您可能想要启动 SQL 控制台。在我们讨论了这些主题之后，我们将重新讨论一些更相关的细节。

```
./bin/spark-sql
```

1.5.2.4. 在这本书中使用的数据

在本书中，我们将使用一些数据源作为示例。如果您想在本地运行代码，可以从本书的官方代码库中下载这些代码。简而言之，您将下载数据，将其放入一个文件夹中，然后在本书中运行代码片段！

第二章 Spark 简介

现在我们关于 Apache Spark 的历史经验已经完成，现在是开始使用和应用它的时候了！

本章对 Spark 进行了一个简要的介绍，其中我们将介绍集群的核心架构、Spark 应用程序和使用 DataFrames 和 SQL 的 Spark 的结构化 api。在此过程中，我们将接触 Spark 的核心术语和概念，以便您可以立即使用 Spark。让我们从一些基本的背景信息开始。

2.1. Spark 的基本架构

通常，当你想到一台“计算机”时，你会想到一台机器在你家里或工作的桌子上。这台机器非常适合看电影或使用电子表格软件。但是，许多用户可能在某个时间点上体验过，有些东西是您的计算机没有足够的能力来运行的。一个特别具有挑战性的领域是数据处理。单个机器没有足够的能力和资源来执行大量的信息(或者用户可能没有时间等待计算完成)。计算机的集群或组，将许多计算机的资源集合在一起，使我们能够像使用一台计算机一样使用所有的累积资源。现在，一组机器本身并不强大，您需要一个框架来协调它们之间的工作。Spark 就是这样做的，它管理和协调跨集群计算机上的数据执行任务。

Spark 将用于执行任务的集群交由集群管理器管理，如 Spark Standalone 集群管理器、YARN 集群管理器、Mesos 集群管理器。然后，我们向这些集群管理器提交 Spark 应用程序，它将为我们的应用程序提供资源，以便我们能够完成我们的工作。

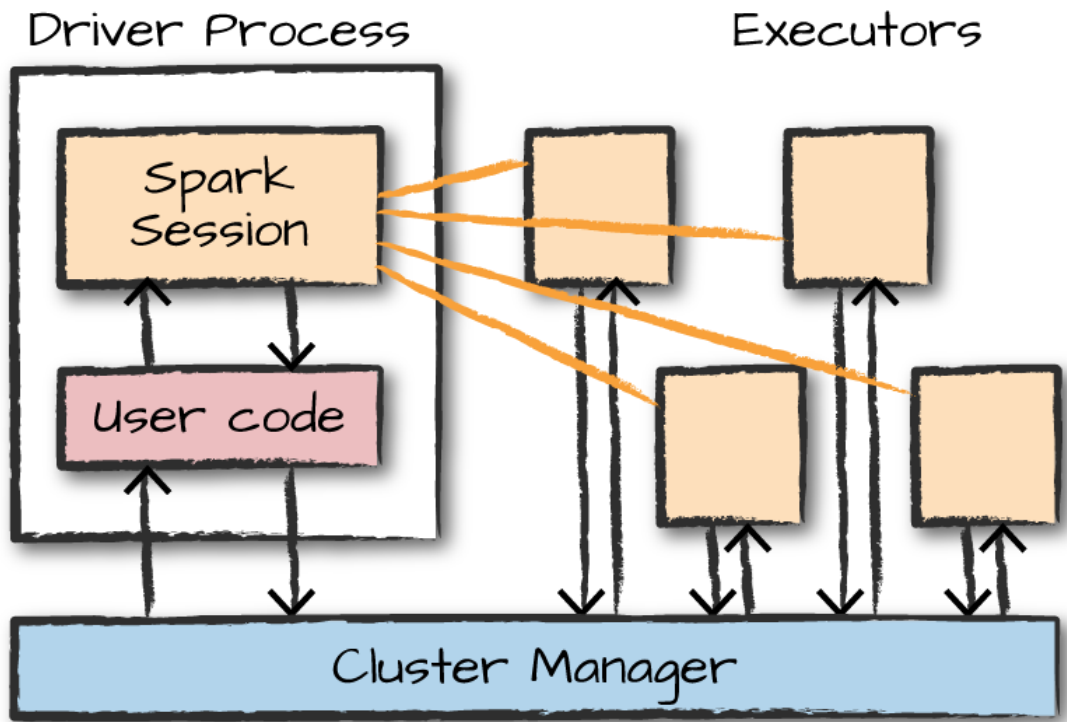
2.1.1. Spark 应用程序

Spark 应用程序由一个 driver 进程(驱动程序)和一组 Executor 进程组成。driver 进程负责运行你的 main 函数，此进程位于集群中的一个节点上。负责三件事：维护有关 Spark 应用程序的信息；响应用户的程序或输入；分析、分配和调度 executor 的工作(稍后讨论)。

driver 驱动程序是非常重要的。它是 Spark 应用程序的核心，并在应用程序的生命周期内维护所有相关信息。

executors 进程实际执行 driver 分配给他们的工作。这意味着每个 executor 只负责两件事: 执行由驱动程序分配给它的代码。并将执行器 executor 的计算状态报告给驱动节点。

图 2-1 演示了集群管理器如何控制物理机器并分配资源给 spark 应用程序。这可以是三个核心集群管理器之一: Spark' s standalone cluster manager, YARN, 或者 Mesos。。这意味着可以同时运行多个 Spark 应用程序。我们将在第四部分讨论集群管理器。



在图 2-1 中，我们可以看到左边的驱动程序和右边的四个执行器。在这个图中，我们删除了集群节点的概念。用户可以通过配置指定每个节点上有多少个执行器。

注意

Spark 除了集群 cluster 模式之外，还具有本地 local 模式。driver 驱动程序和 executor 执行器是简单的进程，这意味着它们可以在同一台机器或不同的机器上运行。在本地 local 模式中，驱动程序和执行程序(作为线程)在您的个人计算机上运行，而不是集群。我们写这本书时考虑了本地模式，所以你应该能够在一台机器上运行所有的东西。

以下是当前需要掌握的理解 Spark 应用程序的要点:

- Spark 使用一个集群管理器来跟踪可用的资源。
- driver 驱动程序负责执行驱动程序的命令，在 executor 执行器中完成给定的任务。

在大多数情况下，executor 执行器将始终运行 Spark 代码。但是，通过 Spark 的语言 api，驱动程序可以由许多不同的语言“驱动”。让我们来看看下一节的内容。

2.2. Spark 语言 API

Spark 的语言 api 使您可以使用各种编程语言运行 Spark 代码。在很大程度上，Spark 在每种语言中都呈现了一些核心的“概念”；然后将这些概念转换成在机器集群上运行的 Spark 代码。如果只使用结构化 api，那么所有语言都具有类似的性能特征。这里有一个简短的纲要：

Scala

Spark 主要是用 Scala 编写的，这使它成为 Spark 的“默认”语言。本书将包括 Scala 代码示例。

Java

尽管 Spark 是用 Scala 编写的，Spark 的作者们还是小心翼翼地确保您可以在 Java 中编写 Spark 代码。本书将主要关注 Scala，但将提供与之相关的 Java 示例。

Python

几乎所有的特性都支持 Python 语言。只要我们包含 Scala 代码示例和 Python API，本书就会包含 Python 代码示例。

R 语言

Spark 有两个常用的 R 库。一个作为 Spark core (SparkR)的一部分，另一个作为 R 社区驱动的包(sparklyr)。我们在第 32 章中讨论了这两个库的集成。

图 2-2 给出了这种关系的简单说明。

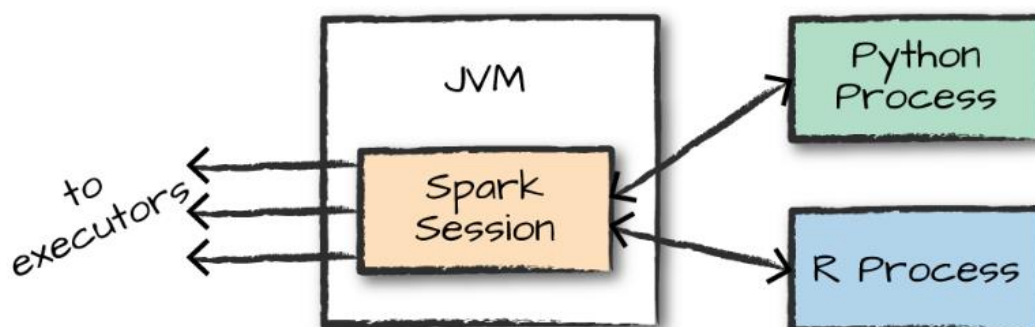


Figure 2-2. The relationship between the SparkSession and Spark's Language API

每个语言 API 都维护我们前面描述的相同的核心理念。用户可以使用 SparkSession 对象，这是运行 Spark 代码的入口点。当使用来自 Python 或 R 的 Spark 时，您不会编写显式的 JVM 指令；相反，您编写的 Python 和 R 代码可以将 Spark 转换为代码，然后可以在 executor jvm 上运行。

Spark's APIs

尽管您可以从各种语言中驱动 Spark，但它在这些语言中提供的功能还是值得一提的。

Spark 有两个基本的 api 集: low-level 的“非结构化”api，以及 higher-level 高级的结构化 api。我们将在本书中讨论这两个问题，但是这些介绍性的章节将主要讨论更高层次的结构化 api。

2.3. 启动 Spark

到目前为止，我们讨论了 Spark 应用程序的基本概念。这在本质上都是概念性的。当我们真正着手编写 Spark 应用程序时，我们需要一种方法来将用户命令和数据发送给 spark，让其为我们计算结果。我们首先创建一个 SparkSession。

注意

为此，我们将启动 Spark 的本地模式，就像我们在第 1 章中所做的那样。这意味着运行 `./bin/spark-shell` 访问 Scala 控制台以启动交互式会话。您还可以使用 `./bin/pyspark` 启动 Python 控制台。这将启动一个交互式 Spark 应用程序。还有一个方式，可以提交独立的应用程序到 Spark，称为 `Spark-submit`，这样您就可以提交一个预编译的应用程序到 spark 集群。我们会在第三章中告诉你们怎么做。

当您在此交互模式中启动 Spark 时，您将隐式地创建一个 `SparkSession` 来管理 Spark 应用程序。当您通过一个独立的应用程序启动它时，您必须在应用程序代码中创建 `SparkSession` 对象。

2.4. SparkSession

正如本章开头所讨论的，您通过一个名为 `SparkSession` 的驱动程序控制您的 Spark 应用程序。`SparkSession` 实例是 Spark 在集群中执行用户定义操作的方式。在 Scala 和 Python 中，当您启动控制台时，`SparkSession` 被实例化为 `spark` 变量，可以直接使用。让我们来看看 Scala 和/或 Python 中的 `SparkSession`：

在刚启动的 Scala 控制台中输入 `spark`，您应该看到如下内容：

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@...
```

在刚启动的 Python 控制台中输入 `spark`，您应该看到如下内容：

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

现在让我们执行创建一系列数字的简单任务。这一系列数字就像电子表格中的一个命名列：

```
// in Scala
```

```
val myRange = spark.range(1000).toDF("number")
```

```
# in Python
```

```
myRange = spark.range(1000).toDF("number")
```

你刚刚运行了你的第一行 spark 代码! 我们创建了一个 DataFrame, 其中一个列包含 1000 行, 值从 0 到 999。这一系列数字代表一个分布式集合。当在一个集群上运行时, 这个范围的每个部分都存在于一个不同的 executor 上。这是一个 Spark DataFrame。

2.5. DataFrames

DataFrame 是最常见的结构化 API, 它只是表示包含行和列的数据表。定义列和列类型的列表称为 *schema* (模式)。您可以将 DataFrame 看作是带有指定列的电子表格。

图 2-3 说明了基本的区别: 位于一台计算机上的电子表格, 存在一个特定位置上。而 Spark DataFrame 可以跨越数千台计算机。把数据放在一台以上电脑上的原因应该是直观的: 要么是数据太大而无法安装在一台机器上, 要么就是花费太长时间在一台机器上执行计算。

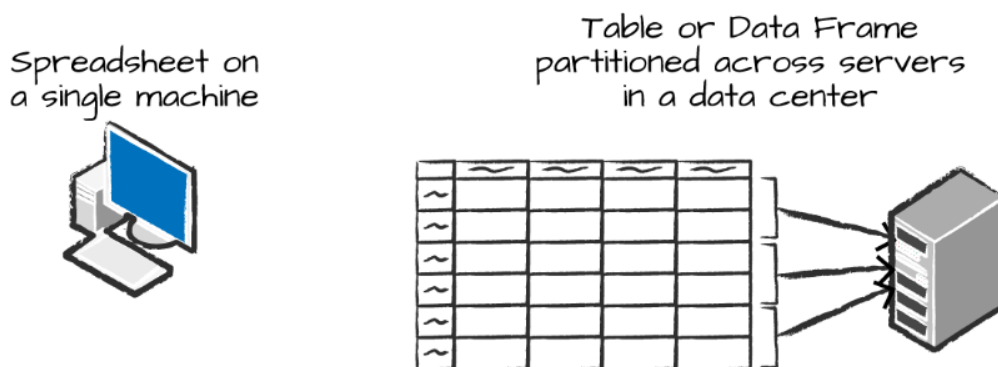


Figure 2-3. Distributed versus single-machine analysis

DataFrame 的概念并不是 Spark 特有的。R 和 Python 都有类似的概念。然而, Python/R DataFrames(有一些例外)存在于一台机器上, 而不是多台机器上。这限制了给定的 DataFrame 只能使用某一特定机器上存在的资源。但是, 因为 Spark 具有 Python

和 R 的语言接口。很容易将 Pandas (Python)的 DataFrames、R DataFrames 转换为 Spark DataFrames。

注意

Spark 有几个核心抽象: Datasets、DataFrames、SQL 表和弹性分布式数据集(RDDs)。这些不同的抽象都表示数据的分布式集合。最简单和最有效的是 DataFrames，它在所有语言中都可用。我们在第二部分的末尾学习 dataset，在第三部分中学习 RDDs。

2.5.1. Partitions

为了使每个 executor 执行器并行执行任务，Spark 将数据分解成块，这些数据块称为 partition（分区）。一个分区是集群中的一个物理机器上的行集合。DataFrame 的分区表示了在执行过程中数据是如何在机器集群中物理分布的。如果您有一个分区，Spark 将只有一个并行任务，即使您有数千个 Executor 执行器。如果您有多个分区，但只有一个 Executor 执行器，Spark 仍然只有一个并行任务，因为只有一个计算资源。

需要注意的一件重要的事情是，对于 DataFrames，您不(大多数情况下)手动或单独操作分区。您只需在物理分区中指定数据的高级转换，Spark 将确定该工作将如何在集群上执行。底层 api 确实存在(通过 RDD 接口)，我们将在第 3 部分中介绍这些 api。

2.6. Transformations

在 Spark 中，核心数据结构是不可变的，这意味着它们在创建之后无法更改。乍一看，这似乎是个奇怪的概念: 如果你不能改变它，你应该如何使用它? 要“更改”一个 DataFrame，您需要指导 Spark 如何修改它以实现您想要的功能。这些指导指令称为 Transformations 转换。让我们执行一个简单的转换，以在当前的 DataFrame 中找到所有偶数:

```
// in Scala
```

```
val divisBy2 = myRange.where("number % 2 = 0")
```

```
# in Python
```

```
divisBy2 = myRange.where("number % 2 = 0")
```

注意，这些返回没有输出。这是因为我们只声明了一个抽象转换 `where`。Spark 将不会对转换进行操作，直到我们调用一个 `action` 操作(我们将在稍后讨论它)。Transformations 转换是使用 Spark 表达业务逻辑的核心方法。有两种类型的转换:窄依赖的转换、宽依赖的转换。

窄依赖的转换是每个输入数据分区只对一个数据输出分区。在前面的代码片段中，`where` 语句指定了一个窄依赖。其中一个输入分区最多一个输出分区，如图 2-4 所示：

Narrow transformations 1 to 1

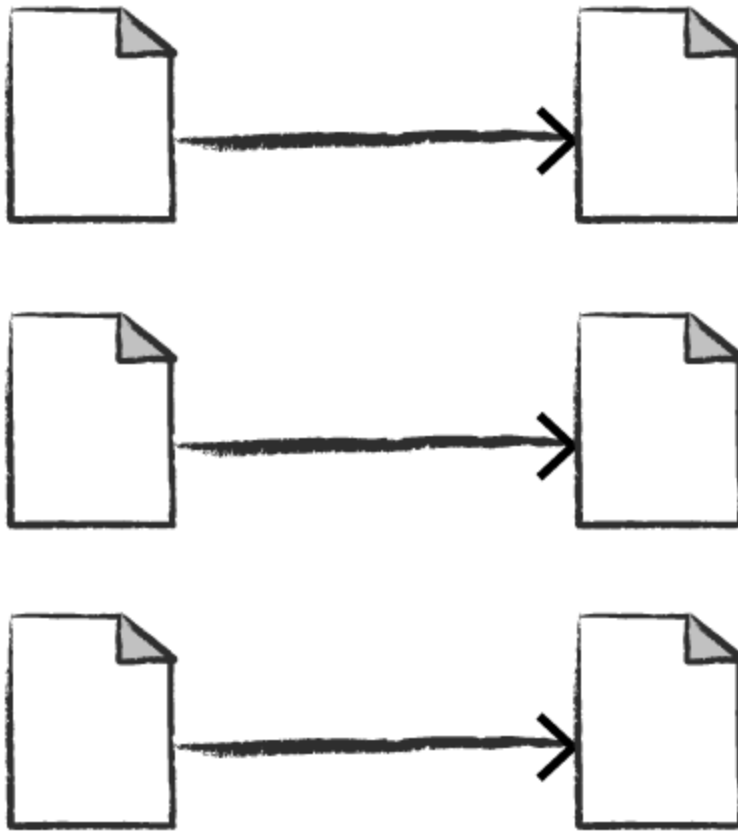


Figure 2-4. A narrow dependency

宽依赖的转换，一个输入数据分区对应多个输出分区。您经常会听到 shuffle 这样的说法，即 Spark 将在集群中交换分区中的数据。窄依赖转换，Spark 将自动执行称为流水线 pipeline 的操作。这意味着如果我们在 DataFrames 上指定多个过滤器。它们都将在内存中执行，不会产生 shuffle。当我们进行 shuffle 洗牌时，Spark 将结果写入磁盘。图 2-5 显示宽依赖转换：

Wide transformations (shuffles) 1 to N

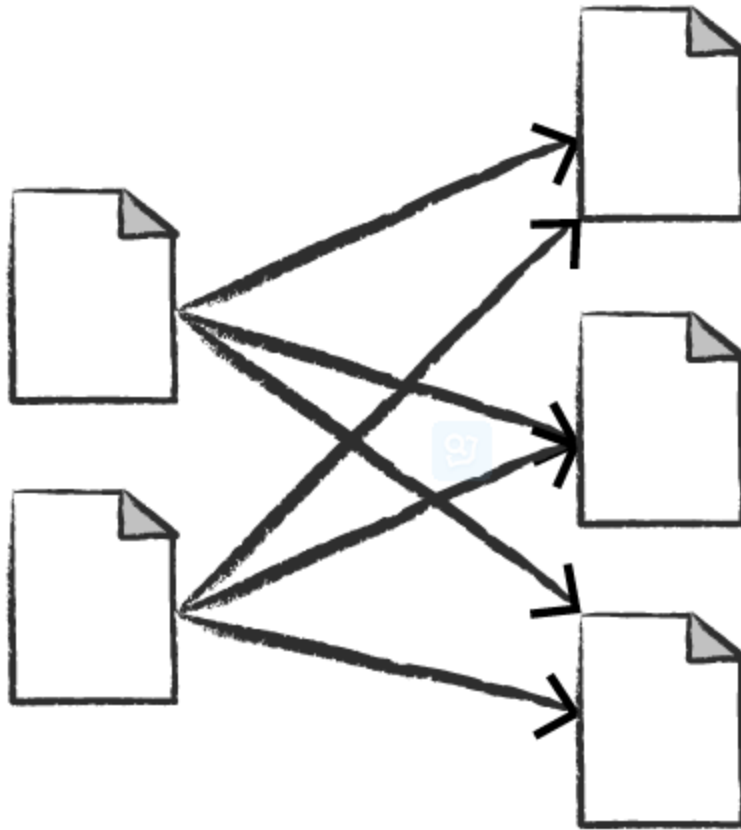


Figure 2-5. A wide dependency

您在网络上会看到许多关于 shuffle 优化的讨论，因为这是一个重要的主题，但是现在，您需要了解的是，有两种类型的转换。现在，您可以看到转换是如何简单地指定不同的数据操作。这就引出了一个叫 Lazy Evaluation 延迟计算的话题。

2.6.1. Lazy Evaluation

延迟 evaluation 意味着 Spark 将等到最后一刻才执行计算一些列指令。在 Spark 中，不会在执行某个转换操作时立即修改数据，spark 会构建了一个您想要应用于您的源数据的转换计划。通过等待直到最后一分钟执行代码，Spark 将这个计划从原始的 DataFrame

转换到一个流线型的物理计划，该计划将在整个集群中尽可能高效地运行。这提供了巨大的好处，因为 Spark 可以从端到端优化整个数据流。其中一个例子是 DataFrames 上的谓词下推 pushdown。如果我们构建一个大型的 Spark 作业，但在最后指定一个过滤器，只需要我们从源数据中获取一行。最有效的执行方式是访问我们需要的单个记录。Spark 实际上是通过自动将过滤器向下推来优化它的。

2.7. Actions

transformation 转换允许我们构建逻辑转换计划。为了触发计算，我们运行一个 action 操作。action 操作指示 Spark 通过执行一系列 transformation 转换计算结果。最简单的 action 操作是 count，它给出了 DataFrame 中记录的总数：

```
divisBy2.count()
```

前面代码的输出应该是 500。然而，count 并不是唯一的 action 操作。有三种类型的 action：

- 在控制台中查看数据的 action
- 数据收集的 action 操作。
- 输出到第三方存储系统的 action 操作。

在指定这个 count 操作时，我们启动了一个 Spark job，运行我们的过滤器 filter 转换(一个窄依赖转换)，然后是一个聚合(一个宽依赖转换)，它在每个分区基础上执行计数，然后是一个收集 action，它将我们的结果 driver 端。通过检查 Spark UI，您可以看到所有这一切。Spark UI 是一个包含在 Spark 中的工具，您可以通过它监视在集群上运行的 Spark 作业。

2.8. Spark UI

您可以通过 Spark web UI 监视作业的进度。Spark UI 在 driver 节点的 4040 端口上可用。如果在本地模式下运行，则将是 `http://localhost:4040`。Spark UI 显示关于您的 Spark 作业状态、环境和集群状态的信息。它非常有用，特别是对于调优和调试。图 2-6 展示了一个 Spark job 的示例 UI，其中执行了包含 9 个任务的两个阶段。

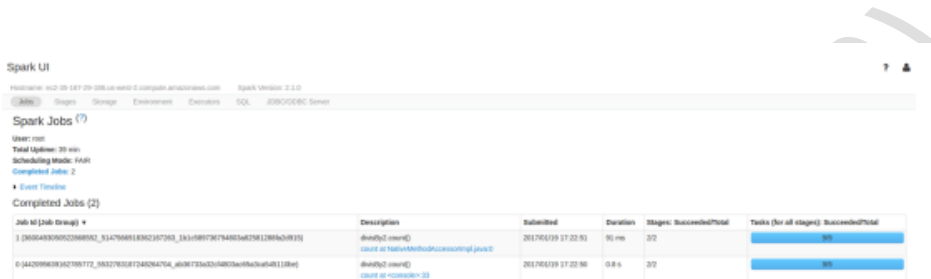


Figure 2-6. The Spark UI

本章将不会详细讨论 Spark 作业执行和 Spark UI。我们将在第 18 章讨论这个问题。此时，您需要了解的是，Spark job 作业表示由单个 action 触发的一组转换，您可以从 Spark UI 监视该作业。

2.9. 一个相对完整的例子

在前面的示例中，我们创建了一个包含一系列数字的 DataFrame；这并不是什么突破性的大数据。在这一节中，我们将用一个更现实的例子来巩固我们在本章中所学到的所有内容，并逐步解释在幕后发生的事情。我们将用 Spark 来分析美国运输统计局的一些飞行数据。

在 CSV 文件夹中，您将看到我们有许多文件。还有一些其他包含不同文件格式的文件夹，我们将在第 9 章中讨论。现在，让我们关注 CSV 文件。

每个文件都有许多行数据。这些文件都是 CSV 文件，这意味着它们是一种半结构化的数据格式，文件中的每一行表示将来的 DataFrame 中的一行：

```
$ head /data/flight-data/csv/2015-summary.csv

DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
```

Spark 具有从大量数据源中读写数据的能力。要读取这些数据，我们将使用与我们的 SparkSession 关联的 DataFrameReader 对象。在这样做时，我们将指定文件格式以及我们想要指定的任何选项。在我们的例子中，我们想做一个叫做 schema inference（模式推理）的东西，这意味着我们希望 Spark 能够对 DataFrame 的 schema（模式）进行最好的猜测。我们还希望指定第一行是文件的头，因此我们也将指定它作为一个选项。

为了获得模式信息，Spark 会读取一些数据，然后根据 Spark 中可用的类型尝试解析这些行中的类型。在读取数据时，您还可以选择严格地指定模式(在生产场景中，我们建议这样做)：

```
// in Scala
val flightData2015 = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/data/flight-data/csv/2015-summary.csv")

# in Python
flightData2015 = spark\
  .read\
```

```
.option("inferSchema", "true")\n.option("header", "true")\n.csv("/data/flight-data/csv/2015-summary.csv")
```

每个 DataFrames(在 Scala 和 Python 中)都有一组列，其中列的数据行数不确定。行数不确定的原因是读取数据是一个 transformation 转换操作，因此是一个延迟操作。Spark 只查看了几行数据，试图猜测每个列应该是什么类型。图 2-7 提供了将被读入 DataFrame 的 CSV 文件的示例，然后将其转换为本地数组或行列表。



Figure 2-7. Reading a CSV file into a DataFrame and converting it to a local array or list of rows

如果我们在 DataFrame 上执行操作，我们将能够看到我们在使用命令之前看到的相同结果：

```
flightData2015.take(3)\n\nArray([United States,Romania,15], [United States,Croatia...
```

让我们指定一些更多的转换！现在，让我们根据 count 列对数据进行排序，这是一个整数类型。图 2-8 说明了这一过程。

注意

记住，sort 不会修改 DataFrame。我们使用 sort 作为 transformation 转换操作，通过转换之前的 DataFrame 返回一个新的 DataFrame。让我们来说明当我们在新 DataFrame 上调用 take 转换方法时，发生了什么(图 2-8)。

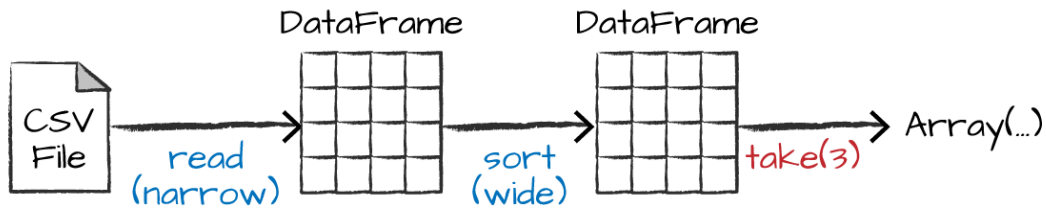


Figure 2-8. Reading, sorting, and collecting a DataFrame

当我们调用 `sort` 时，数据不会发生任何变化，因为它只是一个转换。但是，我们可以看到 Spark 正在构建一个计划，通过 `explain` 查看计划，可以看到 spark 将如何跨集群执行这个计划。

```
flightData2015.sort("count").explain()
```

```
== Physical Plan ==
```

```
*Sort [count#195 ASC NULLS FIRST], true, 0
```

```
+ Exchange rangepartitioning(count#195 ASC NULLS FIRST, 200)
```

```
+ FileScan csv
```

```
[DEST_COUNTRY_NAME#193,ORIGIN_COUNTRY_NAME#194,count#195] ...
```

恭喜你，你刚刚读了你的第一个解释计划！解释计划有点神秘，但稍加练习就会变成第二技能。你可以从上到下阅读解释计划，顶部是最终结果，底部是数据的来源。在本例中，查看每行的第一个关键字。你将会看到 `sort`, `exchange`, 和 `FileScan` 三个关键字。因为数据排序实际上是一个宽依赖转换，因为位于不同分区中的数据行需要相互比较。在这一点上，不要过于担心理解所有的解释计划，它们只是帮助你调试和提高你的知识的工具。

现在，就像我们之前做的那样，我们可以指定一个 `action` 来启动这个执行计划。然而，在此之前，我们将设置一个配置项。默认情况下，当我们执行 `shuffle` 时，Spark 会输出 200 个 `shuffle` 分区。让我们将这个值设为 5，以减少来自 `shuffle` 的输出分区的数量：

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

```
flightData2015.sort("count").take(2)
```

```
... Array([United States,Singapore,1], [Moldova,United States,1])
```

图 2-9 演示了这个操作。请注意，除了逻辑转换之外，我们还包括物理分区计数。

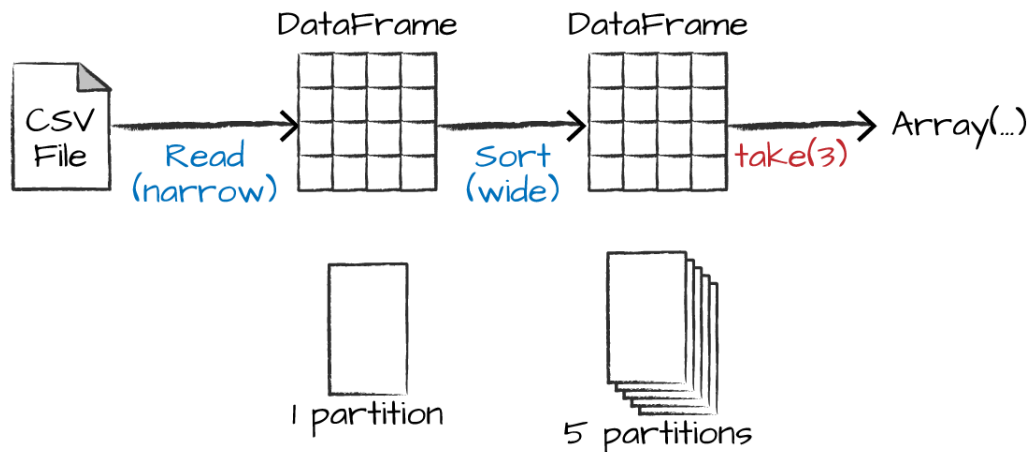


Figure 2-9. The process of logical and physical DataFrame manipulation

我们构建的转换的逻辑计划为 DataFrame 定义了一个血统关系，以便在任何给定的时间点，Spark 都知道如何通过执行之前在相同输入数据上执行的所有操作来重新计算任何分区。这是 Spark 编程模型-函数式编程的核心，当数据的转换保持不变时，相同的输入总是会导致相同的输出。

我们不操纵物理数据;相反，我们通过类似于前面设置的 shuffle 分区参数来配置物理执行特性。我们最后得到了 5 个输出分区，因为这是在 shuffle 分区中指定的值。您可以更改它以帮助控制您的 Spark 作业的物理执行特性。继续尝试不同的值，并查看您自己的分区数量。在尝试不同的值时，您应该看到截然不同的运行时。请记住，您可以通过导航到 4040 端口上的 Spark UI 来监视工作进度，以查看作业的物理和逻辑执行特性。

2.9.1. DataFrames 和 SQL

我们在前面的示例中完成了一个简单的转换，现在让我们学习一个更复杂的例子，并在 DataFrames 和 SQL 中进行跟踪。Spark 可以使用完全相同的方式运行相同的转换，不管语言是什么。您可以在 SQL 或 DataFrames(在 R、Python、Scala 或 Java)中表达业务逻辑，Spark 将在实际执行代码之前将该逻辑编译成一个底层计划(您可以在 explain 计划中看到)。使用 Spark SQL，您可以将任何 DataFrame 注册为表或视图(临时表)，并使用纯 SQL 查询它。在编写 SQL 查询或编写 DataFrame 代码之间没有性能差异，它们都“编译”到我们在 DataFrame 代码中指定的相同的底层计划。

您可以通过一个简单的方法调用将任何 DataFrame 转换为一个表或视图：

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

现在我们可以用 SQL 查询我们的数据了。为此，我们将使用 spark.sql 函数(记住，spark 是我们的 SparkSession 变量)，它方便地返回一个新的 DataFrame。这使得您可以在任何给定的时间点以最方便的方式指定转换，而不牺牲任何效率来这样做！为了理解这一点，让我们来看看两个解释计划：

```
// in Scala

val sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

val dataFrameWay = flightData2015
```

```
.groupBy('DEST_COUNTRY_NAME)
```

```
.count()
```

```
sqlWay.explain
```

```
dataFrameWay.explain
```

```
# in Python
```

```
sqlWay = spark.sql("""
```

```
SELECT DEST_COUNTRY_NAME, count(1)
```

```
FROM flight_data_2015
```

```
GROUP BY DEST_COUNTRY_NAME
```

```
""")
```

```
dataFrameWay = flightData2015\
```

```
.groupBy("DEST_COUNTRY_NAME")\
```

```
.count()
```



```
sqlWay.explain()
```

```
dataFrameWay.explain()
```

```
== Physical Plan ==
```

```
*HashAggregate(keys=[DEST_COUNTRY_NAME#182],
```

```
functions=[count(1)])

+ - Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)

    + - *HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[partial_count(1)])

    + - *FileScan csv [DEST_COUNTRY_NAME#182] ...

== Physical Plan ==

*HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[count(1)])

+ - Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)

    + - *HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[partial_count(1)])

    + - *FileScan csv [DEST_COUNTRY_NAME#182] ...
```

请注意，这些计划编译成完全相同的底层计划!

让我们从我们的数据中找出一些有趣的统计数据。要理解的一点是，Spark 中的 DataFrames(和 SQL)已经有大量可用的操作。您可以使用和导入数百个函数来帮助您更快

地解决大数据问题。我们将使用 `max` 函数，来确定进出任何给定位置的最大航班数。这只是扫描 `DataFrame` 中相关列中的每个值，并检查它是否大于前面所看到的值。这是一个 `transformation`，因为我们可以有效地过滤到一行。让我们看看这是什么样子：

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)

// in Scala

import org.apache.spark.sql.functions.max

flightData2015.select(max("count")).take(1)

# in Python

from pyspark.sql.functions import max

flightData2015.select(max("count")).take(1)
```

很好，这是一个简单的例子，给出了 370,002 的结果。让我们执行一些更复杂的任务，并在数据中找到前五个目的地国家。这是我们的第一个多转换查询，因此我们将逐步进行。让我们从一个相当简单的 SQL 聚合开始：

```
// in Scala

val maxSql = spark.sql("""
    SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
    FROM flight_data_2015
    GROUP BY DEST_COUNTRY_NAME
    ORDER BY sum(count) DESC
    LIMIT 5
""")

maxSql.show()
```

in Python

```
maxSql = spark.sql("""  
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total  
FROM flight_data_2015  
GROUP BY DEST_COUNTRY_NAME  
ORDER BY sum(count) DESC  
LIMIT 5  
""")
```

```
maxSql.show()
```

```
+-----+-----+
```

```
|DEST_COUNTRY_NAME|destination_total|
```

```
+-----+-----+
```

◀	United States	411352
	Canada	8399
	Mexico	7140
	United Kingdom	2025

	Japan	1548
+-----+-----+		

现在，让我们移动到与语义相似但在实现和排序上略有不同的 DataFrame 语法。但是，正如我们提到的，两者的基本计划是一样的。让我们运行查询，并将其结果视为完整性检查：

// in Scala

```
import org.apache.spark.sql.functions.desc
```

```
flightData2015
```

```
.groupBy("DEST_COUNTRY_NAME")
```

```
.sum("count")
```

```
.withColumnRenamed("sum(count)", "destination_total")
```

```
.sort(desc("destination_total"))
```

```
.limit(5)
```

```
.show()
```

in Python

```
from pyspark.sql.functions import desc
```

```
flightData2015\
```

```
.groupBy("DEST_COUNTRY_NAME")\
```

```
.sum("count")\

.withColumnRenamed("sum(count)", "destination_total")\

.sort(desc("destination_total"))\

.limit(5)\

.show()
```

+-----+-----+

|DEST_COUNTRY_NAME|destination_total|

+-----+-----+

| United States| 411352|

| Canada| 8399|

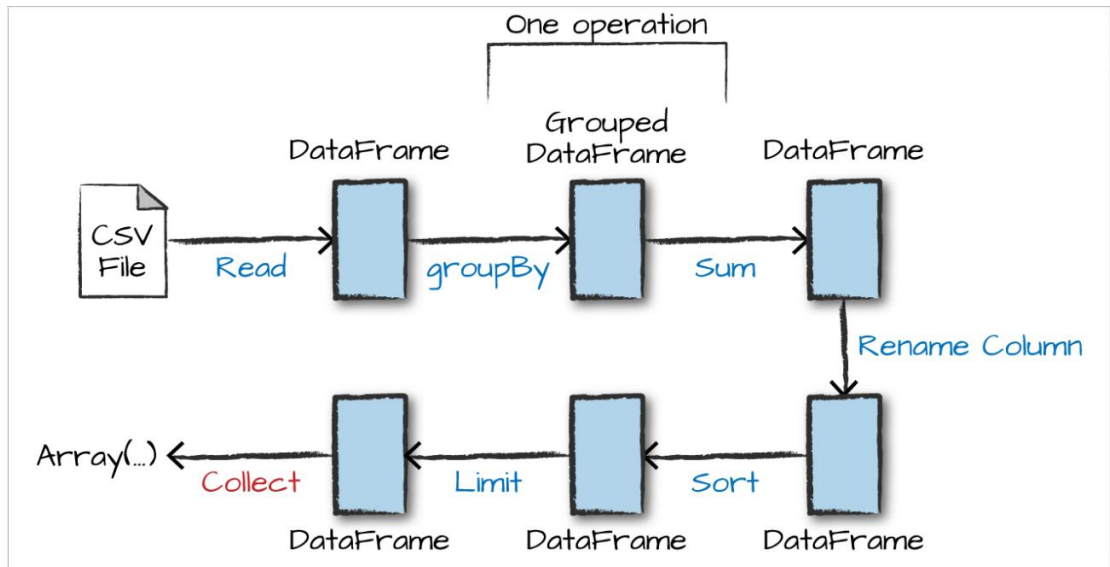
| Mexico| 7140|

◀ | United Kingdom| 2025|

| Japan| 1548|

+-----+-----+

现在有 7 个步骤可以让我们回到源数据。您可以在这些 DataFrames 的解释计划中看到这一点。图 2-10 显示了我们在“代码”中执行的步骤。真正的执行计划(在 explain 中可见)将与图 2-10 所示不同,因为物理执行进行了优化。然而,这是一个很好的起点。这个执行计划是一个有向无环图(DAG)的 transformation, 每一个 transformation 都产生一个新的不可变的 DataFrame, 我们调用一个 action 来生成结果。



第一步是读取数据。我们之前定义了 DataFrame, 但是, 作为提醒, Spark 实际上并没有读取它, 直到在 DataFrame 上调用了 action, 或者从原始 DataFrame 派生出一个 action。

第二步是分组; 当我们调用 groupBy, 我们最终 RelationalGroupedDataset, 这是一个有趣的名称, 用于 DataFrame 分组指定但需要用户指定一个聚合, 才能进一步查询。我们基本上指定了我们将被一个键(或一组键)分组, 现在我们要对这些键的每一个进行聚合。

因此, 第三步是指定聚合。让我们使用 sum 聚合方法。这需要输入一个列表达式, 或者, 简单地说, 是一个列名。sum 方法调用的结果是一个新的 DataFrame。您将看到它有一个新的模式, 但是它知道每个列的类型。需要再次强调的是: 截止目前还没有发生真正的计算。这只是已经表达的另一个 transformation, Spark 仅仅能够通过它跟踪我们的类型信息。

第四步是简单的重命名。我们使用 `withColumnRenamed` 方法，它接受两个参数，原始列名和新的列名。当然，此时仍然不会执行计算:这只是另一个 transformation 转换!

第五步对数据进行排序，这样如果我们在 `DataFrame` 的顶部获取结果，它们将在 `destination_total` 列中拥有最大的值。

那可能已经注意到，我们导入了一个函数 `desc`，来做这个排序。您可能还注意到，`desc` 不返回字符串，而是返回一个 `Column` 对象。通常，许多 `DataFrame` 方法将接受字符串 (作为列名称)或 `Column` 类型或表达式。`Column` 类型和表达式实际上是一样的。

最后，我们将指定一个 `limit`。这只是指定我们只想在最后的 `DataFrame` 中返回前 5 个值，而不是返回所有数据。

最后一步是我们的 action 操作。现在，我们开始真正收集 `DataFrame` 的结果，Spark 将返回我们正在执行的语言中的列表或数组。为了加强这一切，让我们看看前面的查询的解析计划:

```
// in Scala
flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)", "destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .explain()

# in Python
flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
```

```

.sort(desc("destination_total"))\
.limit(5)\
.explain()

== Physical Plan ==

TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC], outpu...
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial_sum...
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
            +- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NAME#7325L]
               +- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME#7579]

```

虽然这个解释计划与我们的“概念性计划”并不相符，但所有的部分都在那里。您可以看到 `limit` 语句以及 `orderBy`(在第一行中)。您还可以看到我们的聚合是如何在两个阶段中发生的，在 `partial_sum` 调用中。这是因为求和的数字列表是可交换的，而 Spark 可以按分区执行和。当然，我们也可以在 `DataFrame` 中看到我们的读取方式。

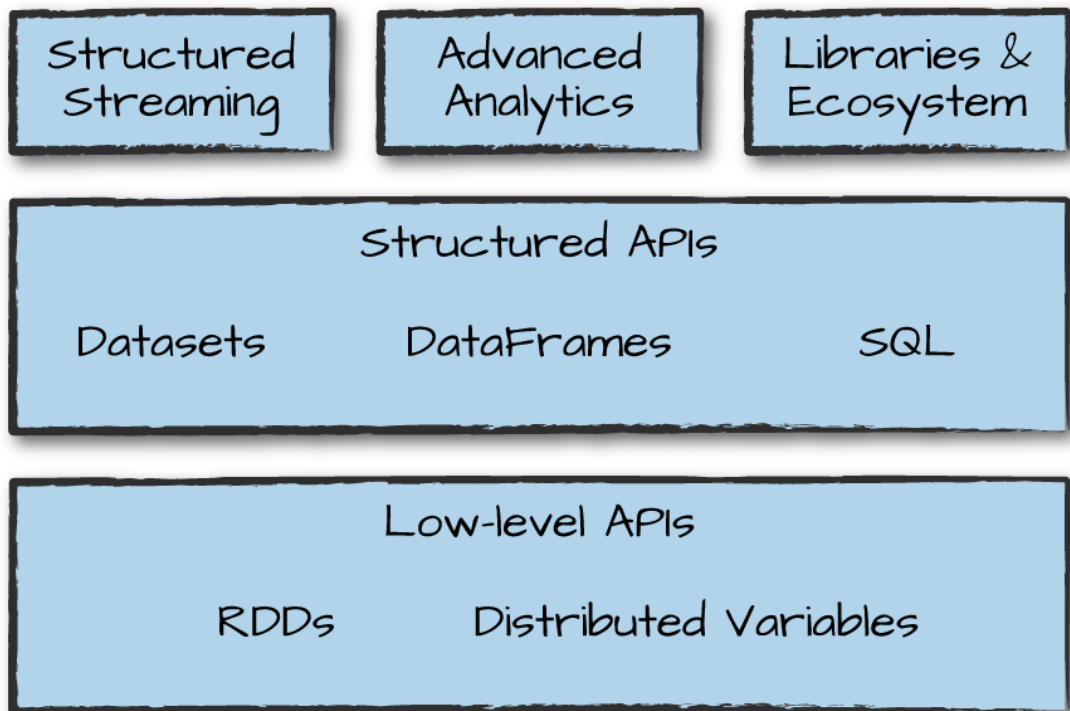
当然，我们并不总是需要收集数据。我们也可以将其写入任何 Spark 支持的数据源。例如，假设我们希望将信息存储在数据库中，比如 PostgreSQL，或者将它们写到另一个文件中。

2.10. 结论

本章介绍了 Apache Spark 的基本知识。我们讨论了 transformation 和 action，以及 Spark 如何延迟执行 DAG 的转换，以优化 DataFrames 的执行计划。我们还讨论了如何将数据组织成 partition 分区，为处理更复杂的转换设置 stage 阶段。在第 3 章中，我们将带您参观一个巨大的 Spark 生态系统，并查看一些更高级的概念和工具，这些概念和工具可以在 Spark 中使用，从流到机器学习。

第三章 Spark 工具集概览

在第 2 章中，我们在 Spark 的结构化 api 中引入了 Spark 的核心概念，比如 transformation 和 action 操作。这些简单的概念构建块是 Apache Spark 庞大的工具和库生态系统的基础(图 3-1)。Spark 是由这些原始的(底层 api 和结构化的 api)组成的，然后是一系列用于附加功能的标准库。



Spark 的库支持各种不同的计算任务，从图计算和机器学习到流处理，到与其他大量的计算机集群和存储系统集成。这一章主要介绍了 Spark 所提供的大部分内容，包括一些我们还没有介绍的 api，以及一些主要的库。对于每一部分，您将在本书的其他部分找到更详细的信息;我们的目的是为您提供一个可能的概述。

本章包括以下内容:

- 使用 spark-submit 运行生产应用程序。
- Dataset : 用于结构化数据的类型安全 api。
- 结构化流处理
- 机器学习和高级分析。

- 弹性分布数据集(RDD): Spark 的低级 api。
- SparkR
- 第三方包生态系统

在你游览完之后，你可以跳到书的相应部分，找到你关于特定主题的问题的答案。

生产应用程序运行

Spark 使开发和创建大数据程序变得很容易。Spark 还使您可以轻松地将交互式探索转换为带有 Spark - submit 的生产应用程序，这是一个内置的命令行工具。spark-submit 只做一件事:它允许您将您的应用程序代码发送到集群并启动它在那里执行。提交后，应用程序将运行，直到它退出(完成任务)或遇到错误。您可以使用 Spark 的所有支持集群管理器，包括 standalone、Mesos 和 yarn。

spark-submit 提供了几个控制项，您可以在其中指定应用程序需要的资源以及它应该如何运行以及它的命令行参数。

您可以在 Spark 的任何支持的语言中编写应用程序，然后提交它们执行。最简单的示例是在本地机器上运行应用程序。我们将通过运行一个带有 Spark 的示例 Scala 应用程序来展示这一点，在您下载 Spark 的目录中使用以下命令:

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local \
  ./examples/jars/spark-examples_2.11-2.2.0.jar 10
```

这个示例应用程序计算 pi 的数字到一定程度的估计。在这里，我们告诉 spark-submit，我们希望在本地机器上运行，我们希望运行哪个类和哪个 JAR，以及这个类的一些命令行参数。

我们还可以使用以下命令运行应用程序的 Python 版本:

```
./bin/spark-submit \
  --master local \
```

```
./examples/src/main/python/pi.py 10
```

通过更改 Spark -submit 的 master 参数，我们还可以将相同的应用程序提交到运行 Spark 的 standalone 集群管理器、Mesos 或 yarn 的集群管理器。

spark-submit 将会方便地运行我们打包的许多示例。我们已经把这本书包装好了。在本章的剩余部分中，我们将通过一些 api 的例子，我们在介绍 Spark 时还没有看到这些 api。

Dataset : 用于结构化数据的类型安全 api。

第二部分 结构化 API--DataFrames, SQL, and Datasets

第四章 结构化 API 概述

本书的这一部分将深入探讨 Spark 的结构化 api。结构化 api 是一种处理各种数据的工具，从非结构化的日志文件到半结构化的 CSV 文件和高度结构化的 Parquet 文件。这些 api 引用了分布式收集 api 的三种核心类型：

- Datasets
- DataFrames
- SQL tables 和 views

虽然它们是本书的独立部分，但大多数结构化 api 都适用于批处理和流计算。这意味着当您使用结构化 api 时，从批处理迁移到流(反之亦然)的过程应该很简单，几乎不需要付出任何努力。我们将在第五部分中详细介绍流计算。

结构化 api 是您用来编写大多数数据流的基本抽象。到目前为止在这本书中,我们采取了概述的方法,说明了 Spark 框架提供了哪些功能。这部分提供了更深入的探索。在本章中，我们将介绍你应该理解的基本概念：

- 类型化和非类型化的 api(及其差异);
- 核心术语是什么;
- Spark 如何实际使用结构化 API 数据流并在集群上执行它。

然后，我们将提供更具体的基于任务的信息，用于处理特定类型的数据或数据源。

注意

在继续之前，让我们回顾一下在第一部分中介绍的基本概念和定义。Spark 是一个分布式编程模型，用户可以在其中指定 transformation。多个 transformation 构建一个有向无

环图 (DAG)。一个 action 开始执行 DAG 的过程，作为一个单一的 job 作业，将它分解成多个 stages 阶段和 task 任务，以便在整个集群中执行。我们使用 transformation 和 action 操作的逻辑结构是 DataFrames 和 Dataset。要创建一个新的 DataFrame 或 Dataset，您需要调用一个转换。要开始计算或转换为本地语言类型，您需要调用一个 action 操作。

4.1. DataFrame 和 Dataset

第一部分讨论 DataFrames。Spark 有两个结构化集合的概念: DataFrames 和 Datasets。稍后我们将讨论(细微的)差别，但是让我们先定义它们都代表什么。

DataFrames 和 Dataset 是(分布式的)类似于表的集合，具有定义好的行和列。每个列必须具有与所有其他列相同的行数(尽管您可以使用 null 来指定值的缺失)，并且每个列都有类型信息，这些信息必须与集合中的每一行一致。对 Spark 来说，DataFrame 和 Dataset 代表着不可变的，延迟计算的计划，这些计划指定应用于驻留在某个位置的数据以生成一些输出的操作。当我们在 DataFrame 上执行 action 操作时，我们指示 Spark 执行实际的 transformation 操作，并返回结果。DataFrame 和 Dataset 表示如何操作行和列来计算用户期望结果的计划。

注意

表和视图基本上与 DataFrames 相同。我们只是针对它们执行 SQL，而不是 DataFrame 代码。我们在第 10 章中讨论了所有这些问题，重点是 Spark SQL。

为了给这些定义添加一些更具体的特性，我们需要讨论 schema，定义分布式集合中存储的数据类型的方式。

4.2. Schemas

schemas 定义了 DataFrame 的列名和类型。您可以手动定义 schemas 模式或从数据源读取 schemas 模式(通常称为读模式)。Schemas 包含列类型,用于声明什么位置存储了什么类型的数据。

4.3. Spark 结构化 API 中的类型概述

Spark 实际上是它自己的编程语言。在内部,Spark 使用一种名为 Catalyst 的引擎,在计划和处理计算任务的过程中,维护自己的类型信息。在这样做的过程中,这打开了各种各样的执行优化,从而产生了显著的差异。Spark 类型直接映射到 Spark 维护的不同语言 api,在 Scala、Java、Python、SQL 和 R 中都存在一个查找表。即使我们从 Python 或 R 中使用 Spark 的结构化 api,我们的大多数操作都将严格地使用 Spark 类型,而不是 Python 类型。例如,以下代码不会在 Scala 或 Python 中执行加法;它实际上只是在 Spark 中执行加法:

```
// in Scala
val df = spark.range(500).toDF("number")
df.select(df.col("number") + 10)

# in Python
df = spark.range(500).toDF("number")
df.select(df["number"] + 10)
```

这个加法操作之所以发生,是因为 Spark 会将用输入语言编写的表达式转换为 Spark 的内部 Catalyst 表示相同类型的信息。然后它会对转换后的内部表示进行操作。

我们马上就会讲到为什么会出现这种情况,但在之前,我们需要讨论下 Datasets。

4.3.1. 对比 DataFrames 与 Datasets

本质上，在结构化的 api 中，还有两个 api，即 “untyped” DataFrames 和 “typed” Datasets。说 DataFrames 是 untyped 的，这是不准确的；它们有类型，但是 Spark 完全维护它们，并且只检查这些类型是否与**运行时模式**中指定的类型一致。然而，另一方面，Datasets 检查类型是否符合**编译时**的规范。Dataset 只适用于 Java 虚拟机(JVM)的语言(Scala 和 Java)，并且我们指定带有 case 类或 Java bean 的类型。

在大多数情况下，您可能会使用 DataFrames。对于 Spark(在 Scala 中)，DataFrames 只是 Row 类型的 Datasets。“Row” 类型是 Spark 对其优化的数据格式的内部表示。这种格式可以进行高效的计算。因为不是使用 JVM 类型（这可能会有 GC 垃圾收集和对象实例化成本），Spark 操作自己的内部数据格式操作，不会产生这些成本。对于 Spark(在 Python 或 R 中)，没有 Dataset 这样的东西：所有的东西都是 DataFrame，因此我们总是能够使用优化的数据格式。

注意

内部的 catalyst 格式在众多的 Spark 演示中都有相关的讲解。鉴于这本书是为更广大的读者准备的，我们将不去深究底层实现。如果你好奇的话，有一些很精彩的演讲可以参考，由 Databricks 公司员工 Josh Rosen 和 Herman van Hovell 提供，都是关于他们在 Spark 的 catalyst 引擎开发中的工作内容。（<https://youtu.be/5ajs8EIPWGI>）

理解 DataFrames、Spark 类型和 schema 需要一些时间来消化。您需要知道的是，当您使用 DataFrames 时，您正在利用 Spark 优化的内部格式。这种格式对 Spark 的所有语言 api 都使用相同的效率增益。如果您需要严格的编译时检查，请阅读第 11 章以了解更多信息。

让我们转到一些更友好和更容易接近的概念：columns 和 rows。

4.3.2. Columns

columns 表示一个简单的类型，如 integer 或 string，复杂类型，如 array 或 map，或 null。Spark 将为您跟踪所有这些类型的信息，并提供多种方式对 columns 进行转换。在第 5 章中广泛讨论了 columns，但是在大多数情况下，您可以将 Spark Column 类型看作是表中的列。

4.3.3. Rows

一行(Row)只是表示数据的一条记录。DataFrame 中的每条数据记录必须是 Row 类型。

我们可以从 SQL、弹性分布式数据集(RDDs)、数据源或手动创建这些 Rows。在这里，我们用一个数值范围来创建一个：

```
// in Scala
spark.range(2).toDF().collect()

# in Python
spark.range(2).collect()
```

这两个结果都是 Row 对象数组。

4.3.4. Spark Types

我们之前提到过 Spark 有大量的内部类型表示。我们在接下来的几页中包含了一个方便的参考表，这样您就可以很容易地引用在您的特定语言中，与 Spark 类型相匹配的类型。

在列出参考表之前，让我们讨论一下如何实例化或声明一个列是某种类型的。

使用相应的 Scala 类型来声明，使用以下代码：

```
import org.apache.spark.sql.types._

val b = ByteType
```

使用相应的 Java 类型，您应该使用以下包中的工厂方法：

```
import org.apache.spark.sql.types.DataTypes;

ByteType x = DataTypes.ByteType;
```

Python 类型有时有特定的需求，您可以看到表 4-1 中列出的。对应于 Scala 和 Java，您可以在表 4-2 和表 4-3 中看到它们。要使用正确的 Python 类型，请使用以下内容：

```
from pyspark.sql.types import *

b = ByteType()
```

下表为每一种 Spark 的语言绑定提供了详细的类型信息。

Table 4-1. Python type reference

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of -128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of -32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of -9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields). Note: fields is a list of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

Table 4-2. Scala type reference

Data type	Value type in Scala	API to access or create a data type
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	java.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is true.
MapType	scala.collection.Map	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is true.

StructType	org.apache.spark.sql.Row	StructType(fields). Note: fields is an Array of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Scala of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]). Note: The default value of nullable is true.

Table 4-3. Java type reference

Data type	Value type in Java	API to access or create a data type
ByteType	byte or Byte	DataTypes.ByteType
ShortType	short or Short	DataTypes.ShortType
IntegerType	int or Integer	DataTypes.IntegerType
LongType	long or Long	DataTypes.LongType
FloatType	float or Float	DataTypes.FloatType
DoubleType	double or Double	DataTypes.DoubleType
DecimalType	java.math.BigDecimal	DataTypes.createDecimalType() DataTypes.createDecimalType(precision, scale).
StringType	String	DataTypes.StringType
BinaryType	byte[]	DataTypes.BinaryType
BooleanType	boolean or Boolean	DataTypes.BooleanType
TimestampType	java.sql.Timestamp	DataTypes.TimestampType
DateType	java.sql.Date	DataTypes.DateType
ArrayType	java.util.List	DataTypes.createArrayType(elementType). Note: The value of containsNull will be true DataTypes.createArrayType(elementType, containsNull).
MapType	java.util.Map	DataTypes.createMapType(keyType, valueType). Note: The value of valueContainsNull will be true. DataTypes.createMapType(keyType, valueType, valueContainsNull)

StructType	org.apache.spark.sql.Row	DataTypes.createStructType(fields). Note: fields is a List or an array of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Java of the data type of this field (for example, int for a StructField with the data type IntegerType)	DataTypes.createStructField(name, dataType, nullable)

值得记住的是，随着 Spark SQL 的持续增长，类型可能会随着时间的推移而改变，因此您可能想要参考 Spark 的文档以备将来更新。当然，所有这些类型都很好，但是您几乎从不

使用纯静态的 DataFrames。你会一直操作和改变 DataFrame。因此，我们要向您介绍结构化 api 中的执行过程。

4.4. 结构化 API 执行过程概述

本节将演示代码是如何跨集群执行的。这将帮助您了解(和调试)编写代码和集群上执行代码的过程，因此，让我们执行一个结构化的 API 查询，了解从用户代码到执行代码的转换过程。以下是这些步骤的概述：

1. 写 DataFrame /Dataset/ SQL 代码。
2. 如果是有效的代码，即代码没有编译错误，Spark 将其转换为一个逻辑计划。
3. Spark 将此逻辑计划转换为物理计划，同时进行代码优化。
4. Spark 然后在集群上执行这个物理计划(基于 RDD 操作)。

要执行代码，我们必须编写代码。然后，将代码提交给 Spark 集群运行。然后，该代码通过 Catalyst 优化器，它决定如何执行代码，并给出一个计划，最后，代码运行，结果返回给用户。

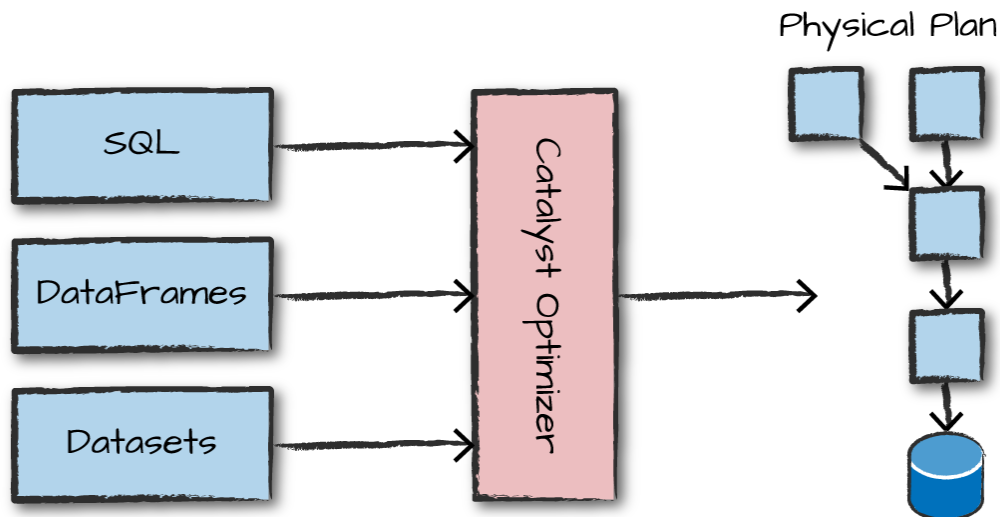


Figure 4-1. The Catalyst Optimizer

4.4.1. 逻辑计划 (Logical Plan)

执行的第一个阶段是将用户代码转换成一个逻辑计划。图 4-2 说明了这个过程。

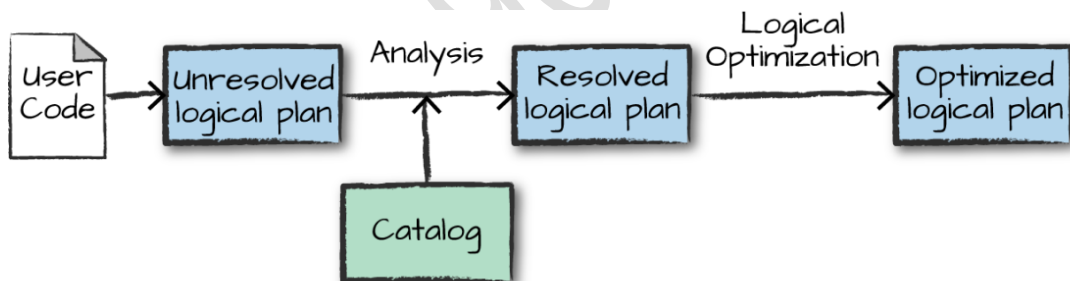


Figure 4-2. The structured API logical planning process

这个逻辑计划只代表一组抽象的转换，不涉及 Executors 执行器或 Driver 驱动程序，它纯粹是将用户的表达式转换成最优的版本。它通过将用户代码转换成 unresolved logical plan. (未解决的逻辑计划) 来实现这一点。这个计划是 unresolved (没有解决)，因为尽管您的代码可能是有效的，但是它引用的表或列可能存在，也可能不存在。Spark 使用 catalog (所有表和 DataFrame 信息的存储库) 来 resolve (解析) analyzer (分析器) 中的列和表。如果 catalog 中不存在所需的表或列名，analyzer (分析器) 可能会拒绝 unresolved logical plan. (未解决的逻辑计划)。如果 analyzer 可以 resolve 这个

unresolved logical plan. (未解决的逻辑计划)，解析的结果会传给 Catalyst 优化器，此优化器是一组规则的集合，用于优化逻辑计划，通过谓词下推、投影等方式进行优化。可以扩展 Catalyst，使其包含特定于领域的优化的规则。

4.4.2. 物理计划 Physical Plan

在成功地创建了一个优化的逻辑计划之后，Spark 就开始了物理计划过程。通常称为 Spark 计划的物理计划指定了逻辑计划如何通过生成不同的物理执行策略，并通过成本模型来比较它们，从而选择一个最优的物理计划在集群上面执行的。如图 4-3 所示。成本比较的一个例子是：通过查看给定表的物理属性(表的大小或其分区有多大)来选择如何执行给定的连接。

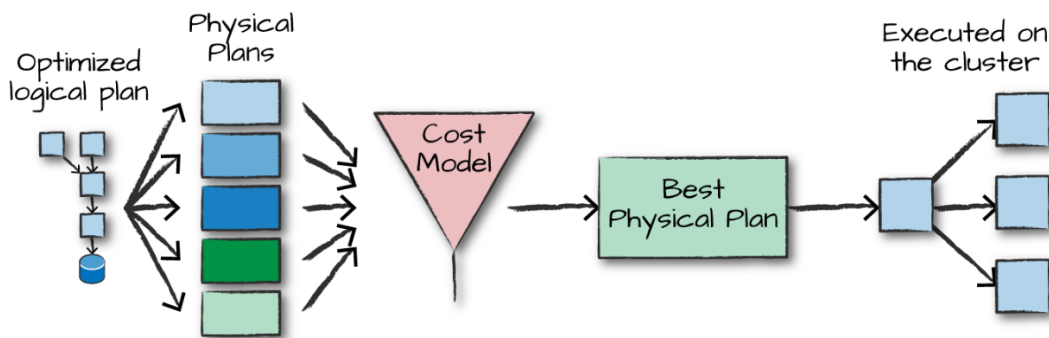


Figure 4-3. The physical planning process

物理规划的结果是一系列的 RDDs 和转换。这个结果说明了为什么有时将 Spark 成为编译器——它接受 DataFrames、dataset 和 SQL 的查询，并将它们编译成 RDD 转换。

4.4.3. 执行

在选择一个物理计划时，Spark 运行所有的 RDDs 代码，即 Spark 的底层编程接口(我们将在第三部分中介绍)。Spark 在运行时执行进一步的优化，生成本地 Java 字节码，可以在执行过程中移除整个任务或阶段。最后将结果返回给用户。

4.5. 结论

在本章中，我们讨论了 Spark 结构化 api，以及 Spark 如何将代码转换为在集群上实际执行的内容。在接下来的章节中，我们讨论了核心概念以及如何使用结构化 api 的关键功能。

第五章 结构化 API 基本操作

在第 4 章中，我们介绍了结构化 API 的核心抽象。本章将从架构概念转向您将使用的工具来操作 DataFrames 和其中的数据。本章专门讨论基本的 DataFrame 操作。聚合操作、窗口函数操作和连接操作将在以后的章节中讨论。

API 查看地址：

<https://spark.apache.org/docs/2.2.0/api/scala/#org.apache.spark.sql.Dataset>

从定义上看，一个 DataFrame 包括一系列的 records（记录。就像 table 中的 rows），这些行的类型是 Row 类型，包括一系列的 columns（就像电子表格中的列。），作用于数据集每条记录之上的计算表达式，实际上是作用于数据记录中的 columns 之上。Schema 定义了每一列的列名和数据类型。DataFrame 的分区定义了 DataFrame 或 Dataset 在整个集群中的物理分布情况。

让我们通过一个例子来看下如何创建一个 DataFrame:

```
// in Scala
val df = spark.read.format("json")
    .load("/data/flight-data/json/2015-summary.json")

# in Python
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
```

我们讨论了 DataFrame 包含列，并且我们使用 schema 来定义它们。让我们看一下当前 DataFrame 上的 schema:

```
df.printSchema()
```

schema 将一切联系在一起，所以它们值得我们深入讨论。

5.1. Schemas

schema 定义了 DataFrame 的列名和数据类型。我们可以让数据源定义模式(称为 schema-on-read)，或者我们可以自己定义它。

警告

在读取数据之前，是否需要定义 schema 取决于您的用例场景。对于某些特殊的分析，schema-on-read 通常是很适合的(尽管有时读取文本格式数据，它可能会慢一些，比如 CSV 或 JSON)。但是，这也会导致精度问题，比如在读取文件时，Long 类型错误地设置为 Integer 类型。当使用 Spark 进行生产提取、转换和加载(ETL)时，手动定义模式通常是个好主意，特别是在使用诸如 CSV 和 JSON 之类的非类型化数据源时，因为 schema infer (模式推理)可能根据所读数据的类型而变化。

让我们从一个简单的文件开始，这是我们在第 4 章中看到的，让半结构化的 Json 文件特性来定义这个 schema。这是美国运输统计局的飞行数据:

```
// in Scala
spark.read.format("json").load("/data/flight-data/json/2015-
summary.json").schema
```

返回结果如下：

```
org.apache.spark.sql.types.StructType = ...
StructType(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true))
```

一个 schema 就是一个 StructType，由多个 StructField 类型的 fields 组成，每个 field 包括一个列名称、一个列类型、一个布尔型的标识（是否可以有缺失值和 null 值）。

schema 可以包含其他 StructType(Spark 的复杂类型)。在第 6 章中,我们将讨论复杂类型。如果数据中的类型(在运行时)与模式不匹配,Spark 将抛出一个错误。下面的示例展示了如何在 DataFrame 上创建和执行特定的 schema。

```
// in Scala

import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
import org.apache.spark.sql.types.Metadata

val myManualSchema = StructType(Array(
  StructField("DEST_COUNTRY_NAME", StringType, true),
  StructField("ORIGIN_COUNTRY_NAME", StringType, true),
  StructField("count", LongType, false,
    Metadata.fromJson("{\"hello\":\"world\"}"))
))

val df = spark.read.format("json").schema(myManualSchema)
  .load("/data/flight-data/json/2015-summary.json")
```

正如第 4 章所讨论的,我们不能简单地通过每种语言类型设置类型,因为 Spark 维护了它自己的类型信息。现在让我们讨论一下 scheme 定义的内容: columns。

5.2. 列和表达式(Columns 和 Expressions)

Spark 中的列类似于电子表格中的列。您可以从 DataFrame 中选择列、操作列和删除列,这些操作称为 Expressions 表达式。

对 Spark 来说,列是逻辑结构,它仅仅表示通过一个表达式按每条记录计算出的一个值。

这意味着,要得到一个 column 列的真实值,我们需要有一行 row 数据,为了得到一行数

据，我们需要有一个 `DataFrame`。您不能在 `DataFrame` 的上下文之外操作单个列。您必须在 `DataFrame` 内使用 Spark 转换来修改列的内容。

5.2.1. Columns

有许多不同的方法来构造和引用列，但最简单的两种方法是使用 `col()` 或 `column()` 函数。要使用这些函数中的任何一个，您需要传入一个列名：

```
// in Scala
import org.apache.spark.sql.functions.{col, column}

col("someColumnName")

column("someColumnName")
```

在这本书中我们将使用 `col()` 函数。如前所述，这个列可能存在于我们的 `DataFrames` 中，也可能不存在。在将列名称与我们在 `Catalog` 中维护的列进行比较之前，列不会被解析，即列是 `unresolved`。列和表解析发生在 *analyzer* 分析器阶段，如第 4 章所述。

注意

我们刚才提到的两种不同的方法引用列。Scala 有一些独特的语言特性，允许使用更多的简写方式来引用列。以下的语法糖执行完全相同的事情，即创建一个列，但不提供性能改进：

```
$"myColumn"
'myColumn
```

`$`允许我们将一个字符串指定为一个特殊的字符串，该字符串应该引用一个表达式。标记(`'`)是一种特殊的東西，称为符号；这是一个特定于 scala 语言的，指向某个标识符。它们都执行相同的操作，是按名称引用列的简写方式。当您阅读不同的人的 Spark 代码时，您可能会看到前面提到的所有引用。我们把选择权留给您，您可以使用任何对您和您工作的人来说最舒适和最容易维护的东西。

显式列引用

如果需要引用特定的 DataFrame 的列，可以在特定的 DataFrame 上使用 `col` 方法。当您执行连接时，这可能非常有用，并且需要引用一个 DataFrame 中的特定列，该列可能与连接的 DataFrame 中的另一个列共享一个名称。

5.2.2. 表达式 Expressions

我们前面提到过，列是表达式，但表达式是什么？表达式是在 DataFrame 中数据记录的一个或多个值上的一组转换。把它想象成一个函数，它将一个或多个列名作为输入，解析它们，然后潜在地应用更多的表达式，为数据集中的每个记录创建一个单一值。重要的是，这个“单一值”实际上可以是一个复杂的类型，比如映射或数组。我们将在第六章中看到更多的复杂类型。

在最简单的情况下，通过 `expr` 函数创建的表达式只是一个 DataFrame 列引用。在最简单的情况下，`expr(“someCol”)` 等价于 `col(“someCol”)`。

列可以作为表达式

列提供了表达式功能的一个子集。如果您使用 `col()` 并希望在该列上执行转换，则必须在该列引用上执行转换。在使用表达式时，`expr` 函数实际上可以从字符串解析转换和列引用，并可以将其传递到进一步的转换中。让我们看一些例子。

`expr(“someCol - 5”)` 与执行 `col(“someCol”) - 5`，或甚至 `expr(“someCol”) - 5` 的转换相同。这是因为 Spark 将它们编译为一个逻辑树，逻辑树指定了操作的顺序。这一开始可能有点让人迷惑，但记住几个关键点：

- 列就是表达式
- 这些列上的列表表达式和转换编译成与解析表达式相同的逻辑计划。

让我们以一个例子来说明:

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```

图 5-1 显示了该逻辑树的概述。

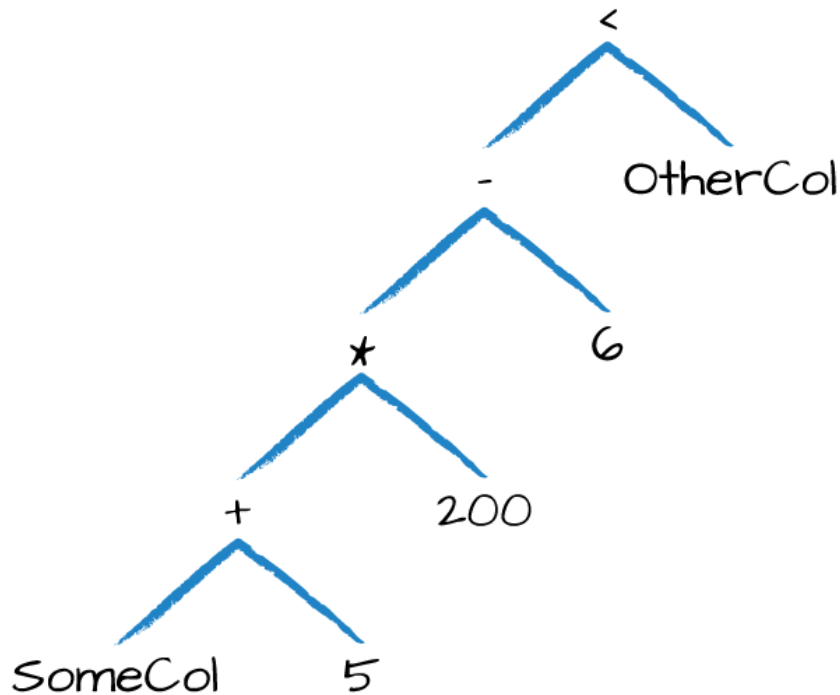


Figure 5-1. A logical tree

这看起来很熟悉，因为它是一个有向无环图。此图等价于以下代码:

```
// in Scala
import org.apache.spark.sql.functions.expr
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

这是一个非常重要的观点。请注意前面的表达式实际上是有效的 SQL 代码，就像您可能放入 SELECT 语句一样？这是因为这个 SQL 表达式和之前的 DataFrame 代码在执行之前编译到相同的底层逻辑树。这意味着您可以将表达式编写为 DataFrame 代码或 SQL 表达式，并获得完全相同的性能特征。

访问 DataFrame 中的 columns

有时，您需要查看 DataFrame 的列，您可以使用类似 `printSchema` 的方法来实现；但是，如果您想通过编程访问列，可以使用 `columns` 属性查看 DataFrame 上的所有列：

```
spark.read.format("json").load("/data/flight-data/json/2015-summary.json")  
  .columns
```

5.3. Records 和 Rows

在 Spark 中，DataFrame 中的每一行都是单个记录。Spark 表示此记录为 Row 类型的对象。即一个 record 是一个 Row 类型的对象。Spark 使用列表达式 expression 操作 Row 对象，以产生有效的结果值。Row 对象的内部表示为：字节数组。因为我们使用列表达式操作 Row 对象，所以，字节数据不会对最终用户展示，即对用户不可见。

5.3.1. 创建 Rows

您可以手动实例化一个 Row 对象，并实例化每一列的值。需要注意的是：只有 DataFrame 有 schema，Row 本身没有 schema。这意味着，如果您手动创建一个 Row 对象，则必须以与它们可能被附加的 DataFrame 的 schema 相同的顺序指定列值(我们将在讨论创建 DataFrames 时看到这个)：

```
// in Scala  
  
import org.apache.spark.sql.Row  
  
val myRow = Row("Hello", null, 1, false)
```

访问创建的 Row 对象中的数据非常容易：你只需指定你想要列值的位置。

```
// in Scala  
  
myRow(0) // type Any  
  
myRow(0).asInstanceOf[String] // String  
  
myRow.getString(0) // String  
  
myRow.getInt(2) // Int
```

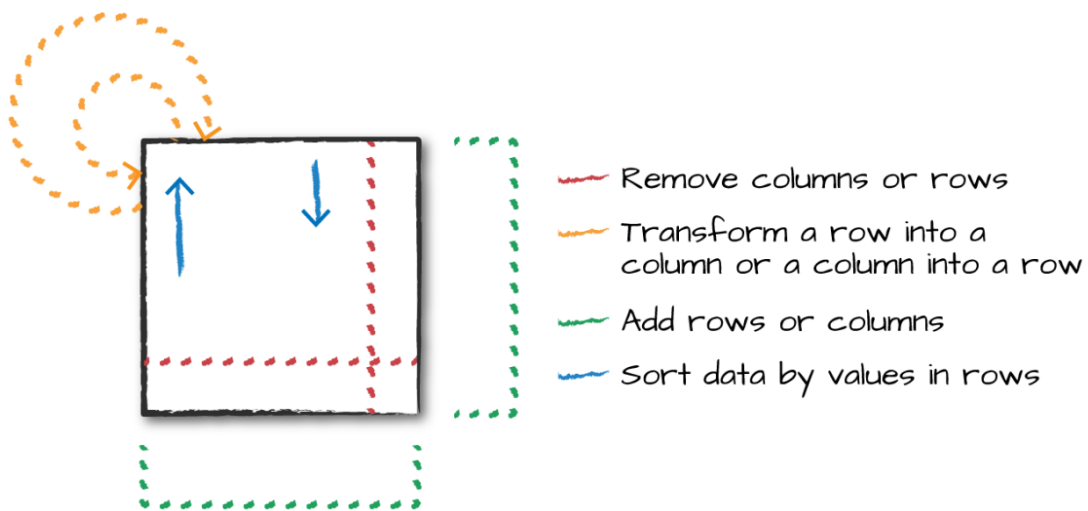
您还可以使用 Dataset api 在相应的 Java 虚拟机(JVM)对象中显式地返回一组数据 (见第十一章)。

5.4. DataFrame Transformations

```
DataFrame  
  
https://spark.apache.org/docs/2.2.0/api/scala/#org.apache.spark.sql.functions  
$
```

现在我们简要地定义了 DataFrame 的核心部分，我们将开始操作 DataFrames。在使用单一的 DataFrames 时，有一些基本的操作。这些分解为几个核心操作，如图 5-2 所示：

- 我们可以添加行或列
- 我们可以删除行或列
- 我们可以变换一行成一系列(反之亦然)
- 我们可以根据列值对 rows 进行排序



幸运的是，我们可以将所有这些转换成简单的 transformation 方法，最常见的是那些使用一个列，逐行更改它，然后返回结果。

5.4.1. 创建 DataFrame

如前所述，我们可以从原始数据源创建 DataFrames。这在第 9 章中得到了广泛的讨论；但是，现在我们将使用它们来创建一个 DataFrame 示例。（在本章后面的演示目的中，我们还将把它注册为一个临时视图，以便我们可以用 SQL 查询它，并显示 SQL 中的基本转换。）

```
// in Scala
val df = spark.read.format("json")
    .load("/data/flight-data/json/2015-summary.json")
df.createOrReplaceTempView("dfTable")

# in Python
df = spark.read.format("json").load("/data/flight-data/json/2015-
summary.json")
df.createOrReplaceTempView("dfTable")
```

我们还可以创建动态 DataFrames，通过转换一组数据为 DataFrame。

```
// in Scala
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, StructType, StringType,
LongType}

val myManualSchema = new StructType(Array(
  new StructField("some", StringType, true),
  new StructField("col", StringType, true),
  new StructField("names", LongType, false)))
val myRows = Seq(Row("Hello", null, 1L))
val myRDD = spark.sparkContext.parallelize(myRows)
val myDf = spark.createDataFrame(myRDD, myManualSchema)
myDf.show()
```

注意

在 Scala 中，我们还可以利用控制台中的 Spark 的 implicits(如果您将它们导入到 JAR 代码中)，则可以在 Seq 类型上运行 toDF。这不能很好地使用 null 类型，因此并不推荐用于生产用例。

```
// in Scala
val myDF = Seq(("Hello", 2, 1L)).toDF("col1", "col2", "col3")
```

现在，您已经知道如何创建 DataFrames 了，让我们来看看它们最有用的方法

`select` 方法：接收列 `column` 或表达式 `expression` 为参数。

`selectExpr` 方法：接收字符串表达式 `expression` 为参数。

还有一些方法，通过函数的形式提供，在 `org.apache.spark.sql.functions` 包中。

使用这三个工具，您应该能够解决您在 `DataFrames` 中可能遇到的大多数数据分析挑战。

5.4.2. `select` 和 `selectExpr`

`select` 和 `selectExpr` 允许您在数据表上执行与 SQL 查询等效的 `DataFrame` 操作：

```
-- SQL 查询方式

SELECT * FROM dataframeTable

SELECT columnName FROM dataframeTable

SELECT columnName * 10, otherColumn, someOtherCol as c FROM
dataframeTable
```

让我们浏览一些 `DataFrames` 的示例，讨论解决这个问题的一些不同方法。最简单的方法就是使用 `DataFrame` 的 `select` 方法，并将列名作为字符串参数：

```
// Scala 方式
df.select("DEST_COUNTRY_NAME").show(2)

-- SQL 方式

SELECT DEST_COUNTRY_NAME FROM dfTable LIMIT 2
```

您可以使用相同的查询样式选择多个列，只需在 `select` 方法调用中添加更多的列名字符串参数：

```
// in Scala
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM dfTable

LIMIT 2
```

正如在“列和表达式”章节中所讨论的，您可以用许多不同的方式引用列；您需要记住的是，您可以交替使用它们：

```
// in Scala
import org.apache.spark.sql.functions.{expr, col, column}

df.select(
  df.col("DEST_COUNTRY_NAME"),
  col("DEST_COUNTRY_NAME"),
  column("DEST_COUNTRY_NAME"),
  'DEST_COUNTRY_NAME,
  $"DEST_COUNTRY_NAME",
  expr("DEST_COUNTRY_NAME"))
.show(2)
```

一个常见的错误是混合使用列对象和列字符串。例如，下列代码将导致编译错误：

```
df.select(col("DEST_COUNTRY_NAME"), "EST_COUNTRY_NAME")
```

正如我们到目前为止所看到的，`expr` 是我们可以使用的最灵活的引用。它可以引用一个简单的列或一个列字符串操作。为了说明这一点，让我们更改列名，然后通过使用 `AS` 关键字来更改它。然后使用列上的 `alias` 方法：

```
// in Scala
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

# in Python
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME as destination FROM dfTable LIMIT 2
```

这将列名更改为 “destination”。您可以进一步操作您的表达式作为另一个表达式的结果：

```
// in Scala
df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))
    .show(2)
```

前面的操作将列名更改为原来的名称。

因为 `select` 后跟一系列 `expr` 是一种常见的模式，Spark 有一个高效实现这一点的简写：`selectExpr`。这可能是日常使用中最方便的方式：

```
// in Scala
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)
```

这开启了 Spark 的真正力量。我们可以将 `selectExpr` 视为一种构建复杂表达式的简单方法，这些表达式可以创建新的 DataFrames。事实上，我们可以添加任何有效的非聚合 SQL 语句，只要能否进行列解析，它就有效。这里有一个简单的例子，它在我们的 DataFrame 中添加了一个新的列 `withinCountry`，该列指定目的地和起点是否相同：

```
// in Scala
```

```
df.selectExpr(
```

```
  "*", // 包含所有的原始列
```

```
  "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry") .show(2)
```

```
-- in SQL
```

```
SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry FROM dfTable LIMIT 2
```

输出为：

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count	withinCountry
United States	Romania	15	false
United States	Croatia	1	false

使用 select expression，我们还可以利用我们拥有的函数来指定整个 DataFrame 上的聚合：

合：

```
// in Scala
```

```
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)
```

```
# in Python
```

```
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)
```

```
-- in SQL
```

```
SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FROM dfTable
```

```
LIMIT 2
```

输出为：

avg(count)	count(DISTINCT DEST_COUNTRY_NAME)
1770.765625	132

5.4.3. 字面常量转换为 Spark 类型(Literals)

有时，我们需要将显式字面常量值传递给 Spark，它只是一个值(而不是一个新列)。这可能是一个常数值或者我们以后需要比较的值。我们的方法是通过 Literals，将给定编程语言的字面值转换为 Spark 能够理解的值。Literals 是表达式，你可以用同样的方式使用它们：

```
// in Scala

import org.apache.spark.sql.functions.lit

df.select(expr("*"), lit(1).as("One")).show(2)

# in Python

from pyspark.sql.functions import lit

df.select(expr("*"), lit(1).alias("One")).show(2)
```

在 SQL 中，字面值只是特定的值：

```
-- in SQL

SELECT *, 1 as One FROM dfTable LIMIT 2
```

输出为：

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count	One
United States	Romania	15	1
United States	Croatia	1	1

当您可能需要检查一个值是否大于某个常量或其他通过编程创建的变量时，就会出现这种情况。

5.4.4. 添加列

还有一种更正式的方式，将新列添加到 DataFrame 中，这是通过在 DataFrame 上使用 withColumn 方法来实现的。例如，让我们添加一个列，将数字 1 添加为一个列：

```
// in Scala

df.withColumn("numberOne", lit(1)).show(2)

# in Python

df.withColumn("numberOne", lit(1)).show(2)

-- in SQL

SELECT *, 1 as numberOne FROM dfTable LIMIT 2
```

输出为：

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count	numberOne
United States	Romania	15	1
United States	Croatia	1	1

让我们做一些更有趣的事情，把它变成一个实际的表达式。在下一个示例中，我们将设置一个布尔标志，当源国与目标国相同时：

```
// in Scala

df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME ==
DEST_COUNTRY_NAME"))

.show(2)

# in Python
```



```
df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME ==  
DEST_COUNTRY_NAME"))\  
  .show(2)
```

注意，withColumn 函数有两个参数：列名和为 DataFrame 中的给定行创建值的表达式。

有趣的是，我们也可以这样重命名列。SQL 语法与前面的相同，因此我们可以在这个示例中省略它：

```
df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns
```

结果为：

```
... DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count, Destination
```

5.4.5. 重命名列

虽然我们可以按照刚才描述的方式重命名列，但是另一种方法是使用 withColumnRename 方法。这会将第一个参数中的字符串的名称重命名为第二个参数中的字符串：

```
// in Scala  
  
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns  
  
... dest, ORIGIN_COUNTRY_NAME, count
```

5.4.6. 保留字和关键词

您可能遇到的一件事是保留字符，比如空格或列名称中的破折号。处理这些意味着适当地转义列名。在 Spark 中，我们通过使用单撇号(')字符来实现这一点。让我们使用 `withColumn`，创建具有保留字符的列。我们将展示两个示例——在这里显示的一个示例中，我们不需要转义字符，但是在下一个示例中，我们需要：

```
// in Scala
```

```
import org.apache.spark.sql.functions.expr
```

```
val dfWithLongColName = df.withColumn("This Long Column-Name",  
    expr("ORIGIN_COUNTRY_NAME"))
```

这里不需要转义字符，因为 `withColumn` 的第一个参数只是新列名的字符串。然而，在本例中，我们需要使用单撇号，因为我们在表达式中引用了一个列：

```
// in Scala
```

```
dfWithLongColName.selectExpr(  
    "This Long Column-Name",  
    "This Long Column-Name as `new col`")  
.show(2)
```

```
dfWithLongColName.createOrReplaceTempView("dfTableLong")
```

```
-- in SQL
```

```
SELECT `This Long Column-Name`, `This Long Column-Name` as `new
col`

FROM dfTableLong LIMIT 2
```

我们可以引用具有保留字符的列(而不是转义它们), 如果我们正在做显式的字符串到列的引用, 它被解释为文字而不是表达式。我们只需要转义使用保留字符或关键字的表达式。

下面两个例子从同一个 DataFrame 中都得到了结果:

```
// in Scala
dfWithLongColName.select(col("This Long Column-Name")).columns

# in Python
dfWithLongColName.select(expr("`This Long Column-Name`")).columns
```

5.4.7. 区分大小写

默认情况下, Spark 是不区分大小写的;但是, 您可以通过设置配置使 Spark case 变得大小写敏感:

```
-- in SQL

set spark.sql.caseSensitive true
```

5.4.8. 删除列

既然我们已经创建了这个列, 那么让我们看看如何从 DataFrames 中删除列。您可能已经注意到我们可以通过使用 select 来实现这一点。然而, 还有一个专门的方法叫做 drop:

```
df.drop("ORIGIN_COUNTRY_NAME").columns
```

我们可以通过将多个列作为参数传递，来删除多个列:

```
dfWithLongColName.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

5.4.9. 更改列的类型(cast)

有时，我们可能需要列从一种类型转换到另一种类型;例如，如果我们有一组 StringType 应该是整数。我们可以将列从一种类型转换为另一种类型，例如，让我们将计数列从整数转换为类型 Long:

```
df.withColumn("count2", col("count").cast("long"))  
-- in SQL  
SELECT *, cast(count as long) AS count2 FROM dfTable
```

5.4.10. 过滤行

为了过滤行，我们创建一个计算值为 true 或 false 的表达式。然后用一个等于 false 的表达式过滤掉这些行。使用 DataFrames 执行此操作的最常见方法是将表达式创建为字符串，或者使用一组列操作构建表达式。执行此操作有两种方法:您可以使用 where 或 filter，它们都将执行相同的操作，并在使用 DataFrames 时接受相同的参数类型。因为熟悉 SQL，我们将坚持使用 where;但是，filter 方法是有效的。

注意

在使用 Scala 或 Java 的 Dataset API 时，filter 还接受 Spark 将应用于数据集中每个记录的任意函数。有关更多信息，请参见第 11 章。

以下过滤器是等效的，在 Scala 和 Python 中，结果是相同的:

```
df.filter(col("count") < 2).show(2)

df.where("count < 2").show(2)

-- in SQL

SELECT * FROM dfTable WHERE count < 2 LIMIT 2
```

输出为：

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Croatia	1
United States	Singapore	1

您可能希望将多个过滤器放入相同的表达式中。尽管这是可能的，但它并不总是有用的，因为 Spark 会自动执行所有的过滤操作，而不考虑过滤器的排序。这意味着，如果您想指定多个过滤器，只需将它们按顺序链接起来，让 Spark 处理其余部分：

```
// in Scala

df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") !=
"Croatia")

.show(2)

-- in SQL

SELECT * FROM dfTable WHERE count < 2 AND

ORIGIN_COUNTRY_NAME != "Croatia"

LIMIT 2
```

输出为：

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Singapore	1
Moldova	United States	1

5.4.11. 获取唯一行

一个非常常见的用例是在一个 DataFrame 中提取唯一的或不同的值。这些值可以在一个或多个列中。我们这样做的方法是在 DataFrame 上使用不同的方法，它允许我们对 DataFrame 中的任何行进行删除重复行。例如，让我们在数据集中获取唯一的起源地。当然，这是一个转换，它将返回一个新的 DataFrame，只有唯一的行：

```
// in Scala
```

```
df.select("ORIGIN_COUNTRY_NAME",
"DEST_COUNTRY_NAME").distinct().count()
```

```
-- in SQL
```

```
SELECT COUNT(DISTINCT(ORIGIN_COUNTRY_NAME,
DEST_COUNTRY_NAME)) FROM dfTable
```

结果为 256.

```
// in Scala
```

```
df.select("ORIGIN_COUNTRY_NAME").distinct().count()
```

```
# in Python
```

```
df.select("ORIGIN_COUNTRY_NAME").distinct().count()
```

```
-- in SQL
```

```
SELECT COUNT(DISTINCT ORIGIN_COUNTRY_NAME) FROM dfTable
```

结果为 125。

5.4.12. 随机样本

有时，您可能只想从 DataFrame 中抽取一些随机记录。

您可以使用 DataFrame 上的 `sample` 方法来实现这一点，这使您可以指定要从 DataFrame 中提取指定比例的数据行，以及您是否希望使用或不使用替换：

```
val seed = 5  
  
val withReplacement = false  
  
val fraction = 0.5  
  
df.sample(withReplacement, fraction, seed).count()
```

5.4.13. 随机分割

当您需要将您的 DataFrame 分割为原始 DataFrame 的随机“分割”时，随机分割将非常有用。它经常与机器学习算法一起用于创建训练、验证和测试集。在下一个示例中，我们将把 DataFrame 分成两种不同的 DataFrame，通过设置权重来划分 DataFrame(这些是函数的参数)，因为这个方法被设计成是随机的，所以我们还将指定一个种子(只需在代码块中以您所选择的数量替换种子)。需要注意的是，如果您没有为每个加起来为 1 的 DataFrame 指定一个比例，那么它们将被规范化，以便：

```
// in Scala  
  
val dataFrames = df.randomSplit(Array(0.25, 0.75), seed)
```

```
dataFrames(0).count() > dataFrames(1).count() // False
```

5.4.14. 连接和附加行(union)

正如您在上一节中了解到的，DataFrames 是不可变的。这意味着用户不能向 DataFrames 追加，因为这会改变它。要附加到 DataFrame，必须将原始的 DataFrame 与新的 DataFrame 结合起来。这只是连接了两个 DataFrames。对于 union two DataFrames，您必须确保它们具有相同的模式和列数；否则，union 将会失败。

注意

union 目前是基于位置而不是模式执行的。这意味着列不会自动按照您认为的方式排列。

```
// in Scala
import org.apache.spark.sql.Row
val schema = df.schema
val newRows = Seq(
  Row("New Country", "Other Country", 5L),
  Row("New Country 2", "Other Country 3", 1L)
)
val parallelizedRows = spark.sparkContext.parallelize(newRows)
val newDF = spark.createDataFrame(parallelizedRows, schema)
df.union(newDF)
  .where("count = 1")
  .where($"ORIGIN_COUNTRY_NAME" != "United States")
  .show() // get all of them and we'll see our new rows at the end
```


在 Scala 中，必须使用 `!=` 运算符，这样您不仅可以将来求值的列表表达式与字符串进行比较，还可以将其与已求值的列表表达式进行比较。

输出为：

```
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
|    United States|          Croatia|    1|
...
|    United States|          Namibia|    1|
|    New Country 2|    Other Country 3|    1|
+-----+-----+-----+
```

如预期的那样，您将需要使用这个新的 DataFrame 引用，以便引用带有新添加行的 DataFrame。一种常见的方法是将 DataFrame 设置为视图，或者将其注册为表，以便在代码中更动态地引用它。

5.4.15. 行排序

在对 DataFrame 中的值进行排序时，我们总是希望对 DataFrame 顶部的最大或最小值进行排序。有两个相同的操作可以执行这种操作：`sort` 和 `orderBy`。它们接受列表表达式和字符串以及多个列。默认是按升序排序：

// in Scala

```
df.sort("count").show(5)
```

```
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
```

```
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(5)
```

要更明确地指定排序方向，需要在操作列时使用 `asc` 和 `desc` 函数。这些允许您指定给定列的排序顺序：

```
// in Scala

import org.apache.spark.sql.functions.{desc, asc}

df.orderBy(expr("count desc")).show(2)

df.orderBy(desc("count"), asc("DEST_COUNTRY_NAME")).show(2)
```

一个高级技巧是使用 `asc_nulls_first`、`desc_nulls_first`、`asc_nulls_last` 或 `desc_nulls_last`，来指定您希望在有序的 `DataFrame` 中显示 `null` 值的位置。

出于优化目的，有时建议在另一组转换之前对每个分区进行排序。您可以使用 `sortWithinPartitions` 方法来执行以下操作：

```
// in Scala

spark.read.format("json").load("/data/flight-data/json/*-summary.json")

  .sortWithinPartitions("count")
```

我们将在第 3 部分中讨论调优和优化。

5.4.16. `limit`

通常，您可能想要限制从 `DataFrame` 中提取的内容；例如，您可能只想要一些 `DataFrame` 的前十位。您可以使用极限法：

```
// in Scala

df.limit(5).show()

# in Python

df.limit(5).show()

-- in SQL

SELECT * FROM dfTable LIMIT 6
```

```
// in Scala
df.orderBy(expr("count desc")).limit(6).show()

# in Python
df.orderBy(expr("count desc")).limit(6).show()

-- in SQL
SELECT * FROM dfTable ORDER BY count desc LIMIT 6
```

5.4.17. Repartition 和 Coalesce(重新分区和合并)

另一个重要的优化方式是根据一些经常过滤的列对数据进行分区，它控制跨集群的数据的物理布局，包括分区计划和分区数量。

Repartition 将导致数据的完全 shuffle，无论是否需要重新 shuffle。这意味着只有当将来的分区数目大于当前的分区数目时，或者当您希望通过一组列进行分区时，您才应该使用 **Repartition**:

```
// in Scala

df.rdd.getNumPartitions // 1

◀

// in Scala

df.repartition(5)
```

如果经常对某个列进行过滤，那么基于该列进行重新分区是值得的:

```
// in Scala  
  
df.repartition(col("DEST_COUNTRY_NAME"))
```

您也可以选择指定您想要的分区数量:

```
// in Scala  
  
df.repartition(5, col("DEST_COUNTRY_NAME"))
```

另一方面, **Coalesce** 不会导致完全 shuffle, 并尝试合并分区。

这个操作将根据目标国家的名称将您的数据转移到 5 个分区中, 然后合并它们(没有完全 shuffle) :

```
// in Scala  
  
df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)
```

5.4.18. 收集 row 到 driver 程序

如前所述, Spark 在驱动程序中维护集群的状态。有时, 您需要收集一些数据给驱动程序, 以便在本地机器上操作它。

到目前为止, 我们还没有明确地定义这个操作。然而, 我们使用了几种不同的方法来实现这一点, 它们实际上都是相同的。

- **collect** 从整个 DataFrame 中获取所有数据
- **take** 选取 DataFrame 的前几行

- **show** 打印出几行数据

```
// in Scala  
  
val collectDF = df.limit(10)  
  
collectDF.take(5) // take works with an Integer count  
  
collectDF.show() // this prints it out nicely  
  
collectDF.show(5, false)  
  
collectDF.collect()
```

还有一种方法可以收集行到驱动程序，以便迭代整个数据集。方法 `toLocalIterator` 将把分区收集到驱动程序作为迭代器。此方法允许以分区为单位方式遍历整个数据集：

```
collectDF.toLocalIterator()
```

注意

任何数据收集到 driver 端，都可能是一个非常昂贵的操作! 如果您有一个大的数据集，调用 `collect`，可能使驱动程序崩溃。如果您使用 `toLocalIterator`，并且有非常大的分区，也可能很容易地崩溃 driver 节点并丢失应用程序的状态。

5.5. 总结

本章介绍了 DataFrame 的基本操作。学习了使用 Spark DataFrames 所需的简单概念和工具。第 6 章更详细地介绍 DataFrame 中操作数据的所有不同方法。

第六章 处理不同类型的数据

6.1. 在哪里查看 API

在我们开始之前，有必要解释一下您作为用户应该在哪里查找 DataFrame API。Spark 是一个不断增长的项目，任何书籍(包括这一本书)都是某个时间的快照。在本书中，我们的首要任务之一是教授在撰写本文时，您应该在哪里寻找转换数据的函数。以下是关键的地方：

DataFrame(Dataset)的方法

这实际上有点小技巧，因为 DataFrame 只是 Row 类型的 Dataset，你最终会看到 Dataset 方法，在这个链接中可以找到：

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset>

Dataset 子模块如 DataFrameStatFunctions(包含各种统计相关的功能) 和 DataFrameNaFunctions(处理空数据时相关的函数)有更多的方法来解决特定的问题集。

Column 方法

这些在第 5 章的大部分内容中已经介绍过。它们包含各种与列相关的通用方法，如 alias 或 contains。您可以在这里找到列方法的 API 引用。

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column>

org.apache.spark.sql.functions 包含一系列不同数据类型的函数。通常，您会看到整个包被导入，因为它们被频繁地使用。您可以在这里找到 SQL 和 DataFrame 函数。

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions>

如此繁多的函数，可能会让人觉得有点难以承受，但不用担心，这些函数中的大多数都是 SQL 和分析系统中可以找到的。所有这些工具的存在都是为了实现一个目的，即将数据行以一种格式或结构转换为另一种格式。这可能会创建更多的行或减少可用的行数。首先，让我们创建用于分析的 DataFrame:

```
// in Scala
val df = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/2010-12-01.csv")
df.printSchema()
df.createOrReplaceTempView("dfTable")
```

下面是 schema 的结果和一个数据样本:

```

root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)

```

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	Unit...
536365	85123A	WHITE HANGING HEA...	6	2010-12-01 08:26:00	...
536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	...
...					
536367	21755	LOVE BUILDING BLO...	3	2010-12-01 08:34:00	...
536367	21777	RECIPE BOX WITH M...	4	2010-12-01 08:34:00	...

6.2. 转换为 Spark 类型

在本章中，您将看到我们所做的一件事是将本地原生类型转换为 Spark 类型。我们用我们在这里介绍的第一个函数来做这个，`lit()` 函数。此函数将另一种语言中的类型转换为其相应的 Spark 类型表示。以下是我们如何将几个不同类型的 Scala 和 Python 值转换为各自的 Spark 类型：

// in Scala

```
import org.apache.spark.sql.functions.lit
```

```
df.select(lit(5), lit("five"), lit(5.0))
```

in Python

```
from pyspark.sql.functions import lit
```

```
df.select(lit(5), lit("five"), lit(5.0))
```

SQL 中没有等价的函数，所以我们可以直接使用这些值：


```
-- in SQL
```

```
SELECT 5, "five", 5.0
```

6.3. 使用 Boolean 类型

当涉及到数据分析时，布尔值非常重要，因为它们是所有过滤的基础。布尔语句由四个元素组成:and, or, true 和 false。我们使用这些简单的结构来构建逻辑语句，以判断真假。

当一行数据必须通过测试(evaluate to true)或被过滤时，这些语句通常被用作条件要求。

让我们使用我们的零售数据集探索使用布尔值。我们可以指定等于以及小于或大于:

```
// in Scala
import org.apache.spark.sql.functions.col
df.where(col("InvoiceNo").equalTo(536365))
  .select("InvoiceNo", "Description")
  .show(5, false)
```

```
// in Scala
import org.apache.spark.sql.functions.col
df.where(col("InvoiceNo") === 536365)
  .select("InvoiceNo", "Description")
  .show(5, false)
```

注意

Scala 有一些关于使用`==`和`===`的特殊语义。在 Spark 中，如果您想要通过等式进行过滤，您应该使用`===`(相等)或`!==`(不相等)。您还可以使用 `not` 函数和 `equalTo` 方法。

另一个选项(可能是最干净的选项)是将谓词指定为字符串中的表达式。这对 Python 或 Scala 有效。请注意，这也使您可以使用另一种表达“不等于”的方式：

```
df.where("InvoiceNo = 536365")  
  .show(5, false)  
  
df.where("InvoiceNo <> 536365")  
  .show(5, false)
```

我们提到，在使用 `and` 或 `or` 时，可以使用多个部分指定布尔表达式。在 Spark 中，您应该始终将 `and` 过滤器连接在一起作为一个序列过滤器。原因是，即使布尔语句是串行的（一个接一个），Spark 将所有这些过滤器压平为一个语句，并同时执行过滤器。虽然您可以通过使用 `and` (如果您愿意的话)显式地指定语句，但是如果您以串行方式指定语句，它们通常更容易理解和读取。`or` 须在同一语句中指明：

```
// in Scala  
val priceFilter = col("UnitPrice") > 600  
val descripFilter = col("Description").contains("POSTAGE")  
df.where(col("StockCode").isin("DOT")).where(priceFilter.or(descripFilter))  
  .show()
```

```
-- in SQL

SELECT * FROM dfTable WHERE StockCode in ("DOT") AND(UnitPrice > 600 OR
instr(Description, "POSTAGE") >= 1)
```

布尔表达式不仅仅是为过滤器保留的。要过滤一个 DataFrame，您还可以指定一个布尔列：

```
// in Scala

val DOTCodeFilter = col("StockCode") === "DOT"
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
df.withColumn("isExpensive", DOTCodeFilter.and(priceFilter.or(descripFilter)))
  .as("isExpensive")
  .select("unitPrice", "isExpensive").show(5)

-- in SQL

SELECT UnitPrice, (StockCode = 'DOT' AND
  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)) as isExpensive
FROM dfTable
WHERE (StockCode = 'DOT' AND
  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1))
```

如果您来自 SQL 背景，那么所有这些语句都应该非常熟悉。实际上，它们都可以表示为 where 子句。事实上，使用 SQL 语句来表达过滤器比使用编程的 DataFrame 接口和 Spark SQL 更容易，这使得我们无需付出任何性能代价就可以做到这一点。例如，以下两个表述是等价的：

```
// in Scala

import org.apache.spark.sql.functions.{expr, not, col}

df.withColumn("isExpensive", not(col("UnitPrice").leq(250)))

  .filter("isExpensive")

  .select("Description", "UnitPrice").show(5)

df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))

  .filter("isExpensive")

  .select("Description", "UnitPrice").show(5)
```

注意

一个可能出现的“陷阱”，如果在创建布尔表达式时使用 null 数据。如果您的数据中有一个空值，那么您将需要以稍微不同的方式处理事情。null 值等于测试：

```
df.where(col("Description").eqNullSafe("hello")).show()
```

6.4. 使用 Numbers 类型

为了构建一个虚构的示例，假设我们发现我们在零售数据集中错误地记录了数量，真实数量等于(当前数量*单价)的平方 + 5。

```
// in Scala

import org.apache.spark.sql.functions.{expr, pow}

val fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) +
```

5

```
df.select(expr("CustomerId"),
fabricatedQuantity.alias("realQuantity")).show(2)
```

in Python

```
from pyspark.sql.functions import expr, pow

fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5

df.select(expr("CustomerId"),
fabricatedQuantity.alias("realQuantity")).show(2)
```

```
+-----+-----+
```

```
|CustomerId|    realQuantity|
```

```
+-----+-----+
```

```
| 17850.0|239.08999999999997|
```



```
| 17850.0|    418.7156|
```

```
+-----+-----+
```

注意我们可以把列相乘因为它们都是数值。当然，我们也可以根据需要进行加减操作。事

实上，我们还可以将所有这些操作使用 SQL 表达式来完成:

// in Scala

```
df.selectExpr(  
    "CustomerId",  
    "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)
```

in Python

```
df.selectExpr(  
    "CustomerId",  
    "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)
```

-- in SQL

```
SELECT customerId, (POWER((Quantity * UnitPrice), 2.0) + 5) as  
realQuantity  
FROM dfTable
```

另一个常见的数值操作：四舍五入操作。如果你想四舍五入到一个整数，通常你可以将值转换成一个整数，这样就可以了。然而，Spark 还具有更详细的函数来显式地执行此操作并达到一定的精度。在下面的例子中，我们四舍五入到小数点后一位：

// in Scala

```
import org.apache.spark.sql.functions.{round, bround}  
  
df.select(round(col("UnitPrice"), 1).alias("rounded"),  
col("UnitPrice")).show(5)
```

round()操作是向上四舍五入。bround()操作是向下舍去小数。

```
// in Scala
```

```
import org.apache.spark.sql.functions.lit
```

```
df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)
```

```
# in Python
```

```
from pyspark.sql.functions import lit, round, bround
```

```
df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)
```

```
-- in SQL
```

```
SELECT round(2.5), bround(2.5)
```

```
+-----+-----+
```

```
|round(2.5, 0)|brround(2.5, 0)|
```

```
+-----+-----+
```

```
|      3.0|      2.0|
```



```
|      3.0|      2.0|
```

```
+-----+-----+
```

另一个数值任务是计算两列的相关性。例如，我们可以看到两列的皮尔逊相关系数，看看便宜的东西是否大量购买。我们可以通过一个函数以及 DataFrame 统计方法来实现这一点:

// in Scala

```
import org.apache.spark.sql.functions.{corr}
```

```
df.stat.corr("Quantity", "UnitPrice")
```

```
df.select(corr("Quantity", "UnitPrice")).show()
```

in Python

```
from pyspark.sql.functions import corr
```

```
df.stat.corr("Quantity", "UnitPrice")
```

```
df.select(corr("Quantity", "UnitPrice")).show()
```

-- in SQL

```
SELECT corr(Quantity, UnitPrice) FROM dfTable
```

```
+-----+
```

```
|corr(Quantity, UnitPrice)|
```

```
+-----+
```

```
◀| -0.04112314436835551|
```

```
+-----+
```


6.5. 使用 String 类型

// in Scala

```
import org.apache.spark.sql.functions.{lower, upper}

df.select(col("Description"),
  lower(col("Description")),
  upper(lower(col("Description")))).show(2)
```

in Python

```
from pyspark.sql.functions import lower, upper

df.select(col("Description"),
  lower(col("Description")),
  upper(lower(col("Description")))).show(2)
```

-- in SQL

```
SELECT Description, lower(Description), Upper(lower(Description)) FROM dfTable
```

```
+-----+-----+-----+
```

```
| Description| lower(Description)|upper(lower(Description))|
```

```
+-----+-----+-----+
```

```
|WHITE HANGING HEA...|white hanging hea...| WHITE HANGING HEA...|
```

```
| WHITE METAL LANTERN| white metal lantern| WHITE METAL LANTERN|
```

+-----+-----+-----+

另一个简单的任务是在字符串周围添加或删除空格。你可以使用 `lpad`、`ltrim`、`rpadd` 和 `rtrim`、`trim`：

// in Scala

```
import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad, trim}
```

```
df.select(
  ltrim(lit(" HELLO ")).as("ltrim"),
  rtrim(lit(" HELLO ")).as("rtrim"),
  trim(lit(" HELLO ")).as("trim"),
  lpad(lit("HELLO"), 3, " ").as("lp"),
  rpad(lit("HELLO"), 10, " ").as("rp")).show(2)
```

in Python

```
from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim
```

```
df.select(
  ltrim(lit(" HELLO ")).alias("ltrim"),
  rtrim(lit(" HELLO ")).alias("rtrim"),
  trim(lit(" HELLO ")).alias("trim"),
  lpad(lit("HELLO"), 3, " ").alias("lp"),
  rpad(lit("HELLO"), 10, " ").alias("rp")).show(2)
```

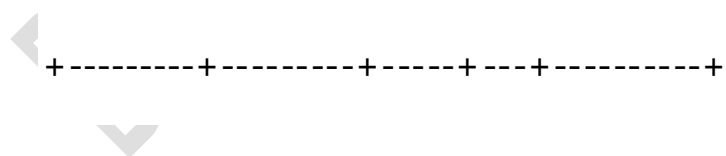
-- in SQL

SELECT

```
ltrim(' HELLOOOO '),
rtrim(' HELLOOOO '),
trim(' HELLOOOO '),
lpad('HELLOOOO ', 3, ' '),
rpad('HELLOOOO ', 10, ' ')
```

FROM dfTable

```
+-----+-----+-----+---+-----+
| ltrim|  rtrim| trim| lp|      rp|
+-----+-----+-----+---+-----+
|HELLO | | HELLO|HELLO| HE|HELLO |
|HELLO | | HELLO|HELLO| HE|HELLO |
```



注意，如果 lpad 或 rpad 取的数字小于字符串的长度，它将总是从字符串的右侧删除值。

6.5.1. 正则表达式

可能最常执行的任务之一是搜索另一个字符串的存在，或者用另一个值替换所有提到的字符串。这通常是通过一个称为正则表达式的工具来实现的，该工具存在于许多编程语言中。

正则表达式使用户能够指定一组规则，以便从字符串中提取值或用其他值替换它们。

Spark 利用了 Java 正则表达式的强大功能。Spark 中有两个关键函数，您需要它们来执行正则表达式任务: `regexp_extract` 和 `regexp_replace`。两个函数分别提取值和替换值。

让我们探讨如何使用 `regexp_replace` 函数替换 `description` 列中的替换颜色名称:

```
// in Scala
```

```
import org.apache.spark.sql.functions.regexp_replace
```

```
val simpleColors = Seq("black", "white", "red", "green", "blue")
```

```
val regexString = simpleColors.map(_.toUpperCase).mkString("|")
```

```
// 在正则表达式语法中, |表示 “或”
```

```
df.select(
```

```
  regexp_replace(col("Description"), regexString,
```

```
  "COLOR").alias("color_clean"),
```

```
  col("Description")).show(2)
```

```
-- in SQL
```

```
SELECT
```

```
  regexp_replace(Description, 'BLACK|WHITE|RED|GREEN|BLUE',
```

```
  'COLOR') as
```

```
  color_clean, Description
```

```
FROM dfTable
```

```
+-----+-----+
```

color_clean	Description
+-----+-----+	
COLOR HANGING HEA...	WHITE HANGING HEA...
COLOR METAL LANTERN WHITE METAL LANTERN	
+-----+-----+	

6.6. 使用 Dates and Timestamps 类型

日期和时间是编程语言和数据库中经常遇到的挑战。总是需要跟踪时区，确保格式正确和有效。Spark 通过明确地关注两种与时间相关的信息，尽力使事情保持简单有专门关注日历日期的 `date` 和包含日期和时间信息的 `timestamp`。正如我们在当前数据集中看到的那样，Spark 将尽力正确地识别列类型，包括在启用 `inferSchema` 时的日期和时间戳。

我们可以看到，这在我们当前的数据集上运行得非常好，因为它能够识别和读取我们的日期格式，而不需要我们为它提供一些规范。如前所述，处理日期和时间戳与处理字符串密切相关，因为我们经常将时间戳或日期存储为字符串，并在运行时将它们转换为日期类型。这在处理数据库和结构化数据时不太常见，但在处理文本和 CSV 文件时更常见。我们将很快对此进行实验。

注意

不幸的是，在处理日期和时间戳时有很多需要注意的地方，特别是在处理时区时。在 2.1 版本和之前，如果您正在解析的值中没有显式地指定时区，Spark 将根据机器的时区进行解析。如果需要，可以通过设置本地会话的时区。SQL 配置：

`spark.conf.sessionLocalTimeZone` 。这应该根据 Java 时区格式进行设置。

```
df.printSchema()
```

```
root
```

```
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)
```

尽管 Spark 会尽力读取日期或时间。然而，有时我们无法处理格式奇怪的日期和时间。理解您将要应用的转换的关键是确保您确切地知道在方法的每个给定步骤中有什么类型和格式。另一个常见的“陷阱”是 Spark 的 `TimestampType` 类只支持二级精度，也就是说，如果你的时间是毫秒或微秒，你需要通过 long 时间的操作来解决这个问题。在强制使用 `TimestampType` 时，将删除精度。

Spark 可以对任何给定时间点的格式进行一些特殊处理。在进行解析或转换时，务必明确说明这样做是没有问题的。最后，Spark 还在使用 Java 日期和时间戳，因此符合这些标准。让我们从基础开始，获取当前日期和当前时间戳：

```
// in Scala
```

```
import org.apache.spark.sql.functions.{current_date,
```

```

current_timestamp}

val dateDF = spark.range(10)

  .withColumn("today", current_date())

  .withColumn("now", current_timestamp())

dateDF.createOrReplaceTempView("dateTable")

dateDF.printSchema()

```

```
root
```

```

|-- id: long (nullable = false)
|-- today: date (nullable = false)
|-- now: timestamp (nullable = false)

```

现在我们有了一个简单的 DataFrame，让我们从今天开始加减 5 天。这些函数接收列名和加上或减去的天数作为参数：

```
// in Scala
```

```

import org.apache.spark.sql.functions.{date_add, date_sub}

dateDF.select(date_sub(col("today"), 5), date_add(col("today"),
5)).show(1)

```

```
-- in SQL
```

```
SELECT date_sub(today, 5), date_add(today, 5) FROM dateTable
```

```
+-----+-----+
```

```
|date_sub(today, 5)|date_add(today, 5)|
```

```

+-----+-----+
| 2017-06-12| 2017-06-22|
+-----+-----+

```

另一个常见的任务是查看两个日期之间的差异。我们可以使用 `datediff` 函数来实现这一点，该函数将返回两个日期之间的天数。大多数时候我们只关心天数，因为每个月的天数不同，还有一个函数，`months_between`，它给出两个日期之间的月数：

// in Scala

```
import org.apache.spark.sql.functions.{datediff, months_between,
to_date}
```

```
dateDF.withColumn("week_ago", date_sub(col("today"), 7))
      .select(datediff(col("week_ago"), col("today"))).show(1)
```

```
dateDF.select(
  to_date(lit("2016-01-01")).alias("start"),
  to_date(lit("2017-05-22")).alias("end"))
    .select(months_between(col("start"), col("end"))).show(1)
```

-- in SQL

```
SELECT to_date('2016-01-01'), months_between('2016-01-01', '2017-
01-01'),
datediff('2016-01-01', '2017-01-01')
```


FROM dateTable

```
+-----+
```

```
|datediff(week_ago, today)|
```

```
+-----+
```

```
|                -7|
```

```
+-----+
```

```
+-----+-----+
```

```
|months_between(start, end)|
```

```
+-----+
```

```
|          -16.67741935|
```

```
+-----+
```

注意，我们引入了一个新函数:to_date 函数。to_date 函数允许您将字符串转换为日期，可以选择指定格式。我们以 Java SimpleDateFormat 指定我们的格式，如果您使用此函数，该格式将非常重要，值得您参考:

```
// in Scala
import org.apache.spark.sql.functions.{to_date, lit}

spark.range(5).withColumn("date", lit("2017-01-01"))

.select(to_date(col("date"))).show(1)
```

如果 Spark 不能解析日期，则不会抛出错误;而是返回 null。在更大的管道中，这可能有点棘手，因为您可能会以一种格式期待您的数据，并在另一种格式中获取数据。为了说明这一点，让我们看一下日期格式从 year-month-day 到 year-day-month 的转换。

```
dateDF.select(to_date(lit("2016-20-12")),to_date(lit("2017-12-11"))).show(1)

+-----+-----+
|to_date(2016-20-12)|to_date(2017-12-11)|
+-----+-----+
|          null|      2017-12-11|
+-----+-----+
```

我们发现这对 bug 来说是一种特别棘手的情况，因为有些日期可能匹配正确的格式，而另一些则不匹配。在前面的示例中，请注意第二个日期如何显示为 12 月 11 日，而不是 11 月 12 日。Spark 不会抛出错误，因为它不知道日期是混在一起的还是特定的行是不正确的。

让我们一步一步地修复这个问题，并想出一个健壮的方法来完全避免这些问题。第一步是记住，我们需要根据 Java SimpleDateFormat 标准指定日期格式。

我们将使用两个函数来修复这个问题：to_date 和 to_timestamp。前者可选地要求格式，而后者需要格式：

```
// in Scala
```

```
import org.apache.spark.sql.functions.to_date

val dateFormat = "yyyy-dd-MM"

val cleanDateDF = spark.range(1).select(
  to_date(lit("2017-12-11"), dateFormat).alias("date"),
  to_date(lit("2017-20-12"), dateFormat).alias("date2"))

cleanDateDF.createOrReplaceTempView("dateTable2")
```

```
-- in SQL
```

```
SELECT to_date(date, 'yyyy-dd-MM'), to_date(date2, 'yyyy-dd-MM'),
to_date(date)
FROM dateTable2
```

```
+-----+-----+
|  date|  date2|
+-----+-----+
|2017-11-12|2017-12-20|
+-----+-----+
```

现在让我们使用 to_timestamp 的一个例子，它总是需要指定一个格式：

```
// in Scala
```

```
import org.apache.spark.sql.functions.to_timestamp

cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()
```

```
-- in SQL
```

```
SELECT to_timestamp(date, 'yyyy-dd-MM'), to_timestamp(date2,
'yyyy-dd-MM')
FROM dateTable2
```

```
+-----+
|to_timestamp(`date`, 'yyyy-dd-MM')|
+-----+
```

```
|      2017-11-12 00:00:00|  
+-----+
```

在日期和时间戳之间进行转换在所有语言中都很简单——在 SQL 中，我们采用以下方法：

```
-- in SQL
```

```
SELECT cast(to_date("2017-01-01", "yyyy-dd-MM") as timestamp)
```

在我们有了正确的格式和类型的日期或时间戳之后，比较它们实际上是很容易的。我们只需要确保使用日期/时间戳类型或者根据正确的 yyyy - mm -dd 格式指定我们的字符串，如果我们在比较日期：

```
cleanDateDF.filter(col("date2") > lit("2017-12-12")).show()
```

一个小问题是，我们还可以将其设置为字符串，它将解析为文本：

```
cleanDateDF.filter(col("date2") > "2017-12-12").show()
```

6.7. 数据处理中的 null 值

作为一种最佳实践，应该始终使用 nulls 来表示 DataFrames 中丢失的或空的数据。

Spark 可以比使用空字符串或其他值更优化地使用 null 值。在 DataFrame 范围内，与 null 值交互的主要方式是在 DataFrame 上使用 na 子包。还有几个函数用于执行操作并显式地指定 Spark 应该如何处理 null 值。有关更多信息，请参见第 5 章(我们将在其中讨论排序)，并参考“与 boolean 一起工作”。

注意

Nulls 是所有编程中具有挑战性的部分，Spark 也不例外。在我们看来，显式处理 null 值总是比隐式处理好。例如，在本书的这一部分中，我们看到了如何将列定义为具有空类型。然而，这是有问题的。当我们声明一个列不可为 null 时，实际上并没有强制执行。重申一下，当您定义一个模式时，其中所有列都被声明为 not null，Spark 将不会强制执行该模式，并且很高兴地让空值进入该列。是否为 null 的标识，只是为了帮助触发 SQL 优化以处理该列。如果列中不应该有 null 值，则可能会得到不正确的结果或看到难以调试的奇怪异常。

可以对空值做两件事：可以显式地删除空值，或者可以用值(全局或每列)填充空值。现在让我们来做实验。

Coalesce

Spark 包含一个函数，允许您使用 coalesce 函数从一组列中选择第一个非空值。在这种情况下，没有空值，因此它只返回第一列：

```
// in Scala

import org.apache.spark.sql.functions.coalesce

df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

ifnull, nullif, nvl, 和 nvl2

您还可以使用其他几个 SQL 函数来实现类似的功能。

- ifnull: 允许您在第一个值为 null 时选择第二个值，并默认为第一个值
- nullif: 如果两个值相等则返回 null，否则返回第二个值
- nvl: 如果第一个值为 null，则返回第二个值，但默认为第一个值

- `nvl2`: 如果第一个值不是 `null` , 则返回第二个值; 否则, 它将返回最后指定的值(下面示例中的 `else_value`)

*-- in SQL***SELECT**`ifnull(null, 'return_value'),``nullif('value', 'value'),``nvl(null, 'return_value'),``nvl2('not_null', 'return_value', "else_value")`**FROM dfTable LIMIT 1**

```

+-----+---+-----+-----+
|      a| b|      c|      d|
+-----+---+-----+-----+
|return_value|null|return_value|return_value|
+-----+---+-----+-----+

```

当然，我们也可以在 DataFrames 的 select 表达式中使用它们。

drop

最简单的函数是 drop，它删除包含 null 的行。默认情况是删除任何值为空的行：

`df.na.drop()``df.na.drop("any")`

在 SQL 中，我们必须逐列执行此操作：

*-- in SQL***SELECT * FROM dfTable WHERE Description IS NOT NULL**

将 “any” 指定为参数将删除任一个字段为 null 的行。使用 “all” 只在该行的所有值为 null 或 NaN 时才删除行：

```
df.na.drop("all")
```

我们也可以通过传入列数组，将此应用于某些列集：

```
// in Scala
```

```
df.na.drop("all", Seq("StockCode", "InvoiceNo"))
```

fill

使用 fill 函数，可以用一组值填充一个或多个列。这可以通过指定一个 map 来完成——它是一个特定的值和一组列。例如，要在 String 类型的列中填充所有空值，可以指定以下内容：

```
df.na.fill("All Null values become this string")
```

对于类型为 Integer 的列，我们也可以使用 `df.na.fill(5:Integer)`，对于类型为 Double 的列，可以使用 `df.na.fill(5:Double)`。

要指定列，我们只需传入列名称数组，就像前面示例中的那样

```
// in Scala
```

```
df.na.fill(5, Seq("StockCode", "InvoiceNo"))
```

我们还可以使用 Scala Map 来实现这一点，其中键是列名称，值是我们希望用来填充空值的值：

```
// in Scala
```

```
val fillColValues = Map("StockCode" -> 5, "Description" -> "No Value")  
  
df.na.fill(fillColValues)
```


replace

唯一的要求是该值与原始值的类型相同:

```
// in Scala
```

```
df.na.replace("Description", Map("" -> "UNKNOWN"))
```

ordering

正如我们在第 5 章中讨论的，您可以使用 `asc_nulls_first`、`desc_nulls_first`、`asc_nulls_last` 或 `desc_nulls_last`，来指定您希望在有序的 `DataFrame` 中显示空值的位置

6.8. 使用复杂类型

复杂类型可以帮助您公司组织数据，从而使希望解决的问题更有意义。有三种复杂类型:

`structs`, `arrays`, 和 `maps`。

`structs`：您可以将结构看作是 `DataFrames` 中的 `DataFrames`。一个示例将更清楚地说明这一点。我们可以通过在查询的括号中封装一组列来创建一个结构体:

```
df.selectExpr("(Description, InvoiceNo) as complex", "*")
```

```
df.selectExpr("struct(Description, InvoiceNo) as complex", "*")
```

```
// in Scala
```

```
import org.apache.spark.sql.functions.struct
```

```
val complexDF = df.select(struct("Description",
```

```
"InvoiceNo").alias("complex"))  
  
complexDF.createOrReplaceTempView("complexDF")
```

现在我们有了一个包含 `complex` 列的 `DataFrame`。我们可以像查询另一个 `DataFrame` 一样查询它，唯一的区别是我们使用点语法来查询它，或者使用列方法 `getField`：

```
complexDF.select("complex.Description")  
  
complexDF.select(col("complex").getField("Description"))
```

我们还可以使用 `*` 查询结构中的所有值。这将把所有列显示到顶级 `DataFrame`：

```
complexDF.select("complex.*")  
  
-- in SQL  
  
SELECT complex.* FROM complexDF
```

第三部分 Low-Level API

第十二章 弹性分布式数据集 RDD

本书的前一部分介绍了 Spark 的结构化 api。在几乎所有的计算场景中，您都应该优先使用这些 api。话虽如此，有时只使用 Higher-Level API 的无法解决你的问题。对于这些情

况，您可能需要使用 Spark 的底层 api，特别是弹性分布式数据集(RDD)、SparkContext，以及分布式共享变量，如累加器和广播变量。接下来的章节将介绍这些 api 以及如何使用它们。

警告

如果你是 Spark 新手，请不要从这个地方学习。从结构化的 api 开始，您将更快地提高生产率!

13.1.什么是 Low-Level api

有两组 low-level api:一组用于操作分布式数据(RDDs)，另一组用于分发和操作分布式共享变量(广播变量和累积器)。

13.1.1. 何时使用 low-level api ?

您应该在以下三种情况下使用较低级别的 api:

- 您需要一些在高级 api 中无法找到的功能; 例如，如果您需要非常严格地控制跨集群的物理数据放置。
- 您需要维护一些使用 RDDs 编写的遗留代码库。
- 您需要执行一些自定义的共享变量操作。我们将在第 14 章详细讨论共享变量。

这些是您应该使用 low-level API 的原因，但是，理解这些 API 仍然很有帮助，因为所有的 Spark 工作负载都可以编译成这些基本元素。当您调用 DataFrame 转换时，它实际上只是一组 RDD 转换。当开始调试越来越复杂的工作负载时，这种理解可以简化您的任务。

即使您是一名高级开发人员，希望能从 Spark 中获得最大的好处，我们仍然建议您关注结构化 api。但是，有时您可能想要使用一些较低级的工具来完成您的任务。您可能需要使用这些 api 来使用一些遗留代码，实现一些自定义分区器，或者在数据管道执行过程中更新和跟踪变量的值。这些工具提供了更细粒度的控制。

13.1.2. 如何使用低级 api?

SparkContext 是低级 API 功能的入口点。您可以通过 SparkSession 访问它，SparkSession 是在 Spark 集群中执行计算的工具。我们将在第 15 章进一步讨论，但是现在，您只需知道您可以通过以下调用访问 SparkContext:

```
spark.sparkContext
```

13.2.关于 RDD

RDDs 是 Spark 1.X 系列版本中的主要 API。在 Spark2.x 版本中仍然可以使用，但并不常用。然而，正如我们在本书前面所指出的，几乎所有您运行的 Spark 代码，无论是 DataFrames 还是 Datasets，都可以编译为 RDD。在本书的下一部分介绍的 Spark UI 中，也描述了关于 RDDs 的作业执行情况。因此，您至少应该对 RDD 是什么以及如何使用它有一个基本的了解。

简而言之，RDD 表示一个不可变的、分区的记录集合，可以并行操作。但是，与 DataFrames（每个记录都是结构化的行，包含字段信息（schema））不同，在 RDDs 中，记录只是程序员选择的 Java、Scala 或 Python 对象。

RDDs 给您完全的控制，因为 RDD 中的每个记录都是一个 Java 或 Python 对象。您可以在这些对象中用任何你想要的格式存储任何你想要的东西。这给了你很大的力量，但不是没有潜在的问题。每个操作和值之间的交互都必须手动写代码定义，这意味着无论你想要执行什么任务，你都必须“重新发明轮子”。此外，优化还需要更多的手工作，因为 Spark 并不像使用结构化 api 那样理解您的记录的内部结构。例如，Spark 的结构化 api 会自动将数据存储存储在优化的、压缩的二进制格式中，从而实现相同的空间效率和性能，您还需要在对象中实现这种格式，并在所有底层操作中进行计算。类似地，在 Spark SQL 中自动出现的重新排序过滤器和聚合等优化需要手工实现。出于这个原因和其他原因，我们强烈建议在可能的情况下使用 Spark 结构化 api。

RDD API 与我们在书的前一部分看到的 Dataset 相似，只是 RDDs 没有存储在结构化的数据引擎中，或者说没有使用结构化的数据引擎操作数据。但是，在 RDDs 和 Dataset 之间来回转换是很简单的，所以您可以使用两个 API，来利用每个 API 的优缺点。我们将在本书的这一部分展示如何做到这一点。

13.2.1. RDD 的类型

如果您查看 Spark 的 API 文档，您会注意到有很多 RDD 的子类。在大多数情况下，这些是 DataFrame API 用于创建优化的物理执行计划的内部表示。然而，作为一个 Spark 开发人员/用户，您可能只会创建两种类型的 RDDs：“通用的”RDD 类型或提供附加功能的键值 RDD，例如按键聚合。就您的目的而言，这将是唯一重要的两种类型的 RDDs。它们都只是表示对象的集合，但是键值 RDDs 有特殊的操作，以及按键进行自定义分区概念。

让我们正式定义 RDD。在内部，每个 RDD 的特征包括五个主要属性：

- 一个分区列表
- 一个计算方法，用于计算每一个数据分区
- 一个 RDD 的依赖列表
- 一个 Partitioner（分区器），对于键值对 RDD，此项可选
- 一个数据分区计算的首选位置列表（例如，用于 Hadoop 分布式文件系统[HDFS]文件的块位置。），此项可选

注意

Partitioner 可能是你在代码中使用 RDDs 的核心原因之一。如果您正确地使用它，指定您自己的自定义分区可以给您显著的性能和稳定性改进。在第 13 章中，当我们引入键值对 RDDs 时，这将更深入地讨论。

上述的 5 个属性决定了 Spark 计划调度和执行用户程序的能力。不同类型的 RDDs 实现各自版本的上述属性，允许您定义新的数据源。

RDDs 遵循我们在前几章中看到的完全相同的 Spark 编程范式。提供了 transformation 操作（具有延迟计算特性）和 action 操作（触发真正的计算），以分布式方式操作数据。这些工作与在 DataFrames 和 Dataset 数据集上的 transformation 和 action 操作相同。但是，在 RDDs 中没有“rows”的概念；单个记录只是原始的 Java/Scala/Python 对象。您可以手动操作这些，而不是使用结构化 api 中的函数库。

RDD api 在 Python 和 Scala 和 Java 中都可以使用。对于 Scala 和 Java，性能在很大程度上是相同的，巨大的性能开销，产生在操作原始对象时。然而，在使用 RDDs 时，Python 可能会损失大量的性能。运行 Python RDDs 等于逐行运行 Python 用户定义函数(udf)。正如我们在第六章中看到的。我们将数据序列化到 Python 进程中，在 Python 中对其进行操作，然后将其序列化回 Java 虚拟机(JVM)。这将导致 Python RDD 操作的高开销。尽管过去有很多人用它们来运行生产代码，但我们建议在 Python 中使用结构化 api，如果不是绝对必要的话，不要使用 RDD Low-Level API。

13.2.2. 何时使用 RDD

一般来说，除非您有一个非常非常具体的原因，否则您不应该手动创建 RDDs。它们是一个更低级的 API，提供了大量的功能，但也缺少结构化 API 中可用的许多优化。对于绝大多数的用例来说，DataFrames 将比 RDDs 更高效、更稳定、更有表现力。

为什么要使用 RDDs 的最可能原因是您需要对数据的物理分布(数据的自定义分区)进行细粒度的控制。

13.2.3. Datasets 和 case class 类型的 RDDs

我们在网上看到这个问题，发现它很有趣：Case class 类型的 RDD 和 Dataset 有什么不同？不同之处在于，Dataset 仍然可以利用结构化 api 所提供的丰富的功能和优化。对于 Dataset，您不需要在 JVM 类型或 Spark 类型之间进行选择，您可以选择最容易做的或最灵活的。

13.3. 创建 RDD

现在我们讨论了一些关键的 RDD 属性，让我们开始应用它们，以便更好地理解如何使用它们。

13.3.1. DataFrames、dataset 和 RDDs 之间的互操作

获取 RDDs 的最简单方法之一是来自现有的 DataFrame 或 Dataset。将这些数据转换为 RDD 很简单：只需在任何这些数据类型上使用 rdd() 方法即可。您会注意到，如果从 Dataset[T] 转换到 RDD，您将获得适当的本机类型 T(记住这只适用于 Scala 和 Java)：

```
// in Scala: converts a Dataset[Long] to RDD[Long]  
  
spark.range(500).rdd
```

因为 Python 没有 Dataset—它只有 dataframe—您将得到类型 Row 的 RDD:

```
# in Python  
  
spark.range(10).rdd
```

要对这个数据进行操作，您需要将这个 Row 对象转换为正确的数据类型或从中提取值，如下面的示例所示。这是一个 RDD 类型 Row:

```
// in Scala  
spark.range(10).toDF().rdd.map(rowObject => rowObject.getLong(0))  
  
# in Python  
spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

您可以使用相同的方法从 RDD 中创建 DataFrame 或 Dataset。你所需要做的就是 在 RDD 上调用 toDF 方法：

```
// in Scala  
  
spark.range(10).rdd.toDF()  
  
# in Python  
  
spark.range(10).rdd.toDF()
```

这个命令创建一个 Row 类型的 RDD。row 是 Spark 用于在结构化 api 中表示数据的内部 Catalyst 格式。这个功能使您可以在结构化和低级 api 之间进行跳转，如果它适合您的用例。(我们在第 13 章讨论过这个问题。)

RDD API 与第 11 章中的 DatasetAPI 非常相似，因为它们非常相似(RDDs 是 Dataset 的底层表示)，但是 RDD 没有结构化 API 所做的许多方便的功能和接口。

13.3.2. 从本地集合创建 RDD

要从集合中创建一个 RDD，您需要使用 `SparkContext`(在 `SparkSession` 中)上的 `parallelize` 方法。这将单个节点集合变为并行集合。在创建这个并行集合时，您还可以显式地声明您想要分配该数组的分区数量。在这种情况下，我们创建两个分区：

```
// in Scala
```

```
val myCollection = "Spark The Definitive Guide : Big Data Processing  
Made Simple"
```

```
.split(" ")
```

```
val words = spark.sparkContext.parallelize(myCollection, 2)
```

```
# in Python
```

```
myCollection = "Spark The Definitive Guide : Big Data Processing  
Made Simple"
```

```
.split(" ")
```

```
words = spark.sparkContext.parallelize(myCollection, 2)
```

另外一个特性是，您可以根据给定的名称在 Spark UI 中显示这个 RDD：

```
// in Scala
```

```
words.setName("myWords")
```

```
words.name // myWords
```

```
# in Python
```

```
words.setName("myWords")
```

```
words.name() # myWords
```


13.3.3. 从数据源中创建 RDD

尽管可以从数据源或文本文件创建 RDDs，但使用 Data Source api 通常更可取。RDDs 不像 DataFrames 那样有“Data Source api”的概念；它们主要定义它们的依赖结构和分区列表。我们在第 9 章中看到的数据源 API 几乎总是一种更好的数据读取方式。也就是说，您也可以使用 sparkContext 读取数据作为 RDDs。例如，让我们逐行读取一个文本文件：

```
spark.sparkContext.textFile("/some/path/withTextFiles")
```

这将创建一个 RDD，其中 RDD 中的每个记录表示该文本文件中的一行。或者，您可以读取每个文本文件成为单个记录的数据。这里的用例是每个文件都是一个文件，它由一个大的 JSON 对象或一些你将作为个人操作的文档组成：

```
spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")
```

在这个 RDD 中，文件的名称是第一个对象，文本文件的值是第二个字符串对象。

13.4. RDD 操作

你操作 RDDs 的方式和你操作 DataFrame 的方式是一样的。如上所述，核心区别在于您操纵原始 Java 或 Scala 对象而不是 Spark 类型。还缺少可以用来简化计算的辅助函数。相反，您必须定义每个 filter，map 函数、聚合以及您想要作为函数的任何其他操作。

为了演示一些数据操作，让我们使用前面创建的简单 RDD(words)来定义更多的细节。

13.4.1. Transformations

在大多数情况下，许多 transformation 对应了在结构化 api 中提供的功能。正如您使用 DataFrames 和 Datasets 一样，您可以在一个 RDD 上指定 transformation 来创建另一个 RDD。在这样做的过程中，我们将 RDD 定义为对另一个 RDD 的依赖，以及对 RDD 中包含的数据的一些操作。

13.4.1.1. distinct 方法

一个 distinct 的方法调用 RDD，从 RDD 中删除重复数据:

```
words.distinct().count()//结果是 10。
```

13.4.1.2. filter 方法

filter()操作相当于创建一个类似 sql 的 where 子句。您可以在 RDD 中查看我们的记录，并查看哪些匹配了谓词功能。这个函数只需要返回一个布尔类型作为 filter 函数。输入应该是给定的数据行。在下一个示例中，我们过滤 RDD，仅保留以字母 “S” 开头的单词:

```
// in Scala
def startsWithS(individual:String) = {
  individual.startsWith("S")
}

# in Python
def startsWithS(individual):
  return individual.startswith("S")
```

现在我们已经定义了函数，我们过滤一下数据。如果您阅读了第 11 章，这将使您感到非常熟悉，因为我们只是使用了一个函数逐行作用于在 RDD 中的记录。该函数定义为在 RDD 中的每个记录上单独工作:

```
// in Scala
words.filter(word => startsWithS(word)).collect()

# in Python
words.filter(lambda word: startsWithS(word)).collect()
```

这就产生了 *Spark* 和 *Simple* 结果。我们可以看到，像 Dataset API 一样，它返回原生类型。这是因为我们从不强制我们的数据转换为 Row 类型，也不需要再在收集数据之后转换数据。

13.4.1.3. map 方法

将传入 map 方法的函数应用于 rdd 中的每一条记录，并返回在代码中指定的值。

```
// in Scala  
val words2 = words.map(word => (word, word(0), word.startsWith("S")))
```

随后，您可以通过在一个新函数中选择相关的布尔值来对此进行过滤

```
// in Scala  
words2.filter(record => record._3).take(5)
```

13.4.1.4. flatmap 方法

flatMap 方法对 map 方法进行了扩展。有时候，一行数据，经过处理后，可能需要返回多行结果。例如：在单词计数程序中，处理每一行语句的时候，需要返回一个单词集合。因为每一句话，包含多个单词，所以，需要使用 flatmap 函数来处理。

13.4.1.5. sortBy 方法

要对 RDD 进行排序，必须使用 sortBy 方法，就像任何其他 RDD 操作一样，通过指定一个函数从 RDDs 中的对象中提取一个值，然后基于此进行排序。例如，下面的例子以单词长度从最长到最短排序：

```
// in Scala  
words.sortBy(word => word.length() * -1).take(2)
```

13.4.2. action 操作

action 操作触发了 transformation 操作的真正执行。action 操作要么向 driver 程序收集数据，要么向外部数据源写入数据。

13.4.2.1. reduce 方法

可以使用 reduce 方法，为其指定一个函数，然后其会将 RDD 中数据任意个数的数据值合并为一个值。例如，一个 RDD 中包含数字集合，可以使用 reduce 方法将这些数字相加，最后形成一个相加结果值。

```
// in Scala  
spark.sparkContext.parallelize(1 to 20).reduce(_ + _) // 210
```

你也可以用这个来得到一些像我们刚才定义的单词中最长的单词。实现关键是要定义正确的函数：

```
// in Scala  
def wordLengthReducer(leftWord:String, rightWord:String): String = {  
  if (leftWord.length > rightWord.length)  
    return leftWord  
  else  
    return rightWord  
}  
  
words.reduce(wordLengthReducer)
```

13.4.2.2. count 方法

可以使用它计算 RDD 中的数据行数

13.4.2.3. countByValue 方法

该方法计算给定 RDD 中值的数量。但是，它最终将结果集加载到 driver 程序的内存中。您应该仅在预期结果 map 很小的情况下使用此方法，因为整个结果集将加载到驱动程序的内存中。因此，这种方法仅在一个场景中，即行总数较低或不同项的数量较低的情况下才有意义：

```
words.countByValue()
```

13.4.2.4. first 方法

返回结果集的第一个值。

13.4.2.5. max/min 方法

13.4.2.6. take 方法

take 方法及衍生方法(takeOrdered, takeSample, 和 top)，首先扫描一个分区 partition，然后使用该分区的结果来估计满足这个限制所需的额外分区的数量。top()实际上是 takeOrdered()的反面，它根据隐含的顺序选择最上面的值：

```
words.take(5)
words.takeOrdered(5)
words.top(5)
val withReplacement = true
val numberToTake = 6
val randomSeed = 100L
words.takeSample(withReplacement, numberToTake, randomSeed)
```

13.4.3. 保存结果数据到文件(action 操作)

保存结果数据到文件意味着写入纯文本文件。对于 RDDs，不能“save”到传统意义上的数据源。您必须遍历这些分区，以便将每个分区的内容保存到一些外部数据库中。这是一种 low-level 的方法，它揭示了在高级 api 中正在执行的底层操作。Spark 将处理每个分区的数据，并将其写到目的地。

13.4.3.1. saveAsTextFile

要保存到文本文件，只需指定路径和可选的压缩编解码器：

```
words.saveAsTextFile("file:/tmp/bookTitle")
```

要设置压缩编解码器，我们必须从 Hadoop 导入适当的编解码器。您可以在 `org.apache.hadoop.io.compress` 包中找到这些内容。

```
// in Scala
import org.apache.hadoop.io.compress.BZip2Codec
words.saveAsTextFile("file:/tmp/bookTitleCompressed", classOf[BZip2Codec])
```

13.4.3.2. sequenceFile

Spark 最初源于 Hadoop 生态系统，因此它与各种 Hadoop 工具的集成相当紧密。`sequenceFile` 是由二进制键值对组成的平面文件。它广泛用于 MapReduce 程序的输入/输出文件格式。

Spark 可以使用 `saveAsObjectFile` 方法或显式地编写键值对来写入 `sequencefile`，如第 13 章所述：

```
words.saveAsObjectFile("/tmp/my/sequenceFilePath")
```

13.4.3.3. hadoop file

您可以保存各种不同的 Hadoop 文件格式。这些允许您指定类、输出格式、Hadoop 配置和压缩方案(有关这些格式的信息，请阅读 Hadoop:权威指南[O'Reilly, 2015]。)。这些格式在很大程度上是不相关的，除非您是在 Hadoop 生态系统中深入工作，或者使用一些遗留的 mapReduce 作业

13.5. Caching 缓存

DataFrames 和 Dataset 有缓存策略，同样的原则也适用于为缓存 RDDs。您可以缓存或持久化 RDD。

默认情况下，缓存和持久化只处理内存中的数据。

```
words.cache()
```

我们可以将一个存储级别指定为 `singleton` 对象中的任何存储级别：`org.apache.spark.storage`。

`StorageLevel`，它仅是组合：磁盘，内存，堆外内存的任意组合。

随后我们可以查询这个存储级别(在第 20 章讨论持久性时，我们讨论了存储级别):

```
// in Scala
```

```
words.getStorageLevel
```

13.6.checkpointing

DataFrame API 中没有一个特性是检查点的概念。检查点是将 RDD 保存到磁盘的行为，以便将来对该 RDD 的引用指向磁盘上的中间结果分区，而不是从原始源重新计算 RDD。这类似于缓存，只不过它不是存储在内存中，而是存储在磁盘中。这在执行迭代计算时非常有用，类似于缓存的用例:

```
spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")
```

```
words.checkpoint()
```

现在，当我们引用这个 RDD 时，它将来自检查点而不是源数据。这可能是一个有益的优化。

13.7.mapPartitions

13.8.foreachPartition

在本章中，您了解了 RDD api 的基础知识，包括单 RDD 操作。第 13 章涉及更高级的 RDD 概念，如连接和键值 RDDs。

第十三章 高级 RDD 操作

第 12 章探讨了单一 RDD 操作的基础。您学习了如何创建 RDDs 以及为什么要使用它们。此外，我们还讨论了 map、filter、reduce 以及如何创建函数来转换单个 RDD 数据。本章将介绍高级的 RDD 操作，并关注键值 RDDs，这是一种用于操作数据的强大抽象。我们还讨论了一些更高级的主题，比如自定义分区，这是您可能希望首先使用 RDDs 的原因。通过使用自定义分区函数，您可以精确地控制数据如何在集群中布局，并相应地操作该分区。在我们开始之前，让我们总结一下我们将要讨论的主要话题：

- 聚合和键值 RDD
- 自定义分区
- RDD join 操作

让我们使用我们在上一章使用的数据集：

```
// in Scala

val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple" .split(" ")

val words = spark.sparkContext.parallelize(myCollection, 2)
```

14.1.Key-Value 基础 (Key-Value RDDs)

在 RDDs 上有许多方法(...ByKey 结尾的方法)要求您将数据以键值格式表示。每当您在方法名称中看到 ByKey 时，就意味着您只能在 PairRDD 类型上执行此操作。最简单的方法是将当前的 RDD 映射到一个基本的键值结构。这意味着在 RDD 的每个记录中有两个值：

```
// in Scala

words.map(word => (word.toLowerCase, 1))
```


14.1.1. keyBy

前面的示例演示了创建 key 的简单方法。但是，您还可以使用 **keyBy** 函数来实现相同的结果，指定一个从当前值创建 key 的函数。在本例中，你是在用单词中的第一个字母作为 key，将原始记录作为 key 对应的值。

```
// in Scala

val keyword = words.keyBy(word => word.toLowerCase.toSeq(0).toString)
```

14.1.2. mapValues

在您拥有一组键值对之后，就可以开始操作它们了。如果我们有一个元组 tuple，Spark 会假设第一个元素是键，第二个元素是值。在这种格式下，您可以显式地选择 map-over 值(并忽略单个键)。当然，您可以手动进行操作，但是当您知道您要修改这些值时，mapValues 可以帮助你防止错误：

```
// in Scala

keyword.mapValues(word => word.toUpperCase).collect()
```

14.1.3. 提取 key 和 value

当我们使用键值对格式时，我们还可以使用以下方法提取的键或值：

```
// in Scala

keyword.keys.collect()

keyword.values.collect()
```

14.1.4. lookup

您可能想要使用 RDD 的一个有趣的任务是查找特定键的结果。如果我们查找 “s” ，我们将得到与 “Spark” 和 “Simple” 相关的两个值:

```
keyword.lookup("s").foreach(println)
```

14.2. Aggregations 聚合

您可以在普通的 RDDs 或 PairRDDs 上执行聚合，这取决于您使用的方法。让我们用我们的一些数据集来证明这一点:

```
// in Scala

val chars = words.flatMap(word => word.toLowerCase.toSeq)

val KVcharacters = chars.map(letter => (letter, 1))

def maxFunc(left:Int, right:Int) = math.max(left, right)

def addFunc(left:Int, right:Int) = left + right

val nums = sc.parallelize(1 to 30, 5)
```

在您做完准备工作之后，您可以做一些类似 countByKey 的事情，它计算每个键值数据。

14.2.1. countByKey

您可以计算每个键的元素数量，将结果收集到本地 map 中。您还可以使用一个近似方法来实现这一点，这使得您可以在使用 Scala 或 Java 时指定一个超时和信心值:

```
// in Scala

val timeout = 1000L //milliseconds
```

```
val confidence = 0.95

KVcharacters.countByKey()

KVcharacters.countByKeyApprox(timeout, confidence)
```

14.2.2. 理解聚合的实现方式

有几种方法可以创建键值 PairRDDs。然而，这个实现对于工作稳定性来说非常重要。让我们比较两个基本的选择，groupBy 和 reduce。

14.2.2.1. groupByKey

查看 API 文档，您可能会认为 groupByKey 与每个分组的映射是总结每个键的计数的最佳方法：

```
// in Scala

KVcharacters.groupByKey().map(row => (row._1, row._2.reduce(addFunc))).collect()
```

然而，对于大多数情况来说，这是解决问题的错误方法。这里的基本问题是，在将函数应用到它们之前，每个执行器必须在内存中保留给定键的所有值。为什么这个有问题？如果您有大量的键倾斜，一些分区可能会被一个给定键的大量值完全超载，您将会得到 outofmemoryerror。这显然不会引起我们当前数据集的问题，但是它会导致严重的问题。这不一定会发生，但会发生。

当 groupByKey 有意义时，就会出现用例。如果每个键的值都是一致的，并且知道它们将适合于给定的执行器，那么您将会很好。当您这样做的时候，很好地知道你到底在做什么。添加用例的首选方法是 reduceByKey。

14.2.2.2. reduceByKey

因为我们执行的是一个简单的计数，更稳定的方法是执行相同的 flatMap，然后执行 map 将每个字母实例映射到数字 1，然后使用求和函数执行一个 reduceByKey 来收集数组。这个实现更稳定，因为 reduce 发生在每个分区中，不需要将所有东西都放在内存中。这大大提高了操作的速度和操作的稳定性：

```
KVcharacters.reduceByKey(addFunc).collect()
```

`reduceByKey` 方法返回一个组(键)的 RDD，并返回没有进行排序的元素序列。

14.2.3. 其他聚合方法

我们发现，在当今的 Spark 中，用户遇到这种工作负载(或需要执行这种操作)的情况非常少见。当您可以使用结构化 api 执行更简单的聚合时，使用这些极低级别工具的理由并不多。这些函数基本上允许您非常具体地、非常低级地控制在机器集群上如何执行给定的聚合。

14.2.3.1. aggregate

这个函数需要一个 `null` 和 `start` 起始值，然后要求您指定两个不同的函数。在分区内的第一个聚合，跨分区的第二个聚合。开始值将用于两个聚合级别：

```
// in Scala  
  
nums.aggregate(0)(maxFunc, addFunc)
```

此方法确实具有一些性能影响，因为它在 driver 端执行最终的聚合。如果从 executors 端返回到 driver 端的结果集太大，会导致 driver 端内存溢出。此时可以使用 `treeAggregate`。它与 `aggregate` (在用户级)做相同的事情，但以另一种方式进行。它基本上是“下推”一些子聚合(从 executor 到 executor 创建树)，然后在驱动程序上执行最终的聚合。拥有多个级别可以帮助您确保在聚合过程中驱动程序不会耗尽内存。

```
// in Scala  
  
val depth = 3  
  
nums.treeAggregate(0)(maxFunc, addFunc, depth)
```

14.2.3.2. aggregateByKey

这个函数和 aggregate 函数一样，但是它不是按分区进行分区，而是按键执行。开始值和函数遵循相同的属性：

```
/ in Scala  
KVcharacters.aggregateByKey(0)(addFunc, maxFunc).collect()
```

14.2.3.3. combineByKey

您可以指定一个组合器，而不是指定一个聚合函数。这个组合在给定的键上操作，并根据某个函数合并值。然后合并不同的组合输出结果。我们还可以将输出分区的数量指定为自定义输出分区器：

```
// in Scala  
val valToCombiner = (value:Int) => List(value)  
val mergeValuesFunc = (vals:List[Int], valToAppend:Int) => valToAppend :: vals  
val mergeCombinerFunc = (vals1:List[Int], vals2:List[Int]) => vals1 ::: vals2  
// now we define these as function variables  
val outputPartitions = 6  
KVcharacters  
  .combineByKey(  
    valToCombiner,  
    mergeValuesFunc,  
    mergeCombinerFunc,  
    outputPartitions)  
  .collect()
```

14.2.3.4. foldByKey

14.3. CoGroup

CoGroup 操作可以用来合并三个 key-value RDD(Scala 语言)。按 key 值合并。这实际上只是一个基于分组的 RDD 连接。

```
// in Scala

import scala.util.Random

val distinctChars = words.flatMap(word => word.toLowerCase.toSeq).distinct
val charRDD = distinctChars.map(c => (c, new Random().nextDouble()))
val charRDD2 = distinctChars.map(c => (c, new Random().nextDouble()))
val charRDD3 = distinctChars.map(c => (c, new Random().nextDouble()))
charRDD.cogroup(charRDD2, charRDD3).take(5)
```

14.4. Joins

RDDs 与我们在结构化 API 中看到的连接非常相似，尽管 RDD 的 join，你可以进行更细粒度的控制。它们都遵循相同的基本格式：

- 我们想要连接的两个 RDDs，
- 它们应该输出的输出分区数量(可选)
- 自定义分区函数(可选)

我们将在本章后面讨论分区函数。

14.4.1. Inner Join

现在我们将演示一个内部连接。注意我们如何设置我们希望看到的输出分区的数量：

```
// in Scala

val keyedChars = distinctChars.map(c => (c, new Random().nextDouble()))

val outputPartitions = 10
```

```
KVcharacters.join(keyedChars).count()
```

```
KVcharacters.join(keyedChars, outputPartitions).count()
```

我们不会为其他连接提供一个示例，但是它们都遵循相同的基本格式。您可以在第 8 章的概念级别了解以下连接类型:

- `fullOuterJoin`
- `leftOuterJoin`
- `rightOuterJoin`
- `Cartesian`(这又是非常危险的!它不接受连接键,可以有一个巨大的输出。)

14.4.2. zips

最后一种连接实际上并不是一个连接，但是它确实合并了两个 RDDs，所以把它标记为连接是值得的。

`zip` 可以让你“zip”两个 RDDs，假设它们的长度相同。这将创建一个 PairRDD。两个 RDDs 必须具有相同数量的分区和相同数量的元素:

```
// in Scala
```

```
val numRange = sc.parallelize(0 to 9, 2)
```

```
words.zip(numRange).collect()
```

14.5. 分区控制