O'REILLY®

Kudu:
构建高性能实时数据分析存储系统

Getting Started with Kudu:
Perform Fast Analytics on Fast Data

[美] Jean-Marc Spaggiari, Mladen Kovacevic,
Brock Noland, Ryan Bosshart 著
常冰琳 译

中国工信出版集团
電子工業出版社

内容简介

要在Hadoop生态系统中实现数据的快速输入和快速分析，一直以来只有少数可用但是不够完美的解决方案。它们要么以缓慢的数据输入为代价实现快速分析，要么以缓慢的分析为代价实现快速的数据输入。这个问题现在有了解决办法，使用Apache Kudu基于列的数据存储，可以很容易地对快速输入的数据进行快速的分析。这就是本书的内容。

在这本书中，你将学习Kudu设计中的关键概念，以及如何用它构建快速、可扩展和可靠的应用程序。通过实际的示例，你将了解Kudu是如何与其他Hadoop生态系统组件（如Apache Spark、Spark SQL和Impala）集成的。

本书适合大数据系统的架构师、开发者和咨询师阅读。

版权贸易合同登记号 图字：01-2018-4157

图书在版编目（CIP）数据

Kudu：构建高性能实时数据分析存储系统 / （美）吉恩-马克·斯帕加里（Jean-Marc Spaggiari）等著；常冰琳译.—北京：电子工业出版社，2019.3

书名原文：Getting Started with Kudu：Perform Fast Analytics on Fast Data

ISBN 978-7-121-29541-6

Ⅰ.①K…Ⅱ.①吉…②常…Ⅲ.①数据处理Ⅳ.①TP274

中国版本图书馆CIP数据核字（2019）第030205号

责任编辑：许艳

封面设计：Randy Comer 张健

印刷：中国电影出版社印刷厂

装订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开本：787×980 1/16 印张：12.5 字数：160.6千字

版次：2019年3月第1版

印次：2019年3月第1次印刷

定价：69.00元

O’Reilly Media，Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是转瞬即逝的机会，尽管大路也不错。”

——Linux Journal

献给所有没日没夜、绞尽脑汁为那些行为诡异、似乎故意出错的软件做架构设计、开发和咨询的人。

献给我们的家人，他们可能根本不关心技术，但仍然给予我们巨大的耐心和支持，让我们能投入时间和精力写书。没有他们，这本书是不可能完成的。爱你们！

前言

选择存储引擎是实施所有大数据项目时要做的最重要的决定之一，而且更换存储引擎的成本也是最高的。Apache Kudu是Hadoop生态系统中的一个全新存储系统。它的灵活性使我们能够更快地搭建和维护应用程序。在Hadoop开发者的大数据工具箱中，Kudu是一个关键工具。它解决了一些使用目前的Hadoop存储技术很难实现或不可能实现的常见问题。

在这本书中，你将学习Kudu设计中的关键概念，以及如何用它构建快速、可扩展和可靠的Kudu应用程序。通过实际的示例，你将了解Kudu如何与其他Hadoop生态系统组件（如Spark、Spark SQL和Impala）集成。

本书假设读者对Hadoop生态系统组件（如HDFS、Hive、Spark或Impala）有一些使用经验，有Java或Scala编程经验，还有SQL和传统关系型数据库管理系统“使用”经验，熟悉Linux shell。

本书使用的约定

本书中使用了下列排版约定：

斜体（Italic

表示新的术语、URL、电子邮件地址、文件名和文件扩展名。

t等ali宽c 字体（Constant Width）

表示程序清单、在段落内引用的程序变量或函数名、数据库名、数据

类型、环境变量、语句和关键字。

等宽粗体（Constant Width Bold）

表示应该由用户输入的命令或其他文本。

等宽斜体（Constant Width Italic）

表示该内容应由用户提供或由上下文决定。

这个图标表示提示或建议。

这个图标表示一般性注释。

这个图标表示警告或注意。

使用代码示例

可以在https：//github.com/kudu-book/getting-started-kudu下载本书的补充材料（代码示例、练习等）。

这本书就是为了帮助你完成工作的。一般来说，如果本书提供了示例代码，你可以在自己的程序和文档中使用它。除非复制了相当多的示例代码，否则你不需要与我们联系。例如，你编写一个程序，其中使用了本书中的几个代码块，这种情况是不需要获得我们的许可的。但是如果你销售或发行一张光盘，其中包含了O’Reilly图书中的例子，则需要获得许可。引用本书和引用示例代码来回答问题，不需要许可。将大量示例代码嵌入到你的产品文档中则需要获得许可。

我们将十分感激你在使用我们的代码时标注相关的版权信息，但是并不强求你这么做。版权信息通常包括书名、作者、出版商和ISBN。例如：“Getting Started with Kudu by Jean-Marc Spaggiari，Mladen Kovacevic，Brock Noland，Ryan Bosshart（O’Reilly）.Copyright 2018 Jean-Marc Spaggiari，Mladen Kovacevic，Brock Noland，Ryan Bosshart，978-1-491-98025-5”。

如果你觉得自己对代码的使用不在合理使用或以上许可的范围内，请通过permissions@oreilly.com与我们联系。

O’Reilly Safari

Safari（以前的Safari Books Online）是企业、政府、教育者和个人的会员制培训及参考平台。

订阅者可以从一个完全可搜索的数据库中获得来自250 多家出版商的成千上万的书籍、培训视频、互动教程，这些出版商包括O’Reilly Media、Har-vard Business Review、Prentice Hall Professional、Addison-Wesley Profes-sional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley&Sons、Syngress、Morgan Kaufmann、IBM Red-books、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones&Bartlett和Course Technology。

若想获得更多资讯，请访问http：//oreilly.com/safari。

联系我们

请将对本书的评价和发现的问题通过如下地址告知出版社。

美国：

O’Reilly Media,Inc.

1005 Gravenstein Highway North

Sebastopol,CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们提供了本书网页，上面列出了勘误表、示例和其他信息。请通过http：//bit.ly/getting-started-with-kudu访问。

要给本书提意见或者询问技术问题，请发送邮件到bookquestions@oreilly.com。

更多有关书籍、课程、会议和新闻的信息，请见网站http：//www.oreilly.com。

在Facebook找到我们：http：//facebook.com/oreilly。

在Twitter上关注我们：http：//twitter.com/oreillymedia。

在YouTube上观看：http：//www.youtube.com/oreillymedia。

致谢

感谢Apache Kudu社区的帮助，包括Apache Kudu的创始人、代码提交者、贡献者、早期采纳者和用户。感谢我们的技术审稿人David Yahalom、Andy Stadtler和Attila Bukor对细节的关注和反馈。还要感谢非官方的技术审稿人，包括Nipun Parasrampuria、Mac Noland、Sandish Kumar、Tony Foerster、Mike Rasmussen、Jordan Birdsell和Gunaranjan Sundararajan。

Ryan Bosshart和Brock Noland感谢他们在phData和Cloudera的同事对本书的支持和投入。

Mladen Kovacevic感谢他在Cloudera的同事，包括解决方案架构师、工程师、技术支持人员、产品管理人员以及其他人，感谢他们的热情和支持。Mladen还要感谢家人在他写作时给予的耐心、支持和鼓励——没有他们，他是不可能完成写作的！

Jean-Marc Spaggiari要感谢所有在此过程中支持他的人。

第1章 为什么会有Kudu

Kudu为什么重要

随着大数据平台的不断创新和发展，无论是在企业内部还是在云上，新产品的发布都让人应接不暇，大家对此已经很习惯了。不过，2017年在一些大公司和各种业务应用中使用过Kudu之后，我们比以往任何时候都更加确信Kudu很重要，让这个项目加入开源大数据世界是非常有价值的。

原因可以归结为以下三点：

1.大数据仍然是一项很难的技术——由于对数据的需求以及用户规模持续增长，Hadoop和大数据系统相对来说还是很难使用的，其大部分的复杂性来源于存储系统的局限性。而Kudu的局限性就小得多，在我们公司，从前冗长的架构讨论现在变成简单的一句话：“用Kudu就好了。”

2.新的应用场景需要Kudu——Hadoop的应用场景正在发生变化，其中的一个变化是越来越多的应用集中在机器生成的数据和实时分析领域。为了展示这种变化带来的复杂性，我们会讨论几个使用现有大数据存储技术进行实时分析的架构方案，然后讨论Kudu能如何简化这些方案。

3.硬件环境正在发生变化——Hadoop诞生时的很多硬件现在发生了变化。因此，我们有新的契机来为新的硬件环境创建新的存储系统，从而提升性能和应用的灵活性。

在本章中，我们会详细讨论设计Kudu的以上三点动机。如果你已经熟知这些，可以跳到本章的后半部分，在那里我们会讨论Kudu的设计目标，将Kudu和其他大数据存储系统对比。在本章的最后，我们会总结为什么世界需要Kudu这个新的大数据存储系统。

易用性驱动接纳度

分布式系统过去一直既昂贵又难用。21世纪头10年的中期，我们曾经为一家大型媒体和信息提供商工作，负责搭建一个输入、处理、搜索、存储和检索数百TB在线内容数据的平台。搭建这样一个平台是一项庞大的工程，需要数百名工程师和不同的团队来构建平台的各个部分。我们有独立的团队分别负责实现不同功能的分布式系统，包括数据的输入、存储、搜索等。为了实现系统的高可扩展性，我们为关系数据存储、搜索索引和存储系统都做了分片，然后在上层构建精心设计的元数据系统，以保持所有数据都整齐一致。这个平台是建立在昂贵的专利技术之上的，这就把那些想要做同样事情的小型竞争对手挡在了外面。

大约在同一时间，Doug Cutting和Mike Carafella开发了Apache Hadoop。由于他们以及整个Hadoop生态社区的努力，搭建高可扩展性的基础架构不再需要数百万美元的成本和庞大的分布式系统工程师团队。Hadoop带来的最主要的进步在于，只要软件工程师对分布式系统有一定的了解，他就能够搭建一个可扩展的数据平台。对软件工程师来说，Hadoop使得分布式计算变得更简单了。

尽管软件工程师们很高兴，但他们也只是使用大数据的人群中的一小部分。大数据生态持续发展和多样化，从Hive开始到Impala，再到其他各种SQL-on-Hadoop引擎，借助于这些系统，没有编程背景知识的数据分析师和SQL工程师也可以使用Hadoop来分析、处理他们的大规模数据。这些系统提供了大家熟悉的SQL接口，使得一大批用户能够迈进Hadoop的大门。SQL-on-Hadoop之所以重要，是因为与编程语言相比，用它进行数据的分析和处理会更简单。这并不是说SQL使Hadoop更加“简单”。随便找一个从事关系型数据库管理系统（Relational Database Management System，RDBMS）相关工作的人问问，他还是会说尽管SQL使Hadoop更易使用，但二者仍然有很大差距，在使用SQL-on-Hadoop时还有不少需要注意的地方。特别是在项目的早期，工程师和分析师需要决定使用哪个引擎或存储格式最适合他们的应用场景。本章的后面会讨论这些具体的挑战，但是现在我们可以这么说，对于习惯使用SQL的数据工程师来说，SQL-on-Hadoop确实使分布式计算更加简单了。

在写本书时，Hive对Kudu的支持还在停留在HIVE-12971中。但我们还是可以创建Hive表，Impala能访问这些表。

还有很多用户也希望得到这种可扩展性带来的好处，随着机器学习和数据科学的兴起，大数据平台正在为统计学家、数学家以及更广泛的数据分析人员提供服务。这些用户的工作效率取决于他们的分析工具是否能快速迭代和是否具有高性能。数据科学家使用的模型的性能通常取决于数据量的大小，所以使用Hadoop来处理是一个自然的选择。这些用户熟悉SQL，但他们不是“工程师”，对于处理Hadoop的一些琐碎细节不感兴趣，他们期望数据平台有更高的可用性、易用性和性能。

此外，做传统的企业商业智能和分析系统的用户也希望能够使用Hadoop，对性能和SQL系统的成熟度提出了更高的要求。

Hadoop从技术和成本上降低了大数据的门槛，使一个对分布式系统的复杂性知之甚少的程序员或软件工程师能够录入、存储、处理和服务大规模的数据。随着时间的推移，Hadoop进一步发展，现在可以服务更广泛的用户群体。因此，我们都目睹了一个相当明显的事实：数据平台越容易使用，就有越多的人接受它，因此它服务的用户就越多，能够提供的价值也就越大。在过去的10年里，Hadoop就是这样发展的。随着Hadoop变得越来越简单和易于使用，我们看到采纳它的用户和它所创造的价值也越来越多（见图1-1）。

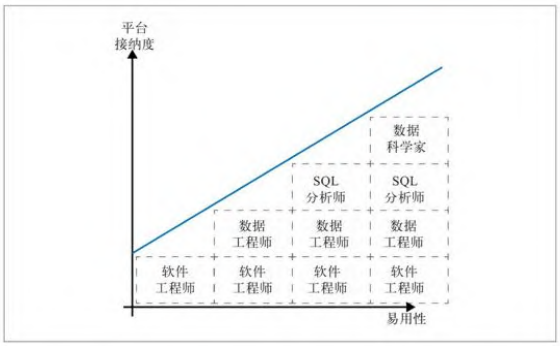

图1-1 Hadoop的接纳度和易用性

Kudu的简单和易用扩大了Hadoop“可应用的市场”，因为它提供了更接近于RDBMS的功能和数据模型。Kudu提供类似于关系型数据库表的存储结构来存储数据，并且允许用户以和关系型数据库相同的方式插入、更新和删除数据。而几乎每一个与数据打交道的人——软件工程师、分析师、ETL（Extract，Transfer and Load）开发人员、数据科学家、统计学家，都熟悉这个模型。此外，Kudu还与Hadoop最新的应用场景需求相匹配。

新的应用场景

Hadoop的应用场景正在扩展，这些应用场景是由几个因素的合力驱动的。第一个因素是大数据的宏观趋势，比如实时分析和物联网（Internet of Things，IoT）的兴起。这类应用很复杂，很难设计好，即使对一些经验丰富的Hadoop开发者来说也有难度。第二个因素是平台的用户群发生了变化，正如上一节所说的那样，新的用户群带来了新的应用场景和对平台新的使用方法。第三个因素是数据的终端用户的期望值提高了。5～10年之前，能做批量处理就可以了，毕竟在此之前我们都没法处理这么大规模的数据。但是到了今天，能够可扩展地存储和处理数据只能算是最起码的要求，用户期望能够得到实时结果，而且平台在各种负载场景下都是高性能的。

让我们来看一看其中的一些应用场景和它们对存储系统的需求。

物联网

到2020年，世界上预计会有200亿～300亿台联网设备。这些设备拥有你所能想象到的各种形状和形式。比如，有数量庞大的“联网货船和客船”，它们都配备了传感器，可以监测从燃料消耗到发动机状态和船只位置的一切信息。现在还有联网的汽车、联网的采矿设备和联网的冰箱。负责提供诸如起搏、除颤和神经刺激等救命治疗手段的植入式医疗设备现在也能够产生并发送数据，这些数据可用于识别患者此时是否发病，或者其体内的植入式设备是否存在问题需要维护或更换。

数十亿台的联网设备带来了明显的可扩展性问题，也使Hadoop成为一个好的选择，但是具体的架构方案却不像你最初想象的那么简单明了。假设我们有一台联网的医疗设备，我们想分析来自该设备的数据。先从一组简单的需求开始：我们希望事件数据从联网设备流入存储层——不管这些事件是什么，然后再以一些方式查询这些数据。首先，我们希望能看到设备现在的情况。比如，在更新设备的软件之后，我们希望能查看设备最近发出的信号，以了解我们的软件更新是否符合预期或者遇到了什么问题。另一种访问设备的模式是分析，我们的数据分析师要研究数据的趋势，以获得新的见解来了解和报告设备的性能，例如，研究电池的使用和优化之类的事情。

为了实现这些基本访问模式（见图1-2），存储层需要具备以下能力：

逐行插入

当应用服务器或者网关设备接收一个事件时，需要将这个事件保存在存储系统中，使其立即可读。

低延迟随机读

对设备进行更新之后，需要分析一部分设备在一段时间内的性能。这意味着需要高效地访问一小块范围内的行。

快速分析和扫描

为了满足报表和即席查询分析的需求，需要能够高效地扫描存储系统中的大规模数据。

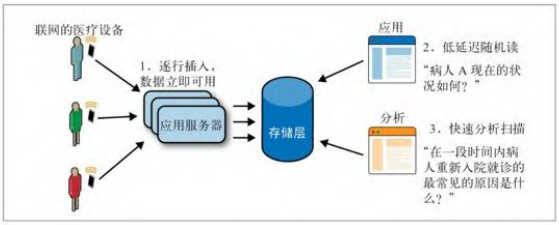

图1-2 物联网的访问模式

现有的实时分析方案

我们通过一个简单的示例来看，在没有Kudu的情况下，要成功实现实时数据流分析应用需要做些什么。

在这个例子中，我们有一个产生连续事件流的数据源（见图1-3）。我们需要近乎实时地存储这些事件，因为用户要求这些事件是马上可用的——这些数据的价值在其被创建时是最大的。这个架构包含一个数据生产者（producer），生产者从源获取事件，然后将它们保存到某个待定的存储层，以便业务用户可以通过SQL来分析，而数据科学家和开发者则可以使用Spark来分析数据。生产者的概念是通用的，它可以是一个Flume agent、一个Spark Streaming作业（job）或者一个独立的应用程序。

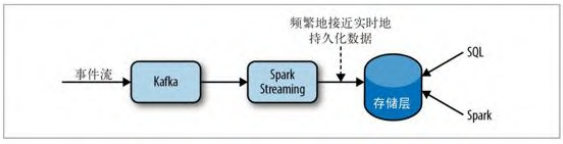

图1-3 简单的实时分析数据流

为这个应用选择一个存储引擎是一件让人相当头疼的事情。我们来看如果使用传统的Hadoop存储引擎的话，有哪些选择。

迭代1：Hadoop分布式文件系统

在第一次迭代中，我们将尽量保持简单，将数据以Av r o文件的格式保存在HDFS（Hadoop Distributed File System，Hadoop分布式文件系统）中。虽然在本例中我们选择了HDFS，但是如果使用其他存储引擎，比如亚马逊的S3，这些原则也适用。Avro是基于行的数据存储格式。基于行的格式非常适用于流式系统，因为在写Parquet这样的列存储格式的数据时，需要在内存中缓冲大量的行。我们创建了一个Hive表，按日期分区，然后生产者持续将小批次（batch）的数据以Avro格式写入HDFS。因为用户希望数据被创建之后马上就能够访问，所以生产者需要频繁地向HDFS写入新文件。例如，我们的生产者是一个Spark Streaming程序，每10s运行一次并生成一个小批次（micro-batch）数据，那么该应用程序也将每10s保存一个新批次数据，以便数据对消费者而言是可用的。

我们将新系统部署在生产集群上。整体的数据流如图1-4所示。

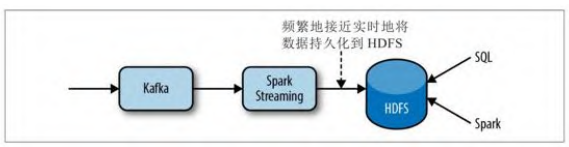

图1-4 使用HDFS的实时数据流

应用被部署之后，过了几天，工单开始陆续来了。运营团队收到用户的报告，说性能太差，他们的作业无法完成。在查看一些作业以后，我们发现我们的HDFS目录下有上万个小文件。结果就是，每读一个小文件都需要HDFS做一次磁盘寻址操作，严重影响了Spark作业和Impala查询的性能。不幸的是，解决这个问题只有一个方法，就是减少小文件的数量，这也是我们接下来要做的工作。

迭代2：HDFS+compaction1

在Google上查询一番之后，我们发现这个问题有一个众所周知的解决方案：添加一个HDFS compaction过程。之前的架构大部分都不需要改动；然而，由于在输入数据的过程中会快速创建小文件，所以需要有一个离线的对小文件做compaction处理的过程。compaction过程将许多小文件改写成数量较少的大文件。这个解决方案看似简单，实际上有一些地方比较复杂。例如，compaction过程需要在一个分区不再活跃时进行，而且不能将结果覆盖到同一个分区中，不然这个分区的数据会暂时“丢失”，此时正在运行的查询可能失败。你可以将结果写到一个新的单独的位置，然后使用HDFS命令将新老位置互换，但即便如此，也需要用两个HDFS命令，而且并不能保证数据的一致性。

复杂性并未止于此。最终的基于HDFS的方案需要有两个单独的“landing”目录（一个主动版本和一个被动版本），以及一个“base”目录用来保存已经做过compaction的数据。现在的架构如图1-5所示（来自Cloudera Engineering博客）。

这个架构包含多个HDFS目录、多个Hive/Impala表以及若干视图。开发者必须开发相应的逻辑：在主动/被动“landing”目录之间切换写，使用compaction过程把数据从“landing”目录移动到“base”目录，修改表和元数据，将其指向新的compact的分区，清理旧数据，并且利用修改视图的技巧来保证用户能够看到一致的数据。

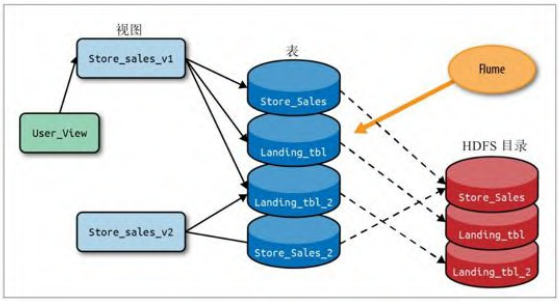

图1-5 使用了HDFS+compaction的实时数据流

尽管你已经创建了能够以较低的延迟处理大规模数据的架构，但是这个方案仍然做了许多妥协。首先，该方案是伪“实时”的，因为其中还是有小文件，只不过数量比原来少了一些。数据写入HDFS的频率越高，小文件的数量就越多，作业处理的效率会急剧下降，导致集群效率和作业性能降低。因此，你可能每分钟只向磁盘写入一次结果，因此这个架构所谓的“实时”只不过是广义的“实时”。此外，因为HDFS没有主键的概念，并且许多流处理系统都有可能产生重复数据（例如，比较“至少一次”＜at-least-once＞语义与“最多一次”＜at-most-once＞ 语义），所以我们需要一个可去重的机制。这样就导致了我们在compaction过程中、在数据库视图里都需要做去重处理。最后，如果有延迟到达的数据，这些数据是不会落在当天的compaction范围之内的，这样就会产生更多的小文件。

迭代3：HBase+HDFS

之前的例子中提到的复杂性和缺点，主要是由于我们试图让HDFS去做不符合其设计初衷或者并非其长项的工作而产生的。还有一个大家可能都熟知的选项，就是我们可以根据两个存储层（存储引擎）各自的优势来组合使用它们，而不是优化单个存储层——只能得到次优的性能和应用特性。这个想法类似于Lambda架构，这种架构中有一个“实时层（speed layer）”和“批处理层（batch layer）”。实时层接收流数据，能够查找最近的数据，可更改，并且最关键的是能够低延迟地读/写数据。对于需要实时数据的客户，实时层可以提供这些数据。在传统的Hadoop生态中，对于实时层，可选择用HBase或者Cassandra来实现，它们都属于“big table”家族。在在线、实时、高并发的应用场景中，HBase和Cassandra发展得很不错；然而，它们的致命弱点是不能提供HDFS和Parquet的快速分析和扫描性能。因此，为了实现快速扫描和分析，必须将数据从实时层转移到HDFS和Parquet的批处理层中。

数据不断流入基于HBase或Cassandra的实时层，当积累了“足够多”的数据时，会有一个“flush”过程，将数据从实时层移到批处理层。通常，当实时层中积累了足以填充一个HDFS分区的数据之后这个过程就开始了。“flush”过程负责从实时层读取数据，将其重写为

Parquet，往HDFS添加一个新分区，然后通知Hive和Impala出现了新分区。因为数据现在存储在两个独立的存储系统中，使用该数据的客户端需要知道数据的这两个存储位置，不然你就必须添加一个服务层将两个层“缝”在一起，让用户从这些细节中抽离出来。最后，我们的架构看起来就有点像图1-6所示的样子。

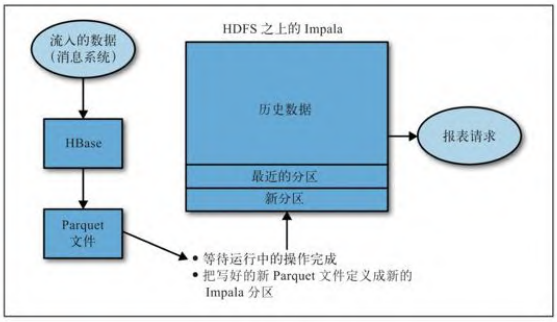

图1-6 使用了HBase和HDFS的实时数据流

这种架构展示了一种可扩展方案，它有很多优点：支持秒级实时数据，支持修改数据和快速扫描。然而，它的开发和运维却十分复杂。开发人员不仅需要开发和维护数据输入部分的代码，还需要开发和维护将数据从实时层移到批处理层的“flush”模块的代码。运维人员现在必须多维护一个存储系统（如果他们之前只用一个存储系统的话），还必须负责对这两个系统的监控、维护和问题定位。最后，由于实时数据和历史数据分散在两个存储系统中，客户端很难得到一个统一的视图。

这些方案要解决的是一个看似简单的问题：对快速生成的数据进行可扩展的、秒级的分析，但是它们都很复杂，其开发、维护或操作都不容易。因此，只有应用场景值得这种高投入时，人们才会使用这些方案。我们观察到，由于太过复杂，上述解决方案很少被采用；在大多数情况下，用户会退而求其次，选择更简单的基于批处理的解决方案。

实时处理

目前，Hadoop生态的一个趋势是，可扩展的实时流式处理系统正在快速发展。你可以看到很多项目，很多公司都在做这个领域的产品。流式处理背后的思想是，在数据流中直接处理数据，而不是将数据保存到存储系统，然后再批量处理。在批量处理中，每个批次

（batch）有明确的开始和结束点，而在流式处理中，数据处理模块是长期运行的，它持续地对流入的小批量数据进行处理。流式处理通常用来观察数据，并且对于如何转换、聚合、筛选数据或者根据数据报警等问题，会立即做出决策。

实时处理可以用于模型评分（model scoring）、复杂事件处理、数据扩充（data enrichment）以及许多其他类型的处理。这些处理模式能够被应用到数据的“时间价值”很高的很多领域，在这些领域里，数据在刚产生时的价值最高，然后其价值会随着时间的推移逐渐降低。许多欺诈检测、医疗分析和物联网应用场景都符合这种模式。限制实时处理技术落地的一个主要因素是其对数据存储系统的挑战性需求。

流式处理通常要有外部上下文（context）。这个上下文可以有很多不同的形式。有些情况下，你需要历史上下文，因为你想知道最近的数据与历史数据相比较是怎样的；另一些情况下，你需要引用其他数据。例如，欺诈检测十分倚重历史数据和参考数据。历史数据将包括过去24小时或过去一周的交易数量等特征。参考数据可能包括如客户的账户信息或IP地址等信息。

尽管诸如Apache Flume、Storm、Spark Streaming和Flink之类的处理框架提供了实时读取和处理事件的能力，但它们还需要倚赖外部系统来存储和访问外部上下文。例如，使用Spark Streaming可以每隔几秒从Kafka读取微批量事件，但如果你希望能够保存结果、读取外部上下文、计算风险评分和更新患者档案，存储系统就要满足以下各种各样的需求：

逐行插入

当事件产生时，需要立即存储，并且可供其他工具和流程做分析。

低延迟随机读

当流式引擎处理一个事件时，它可能需要查找与该事件相关的参考数据。比如，如果该事件代表的是医疗设备产生的数据，很可能就需要查找与病人相关的上下文信息，比如病人所就诊诊所的名字，或者跟病情相关的特定信息。

快速分析和扫描

这个事件和历史数据有怎样的关联呢？对数据做快速分析和扫描或者SQL查询，得到历史的上下文信息，这种能力对应用来说通常是很重要的。

更新

上下文信息是会变化的。例如，来自OLTP（Online Transaction Processing，在线事务处理）应用的上下文信息可能会添加像联系人信息之类的参考数据，而病人的风险评分可能在新的信息被计算出来后发生变化。

当研究这些应用场景时，你肯定注意到了“实时能力”这个主题。你可能会指出，许多组织没有使用Kudu，但是他们也通过Hadoop成功实现这些应用。没错，分开来看，Hadoop的这些存储层可以处理快速插入数据、低延迟随机读、更新和快速扫描。HBase能够处理数据的低延迟随机读/写。那么，快速分析和扫描呢？HDFS可以做得很好。想要简单易用？没问题，使用一个HDFS命令就可以输入你所有的批处理数据文件。问题是，当你想拥有所有这些特性时——逐行插入、低延迟随机读、快速分析和扫描以及更新，这个架构就会变得十分复杂而且难以维护，就像图1-7所示的这样。

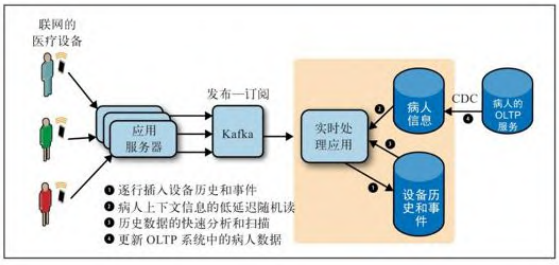

图1-7 实时数据流

同时满足这些存储特性对许多数据库系统来说是可能的，但是使用Hadoop和大数据技术来实现它们却非常困难。现实就是这么残酷，使用Hadoop来搭建这一类应用比较难，因为你选择的存储层只能满足这些特性中的一部分。

硬件环境

Hadoop当初是针对特定的硬件性能和成本考虑而设计的。如果我们回到大约15年前，Hadoop的想法刚诞生时，一台价格合理的服务器会包含几个CPU、8 GB或16 GB的内存（DRAM），只有传统机械硬盘。Hadoop的设计巧妙地让成本和性能之间达到最佳的平衡，将对昂贵的DRAM的使用降到最少，并尽量避免磁盘寻址来最大化I/O吞吐量。摩尔定律的作用确保了内存芯片的容量越来越大。现在廉价的服务器也有数百GB的DRAM内存、数十TB的磁盘容量，以及越来越多的新型非易失性存储硬件。

硬件领域发生了巨大变化。如今普通的服务器越来越可能会配备固态或者NAND存储，带来了新的处理潜力。SSD（Solid-State Drive）硬盘给计算机的性能带来翻天覆地的变化，消除了计算系统中很多传统的性能瓶颈。具体来说，基于NAND的存储器可以处理更多的I/O，吞吐量大，而且与传统硬盘相比，其寻址有更低的延迟。像3D NAND存储这种新趋势正在进一步加剧这种性能变化。一次L1缓存存取大约需要0.5 ns，DRAM内存大约需要200 ns，3D-XPoint（Intel的3D NAND技术）大约需要7000 ns，SSD硬盘大约需要100，000 ns，磁盘查找需要100，000 ns。为了形象地理解这些存储介质之间的性能差异，假设L1缓存存取的时间是1 s，那么一次DRAM读约为6分钟，3D NAND（在本例中为3D XPoint）约为4小时，SSD大约要超过两天，HDD寻址约为7个月。

速度和性能特性对系统设计有深远的影响。想一想20世纪交通工具的速度是如何重塑世界的城市和郊区的。随着汽车的诞生，人们突然能够住在离市中心更远的地方，同时仍然可以依靠汽车使用城市的设施和服务。于是城市向外蔓延，郊区诞生了。然而，由于这些快速行驶的汽车都跑在路上，一个新的问题出现了，开始堵车了！由于这些快速行驶的汽车的出现，我们不得不建造更大规模和更高效的公路。

在Hadoop和Kudu这个领域，存储介质转移到NAND存储的这个趋势显著降低了磁盘寻址的计算成本，这意味着存储系统可以进行更多的寻址操作。如果你使用的是SSD存储，你会期望它能够支撑更灵活的随机读/写的应用。SSD还提高了每秒输入/输出操作数（IOPS）和吞吐量，因此数据可以被更高效地传送给CPU，这又让CPU成为新的潜在性

能瓶颈。当我们展望未来时，这些趋势又被进一步放大，数据平台应该充分利用被改善的I/O性能，还要高效地利用CPU。

如果你买了一台（或者1000台）这种服务器，你肯定希望能够充分利用其优势。实际上，这意味着如果你的服务器有数百GB的内存，你就能扩展堆空间（heap）来服务更多来自内存的数据，并降低延迟。

硬件领域在不断发展，硬件的瓶颈、成本和性能都与15年前设计Hadoop时大不相同。这些趋势会持续下去并且会迅速发生变化。

Kudu在大数据生态中的独特位置

和Hadoop中的其他组件一样，Kudu也被设计成可扩展和可容错的。Kudu仅仅是一个存储层，因此它并不处理数据，而是依赖外部的Hadoop处理引擎，比如MapReduce、Spark或Impala来处理。尽管Kudu能与许多其他Hadoop组件集成，但是它也能作为一个独立的存储引擎而运行，不依赖于HDFS或者ZooKeeper等其他框架。Kudu把数据按照自己的列存储格式存储在底层Linux文件系统中，不像HBase那样，Kudu不以任何方式使用HDFS。

Kudu的数据模型对于任何拥有RDBMS背景的人来说都是熟悉的。尽管Kudu本身不是SQL引擎，但它的数据模型与数据库的相似。表是由固定数量的拥有类型的列组成的，这些列的子集将构成主键。像在RDBMS中一样，主键保证了行的唯一性。Kudu在磁盘上使用列式存储格式，这样能保证对一部分列进行高效的编码和快速扫描。为了实现可扩展性和容错，Kudu的表被水平分割成很多块，称为tablet，其写操作使用Raft一致性算法在tablet之间复制数据。

Kudu的设计源自对当今Hadoop的复杂架构和局限的敏锐理解，以及开发人员和架构师在选择存储引擎时做出的截然不同的决定。Kudu的目标是成为一种各方面条件都适中的选择，既拥有强大而广泛的功能，又对多种负载有较好的性能。具体而言，就是Kudu将低延迟随机访问、逐行插入、更新和快速分析扫描融合到一个存储层中。正如之前讨论过的，HDFS、HBase和Cassandra等存储引擎拥有其中的某一项或几项能力，但它们之中没有一个拥有全部的能力。现有Hadoop存储

引擎之间负载特性的差异被称为分析能力的空白（analytical gap），图1-8对此做了形象的解释。

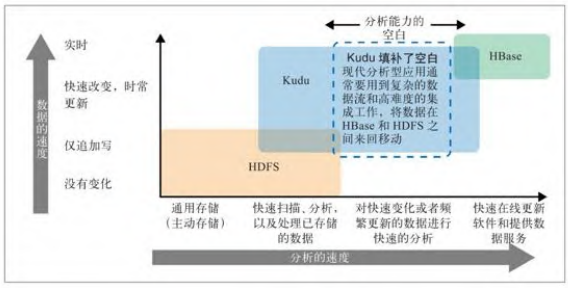

图1-8 Kudu填补了分析能力的空白

HDFS是一个仅支持追加写的文件系统，对于需要顺序扫描大规模数据的处理引擎而言，它的表现最好。在图1-8的另一端是HBase或者其他类big table的系统，比如Cassandra。HBase和Cassandra支持实时随机读/写，以及OLTP应用需要的其他特性。HBase非常适用于那种在线、实时、高并发，而且其中大多数操作都是随机读/写或短扫描的环境。

你会注意到，在图1-8中Kudu并没有声称其对于某一类特定负载要比HBase或者HDFS快。Kudu的目标是，在扫描性能上达到HDFS上的Parquet或者ORCFile格式的两倍；同时，像HBase一样，拥有快速随机读/写的能力，在SSD存储上的读/写延迟只有1 ms。

重新审视前面的实时分析应用场景例子，如果这一次改用Kudu的话，你会注意到我们的架构变得简单多了（见图1-9）。首先，只有一个存储层，不再存在用户发起的compaction或者存储优化过程。对运维人员来说，他们只需要监控和维护一个系统，并且不需要设置定时任务在实时层和批处理层之间或者在Hive的分区之间移动数据。开发人员不需要编写代码来维护数据，或者处理数据延迟到达的特殊情况，并且他们能很轻松地更新数据。而用户则可以即时访问实时和历史数据，因为它们都放在一个地方。

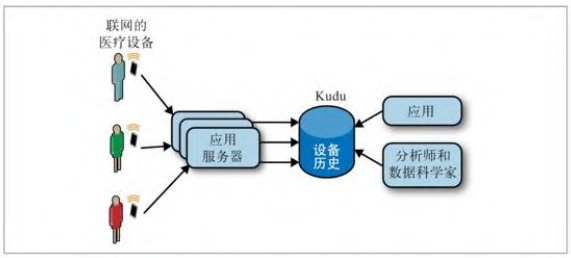

图1-9 使用了Kudu的架构

与其他生态系统的组件对比

Kudu是一个新的存储系统。在过去的10年里，已经有数百个数据库系统被创造出来，人们对此感到疲惫不堪。为什么要重新做一个存储系统呢？在本节，我们会继续通过比较Kudu和传统生态系统的组件来回答这个问题。

最简单的是与SQL引擎的对比，比如Hive、Impala和Spark SQL。Kudu并不会完全执行SQL查询。SQL查询的一部分操作可以下推到存储层，比如投影操作和谓词操作，这些确实是在Kudu中执行的。然而，SQL引擎需要有解析器（parser）、优化器（planer）和执行查询的方法。Kudu仅在执行查询的过程中起作用并向优化器提供一些信息。此外，S Q L引擎必须把投影操作和谓词操作下推到Kudu，或者在此过程中与Kudu通信，所以Kudu参与了查询的执行过程，它不仅仅是一个简单的存储引擎。

现在让我们拿Kudu和传统数据库做比较。为了讨论这个话题，我们需要定义关系型数据库的类型。广义地说，有两种关系型数据库：OLTP（Online Transactional Processing，联机事务处理）和OLAP（Online Analytical Processing，联机分析处理），见图1-10。OLTP数据库用于网站、销售网点系统（Point of Sale，PoS），以及对响应速度和数据完整性要求很高的其他在线服务应用。然而，传统O LT P数据库的大规模扫描的性能并不好，而这种操作在分析类应用中是很常见的。例如，在一个网站上，订单页面通常会显示用户的所有订单，比如所有在线网站都有的“我的订单”这个页面。但是在分析类应用场景中，你通常并不关心某个特定用户的订单，而是关心一个产品的所有销售数据，或者按地区分组的某个供应商的所有销售数据。OLAP数据库能很快地执行这些类型的查询，因为它们针对扫描做了优化。

通常，OLAP数据库在OLTP数据库适用的应用场景中表现得比较差，比如只需选择少量行的操作或者对数据完整性有要求的场景。你可能会感到惊讶：任何数据库都会牺牲一定的数据完整性，但是OLAP

数据库通常是批量加载数据的，如果出现任何错误，OLAP数据库可以重新加载整个批次的数据，而不会丢失数据。

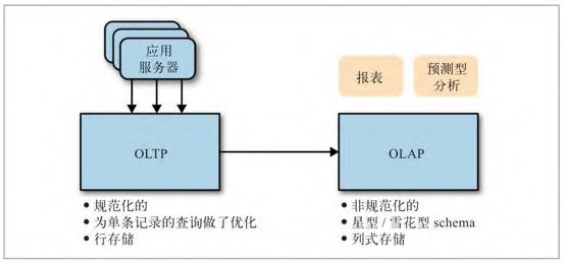

图1-10 OLTP和OLAP

在继续往下讲之前，我们必须了解另一个概念。多行数据可以以行或列的格式存储（参见图1-11和图1-12）。通常，OLTP数据库使用行格式，而OLAP数据库使用列格式。行格式非常适用于全行检索和更新操作。列格式非常适用于扫描所有列中的某几列的场景。典型的OLTP查询是获取完整的行，而典型的OLAP查询仅读取行的一部分。

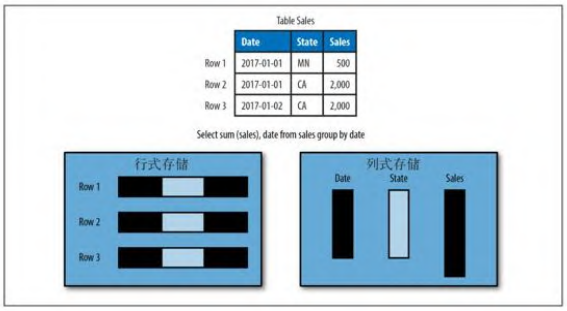

图1-11 行式和列式存储（第1部分）

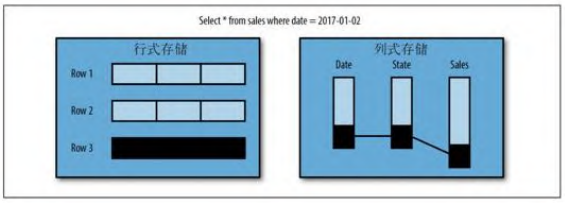

图1-12 行式和列式存储（第2部分）

想象一下在线商店的“我的订单”页面，上面有客户订单的所有相关信息，包括账单和发货地址，以及产品名称、产品代码、数量和价格信息。然而，如果要按地区分组来查询某个产品的销量，需要读取的字段仅包括产品名称、数量、价格和账单状态。

既然我们已经定义了OLTP、OLAP以及行格式和列格式，那么就可以对Kudu和关系型数据库系统做一番比较了。如今，Kudu经常被认为

是支持Hive、Impala和Spark SQL等OLAP SQL查询引擎的列存储引擎。然而，由于Kudu支持根据主键快速检索，并且能够在对表进行分析查询扫描时连续不断地插入数据，所以它同时具有OLTP和OLAP系统的一些属性，因此我们将Kudu归为第三类存储引擎（稍后会讨论）。

在OLTP和OLAP领域中都有很多关系型数据库，从SQLite（OLTP）这样的单进程数据库，到Oracle RAC（OLTP/OLAP）这样的共享所有数据（shared-everything）的数据库，再到MySQL集群（OLTP）和Vertica（OLAP）这样的不共享数据（shared-nothing）的数据库。Kudu与关系型数据库之间有许多共同点。例如，Kudu表有一个唯一的主键，HBase和Cassandra则不是这样。但是，关系型数据库中常见的特性，比如事务、外键和非主键的索引等，在Kudu中是不支持的。这些特性在Kudu中都是可能实现的，而且也已经出现在其路线图上，只是现在还没有实现。

Kudu和Impala的组合与Vertica最相似。Vertica是一个基于C-Store的Postgres的变种，Kudu也继承了C-Store的一些思想。然而，它们之间也有很大差别。首先，因为Hadoop生态不像传统数据库那样是垂直集成在一起的，我们可以让多个查询引擎支持同一个存储系统。其次，由于Kudu实现了基于法定人数（quorum-based）的存储系统，所以它有更强的持久性保证（durability guarantee）。再次，基于Hadoop的查询引擎一般支持本地化调度任务，而不是使用像SAN这样的共享存储系统，这种数据局部性（data locality）有利于提高查询的执行效率。数据局部性是指任务可以从本地磁盘读取数据，与共享存储相比，其出现资源竞争的情况更少。一些不共享数据的数据库，比如Vertica，也能将计算任务调度到离数据更近的地方，但是仅限于Vertica自己的查询。由于这些数据库具有垂直整合的特点，你可以通过添加新的引擎和负载来扩展它们。

Kudu现在拥有一些OLAP和OLTP的特性，但是缺少对跨行的原子性、一致性、隔离性、持久性事务的支持，可以把它归为混合事务/分析处理（Hybrid Transaction/Analytic Processing，HTAP）类型的数据库。例如，Kudu支持快速主键检索，并且能够在数据持续输入的同时进行分析。OLAP数据库在这种应用场景下性能通常不是很好。另外，与OLAP数据库相比，Kudu的持久性保证和OLTP数据库更接近。从

长远来看，有了类似Google Spanner（一种OLTP系统）的跨数据中心的同步复制后，Kudu的持久性会更好。Kudu的quorum能力可以实现一种名为Fractured Mirrors的机制：一个或两个节点使用行存储，另外的节点使用列存储。这样就可以在行存储的节点上执行OLTP类型的查询，在列存储的节点上执行OLAP查询，混合两种负载。

最后，底层的硬件一直在变化，如果硬件被充分利用，这两种数据库的界线就会变得模糊。比如，使用列数据库支持O LT P负载的一个大问题是，OLTP负载通常会读取一行的大部分字段，这在列存储中就意味着需要做很多次磁盘寻址，不过使用SSD或者可持久的内存基本上可以避免这个问题。

与大数据组件对比——HDFS、HBase和Cassandra

我们了解这些以后，如何将Kudu与其他大数据存储系统（比如HDFS、HBase、Cassandra）做比较呢？让我们先来看看这些系统的优势在哪里。HDFS非常擅长扫描大量的数据，也即它的“全表扫描”非常出色，这是分析类负载中很常见的操作。HBase和Cassandra很擅长随机访问，随机读取或者修改数据。HDFS不擅长随机读，并且严格来说，它并不支持随机写，不过你可以通过合并的方式来模拟随机写，但是这么做成本很高。HBase和Cassandra在大规模扫描方面的性能比HDFS差得多。Kudu的目标是把扫描的性能做到HDFS上的Parquet的两倍以内，而随机读的性能则要接近HBase和Cassandra。实际的性能目标是在SSD上做到随机读/写的延迟在1 ms以内。

我们来逐一详细解释为什么HDFS、HBase、Cassandra的性能特性会是这样子的。HDFS是一个纯粹的分布式文件系统，其就是设计来执行快速的大规模扫描的，毕竟这种设计的原始应用场景就是批量构建We b索引。对于该场景和许多其他场景，你只需要能够高效地扫描整个数据集即可。HDFS对数据分区，并将其分散到大量物理磁盘上，以便这些大扫描可以并行利用多个磁盘。

HBase和Cassandra与Kudu类似，因为它们以行和列的形式存储数据，并提供随机访问数据的能力。然而，当谈到数据在磁盘上的物理存储时，它们的存储方式与Kudu就大不相同了。Kudu比这些系统能更快地扫描数据是有很多原因的，但其中一个主要的原因是，HBase和

Cassandra将数据存储在列簇（column family）中，而不是采用真正的列式存储。最终的结果是双重的：首先，同一列的数据不能在一起编码，而编码能够带来极限的压缩（如稍后讨论的那样，“离得近”的数据存储在一起的话压缩效果更好）；其次，在扫描列簇中的一个列时，必然会物理读取列簇中的其他列。

Kudu做大扫描时性能高的另一个原因是，它不需要在扫描时进行合并（merge）操作。Kudu不保证扫描返回的数据是按准确的主键顺序排列的（这对于大多数分析型应用来说都是可以接受的），因此我们不需要在不同的RowSet（若干行组成的块）之间执行“合并”操作。要在HBase中执行“合并”操作，有两个理由：第一是需要保证返回结果的顺序，第二是允许字段（cell）拥有版本，新版本字段或字段的删除可以覆盖早期版本。我们不需要第一个理由，因为我们不需要保证顺序，也不需要第二个理由，因为我们使用完全不同的增量记录方式。我们没有保存每个字段的不同版本，而是保存基线数据（base data），然后将这个基线时间点向前和向后的增量数据分开，以多个块的形式存放。如果你有若干次更新操作，基线时间点之后的增量数据会被压缩合并成REDO日志，而只有基线时间点之前的UNDO日志会被保存。对于当前时间点的查询，我们可以忽略所有基线时间点之前的增量，这样就只需要读取基线数据。

读取基线数据是非常高效的，因为它们是以高密度的列形式存储的。Kudu表有schema，这意味着不需要为每一个字段都保存列的名字或者值的长度，而且将增量保存在其他地方意味着不需要读取时间戳，这就使Kudu的扫描性能几乎和Parquet一样高。

Kudu比HBase高效的另一个特别的原因是，Kudu避免了很多字段的比较操作，这样就避免了很多C P U分支预测失败。过去的10年里，每一代CPU的流水线长度都在增加，因此分支预测失败的成本也在增加，从这个方面来说，Kudu的CPU利用率更高。

小结

Kudu既不适合所有应用场景，也不会完全取代HDFS等令人尊敬的存储引擎，或者亚马逊S3和微软Azure Data Lake Store这一类新的云

存储引擎。此外，的确有一些应用场景是更适合使用HBase和Cassandra的，Kudu还不能取代它们。然而，市场上有一股强劲的趋势，推动着数据系统向着更大规模、数据分析和机器学习的方向发展，并且还要能实时运营。也就是说，这些系统是生产级别的运营系统，运行机器学习或者数据分析任务，直接将产品或者服务交付给终端用户。Kudu的独特之处就在于同时拥有这种运营和分析的能力。正如我们在本章中阐释的，有很多种方法可用于构建这样的系统，但是如果不用Kudu，此类系统的架构开发和运维可能会很复杂，甚至数据也可能被存储在不同的存储引擎里。

---

[1] 译者注：此处的“compaction”意为合并压缩，并不能按照字面简单译为“压缩”。这个词目前没有特别合适的对应的中文术语，因此在文中直接使用英文“compaction”，未做翻译。

第2章 Kudu简介

Kudu经常可以用一句话来概括：一个能对快速数据进行快速分析的Ha-doop存储引擎。虽然这句话简单易懂，但是到目前为止，要实现这个目标并不容易。

我们可以用现有的大数据技术来做数据分析。也就是说，将数据以高效的列存储格式保存，比如Parquet和ORC这种格式，以便计算引擎能够按顺序快速读取整个HDFS分布式文件系统中的数据。分析型的查询会在部分列上执行大规模的聚合操作，这会被有效地转换为查询（query）请求到这些列的投影，然后对这些列中的值进行数值操作。这样的话，用列存储格式就非常合适。首先，对某一列进行投影，意味着只需要对磁盘中保存该列的页做I/O操作，这是可行的，因为这种格式本身就是按照列来划分的；其次，对于数字类型的数据，可以使用各种编码和打包机制把很多行的数据塞进磁盘的一个页里。这意味着I/O可以非常高效，对一列中的值进行计算时速度也会很快。简而言之，HDFS文件系统与列式文件格式相结合，带来了高性能的I/O和计算能力，使得分析型查询能被快速地处理。

另一方面，我们也可以用现有的大数据技术支持快速数据。看一看HBase、Cassandra和其他NoSQL存储引擎，它们都是从头开始构建的，不管是使用批量加载，还是使用批量put操作，它们都允许以极高的吞吐率输入和存储数据。这些存储引擎允许对有唯一主键的记录进行有效的INSERT（put（））、UPDATE（put（））或DELETE（delete（））操作并极快地执行。这些存储引擎通常对扫描请求的响应时间也很短，但是只有基于主键本身扫描时才有这样的性能。因为只有主键的值是经过排序和被维护的，所以如果对数据集中的其他列扫描，性能会很差。在这些存储引擎中，查找指定主键也是非常高效的，比如查找用户或对象的信息。

表2-1对比了HBase、Parquet和Kudu的不同特性。

表2-1 几个存储引擎的对比

引擎	单行访问	事务	一致性	随机操作	列式	压缩	键编码	数据编码
HBase	是	否	行级别	是	是	是	是	否
Parquet	否	否	分区级别	否	是	是	是	是
Kudu	是	否	可配置[1]	是	是	是	是	是

我们发现在当今的Hadoop生态系统中，可以实现对“快速数据”的“快速分析”，但关键是无法通过单个存储引擎来实现。你可以以“慢数据”为代价来实现“快速分析”，或者以“慢分析”为代价实现“快速数据”。

Kudu最开始的设计目标是成为一个新的存储引擎，既能够实现快速分析，又能处理快速数据，这样就缩短了数据从生成和存储在Hadoop中到能被用于分析的时间间隔，并且简化成一个单一的存储引擎。

Kudu具有列存储特性，在很多典型的文件存储格式中都能找到这种特性，比如HDFS上的Parquet和ORC，Kudu用类似于Parquet的格式存储数据，但不再使用HDFS作为底层文件系统。像HBase这样的NoSQL引擎一样，Kudu为用户提供了一组类似于NoSQL的API，比如put、get、delete和scan。

最后，为了满足大数据系统对发生故障时的高可用和高弹性的要求，Kudu采用了一些架构上的措施来保证高可用性、高弹性、持久性和可扩展性，以便与Hadoop生态系统的其他部分相适应。

Kudu的高层设计

正如我们在第1章中所看到的，Kudu的几个主要设计目标是：

- 允许用户尽可能快地扫描数据，达到在HDFS中扫描原始文件的速度的两倍。
- 允许用户尽可能快地执行随机读/写，响应时间大约为1 ms。
- 以高可用、容错、持久的方式实现以上这些目标。

为了实现这些目标，Kudu提供了一个架构，能够极其快速且可伸缩地对数据进行列式存储和访问，还具有真正的分布式和容错模型。它基于现在已经可用的下一代硬件而设计，利用了最快的固态驱动器（Solid-State Drive，SSD）、Streaming SIMD Extension 4（SSE4）和Advanced Vector Extension （AVX）指令集的特性。

Kudu中的角色

Kudu的开发借鉴了很多大数据领域已有的软件项目。有些概念源自Big-Table（其开源派生项目是HBase）、GFS（其开源派生项目是HDFS）、Cassandra等，有点集众家之所长的感觉，这些概念帮助Kudu达到对快速数据快速分析的目的。

Kudu的核心是一个基于表的存储引擎。它不提供SQL，不存在关系型数据库系统（RDBMS）中的SQL重写和SQL执行计划。它只存储结构化的强类型的表，并提供高效地访问数据和载入数据到这些表的方法。这些表有名称，并且是结构化的，这意味着它们有一组定义好的列，而且这些列有数据类型和列名。

Kudu存储自己的元数据信息。表名、列名、列类型以及你在表中存放的数据，都必须存储在这个平台中。所有的数据，不管是你自己的用户数据还是有关表的数据（元数据），到最后都是数据，都需要存储。在底层，Kudu将数据保存在tablet中。Kudu有两个不同的角色来管理这些类型的数据。

元数据存储在由master服务器管理的tablet中。表中的用户数据则存储在由tablet服务器管理的tablet中。因为Kudu保存自己的元数据，所以它不需要访问Hive的Metastore服务。但是，如果你打算在Kudu之上使用Im-pala，Impala就需要使用这个Hive服务。

master服务器和tablet服务器的类似之处在于，这些服务器上的数据都是以tablet的形式存储的，而这些服务器则管理着这些tablet。每个角色通常至少有三个服务器，tablet中的数据可以在这些服务器之间复制，通过Raft一致性算法，其中一个服务器会被选举为某个tablet的Leader。

图2-1展示了一个对于master服务器和tablet服务器来说相同的概念。虽然这些角色在数据的存储、复制和管理方面的概念是相似的，但它们的用途和所存储的数据类型是不同的。

因此，所有的Kudu服务器都可被分为两种类型：master服务器和tablet服务器。

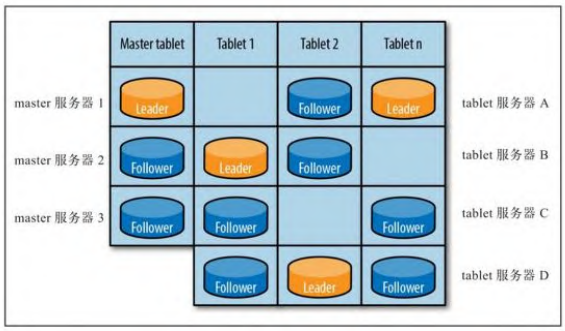

图2-1 Kudu中的角色（Leader为领导者，Follower为跟随者）

master服务器

master服务器的职责是管理Kudu集群的各种操作。客户端与单个master服务器交互来定义表或者获取表的属性或元数据。实际上，master服务器是一个单tablet的表，存储诸如表名、列名、列类型和数据位置之类的元数据，以及诸如状态之类的信息，比如表是否正在被创建、在运行、被删除等。本质上，它管理着系统的“目录”，而这个“目录”也是所有RDBMS都会使用的。

每个表的目录数据的量很小。因此，为了提高性能，系统始终在内存中保存了完整的目录数据的write-through缓存。master服务器负责存储元数据信息，客户端应用程序定位数据的位置时需要用到它们。即使master服务器对于访问数据非常重要，它也无须执行很多操作。因此，你可以在小服务器（硬件）上安装它们。因为master服务器会使用配置中的复制因子（replication factor）来复制元数据存储，所以拥有与复制因子相同数量的master服务器非常重要。默认情况下，应该安装三个服务器。这也将确保Kudu集群的高可用性。

尽管我们建议在生产环境中使用多个master服务器，但我们将使用单个master服务器运行大多数测试。

tablet服务器

我们可以认为tablet服务器是一个工作节点，如果将它与现有的大数据技术比较，它的角色就相当于HDFS DataNode和HBase region server的混合体。tablet服务器的作用是执行所有与数据相关的操作：存储、访问、编码、压缩、compaction和复制。正如你所看到的，与master服务器相比，tablet服务器实际上做了所有繁重的工

作。这正是我们需要可扩展性的地方。tablet服务器还负责将数据复制到其他tablet服务器上。

● K u d u最多可以支持300个服务器。但是，为了获得最高的稳定性，建议tablet服务器不超过100个。

● 建议每个tablet服务器最多包含2000个tablet（包含副本）。

● 建议每个表在每个tablet服务器上最多包含60个tablet（包含副本）。

● 建议每个tablet服务器最多在磁盘上存储8 TB的数据。服务器上所

有磁盘的容量之和可以超过8 TB，并且可以和HDFS共享。但是，我们建议Kudu的数据不要超过8 TB。

● 为了获得对大事实表的最优扫描性能，建议保持一个tablet对应一个CPU核的比例。不要将被复制的表计算在内，应该只考虑Leader tablet的数量。对于小维度的表，可以只分配几个tablet。

这里推荐的所有数值都是基于写本书时的Kudu版本的，欲知最新的数值请参考Kudu的文档（https：//kudu.apache.org/docs/known_issues.html＃_scale）。

存储

Kudu能够实现高性能以及能够优化读/写操作的原因之一在于它存储数据的方式。尽管许多现有的大数据工具使用了特定的格式（例如HBase的HFile），但是Kudu吸取了所有这些不同的应用和格式的精华。如果你一直在用Hadoop生态系统，那么下面列出的许多存储优化方法你都会感到熟悉。

列格式。首先要明白，Kudu以列格式存储所有数据。如果要理解这种格式如何工作以及它对Kudu的性能如何有利，最好的方式是看一个例子。

假设有一个常规文件，比如一个CSV文件，其中一行表示一条记录，它包含所有的列，以任何类型的格式（字符串、数字等）展现，一个接一个，并且由给定字符分隔（见图2-2）。假设这个文件有100行，每行有10列，第一列是id，还有一列是score（分值）。如果想计算整个数据集的平均分，为了执行这个操作，我们需要打开这个文

件，读取第一条记录，抽取score的值，然后读取第二条记录，抽取score的值，以此类推，直到到达文件末尾。最后，为了计算平均值，必须读取所有记录的所有列。所以，我们需要从磁盘中将所有的文件内容读取到系统中并解析它们。

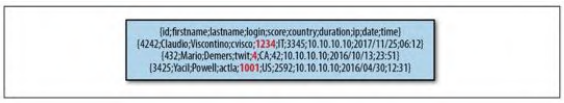

图2-2 平面文件CSV格式

列存储背后的理念不是像在常规文件中那样按行对数据进行分组，而是像其名字的字面意思那样对数据按列分组。因此，列存储就是先存储所有行的第一列的所有数据，然后是所有行的第二列数据，以此类推。同样，所有行的所有score列的内容都将存储在同一区间。如果一个应用想读取所有行的score列，只需要读取列索引，找出相关区间的位置，然后读取所需数据。所有其他的列和区间都会被跳过。如果我们假设所有列的大小完全相同，那么对于求平均值的聚合操作而言，处理列格式文件比处理常规文件将减少大约90%的磁盘传输工作。图2-3解释了这种格式的好处。

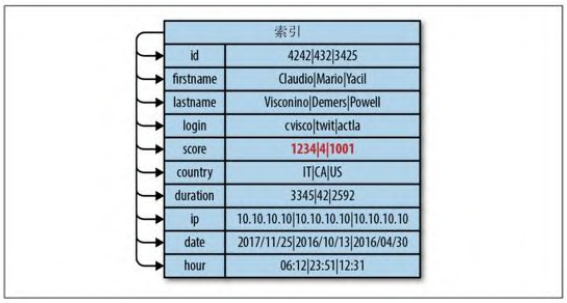

图2-3 列格式

从图2-3中很容易确定，扫描整个列的操作变得非常高效。这样做的缺点是，如果检索整个行将需要访问许多数据块以重新组装这一行，这显然很低效。

因此，列格式的第一个优点是，对于只访问部分列的查询，需要做的I/O操作更少。还有一些额外的跟I/O效率有关的优点，我们先做简单的介绍，在后面的章节再详细介绍。

既然某一列的所有值都存储在一起，我们就可以利用这一点，使用几种方法减少所存储的数据的空间大小。想象一下一家位于北美洲

的公司的客户表，其中有一列是国家。这一列大多数行的值将是“加拿大”、“墨西哥”或“美国”。这个列的值会存储在一起，而不是分散到包含很多不相关的字符串和数字的其他列的值之间。在我们不做其他处理的情况下，与基于行的存储格式相比，对列存储格式进行压缩（LZ4、Snappy或zlib）能够有更显著的压缩效果。

然而，我们还能做得更好。当有了列格式以及上文中提到的列值局部性之后，人们开发出一种新的压缩方式，称作编码（encoding）。表2-2所示的这些编码方式和压缩的作用相同（除了Plain编码），它们将数据转换为更小的表现形式，然后再使用前面提到的通用压缩算法来处理。这些编码通常是基于特定数据类型的。

表2-2 编码类型

编码名称	可用数据类型	描述
Plain	所有类型	使用数据的自然格式存储，比如使用4个字节存储int32型数据，完全不压缩数据的存储空间。当CPU成为瓶颈或者数据已经预压缩过时，这种方式有用
BitShuffle	int8、int16、int32、int64、unixtime_micros、float和double	在自动使用LZ4压缩之前重新排列字节。对于有许多重复值的列或按主键排序时变化较小的值，BitShuffle是一个不错的选择
Run Length	boolean、int8、int16、int32、int64和unixtime_micros	存储重复值的个数与其值本身。例如，对于存储应用工单的表来说，90%的记录都是关闭状态，这种编码可以节省大量空间
Dictionary	string和binary	建立一个唯一值的列表，并存储到这个列表的索引。在前面的例子中，我们只有三个国家，“加拿大”将被编码为1，“墨西哥”将被编码为2，“美国”将被编码为3。然后，对于每个值，Kudu只存储从1到3的数字，而不是存储长字符串

续表

编码名称	可用数据类型	描述
Prefix	string和binary	在连续的列值中，公共前缀可以被压缩。有很多好的例子，比如作为字符串存储的日期字段，其中字符串的开头几乎永远不变（前缀），而字符串的结尾几乎总是变化的（后缀）

文件布局和compaction。在存储系统中进行随机更新的最大挑战之一便是当数据过期后删除这些不需要的数据，并保证操作者访问的是最新版本的数据——这些操作都要尽可能地快。Kudu使用多个机制来实现这一点。

关于Kudu存储数据的方式，让我们来考虑两个主要场景。一个是插入新数据，另一个是更新现有数据。

总的来说，每一个写入操作都将遵循以下步骤：

1.写操作被提交到一个tablet的Write Ahead Log（WAL）。

2.把插入操作添加到MemRowSet中。

3. 当MemRowSet 满了之后，就被刷新（flushed）到磁盘，成为DiskRowSet。

每次Kudu收到新数据时，就将这份数据存储到内存中我们称为Mem-RowSet的地方。你可以把MemRowSet看作一个临时的写缓冲区。当MemRowSet满了之后，就被刷新到磁盘。假设Kudu接收到对名为users的表的插入操作，将主键为“jmspaggi”的行的password列的值设置为“secret”，将last列设置为空，并将其存储到内存中。在刷新时，Kudu会创建DiskRowSet，并且在DiskRowSet中将那一行的所有列分成不同的区间，每一列为一个区间。你可以将DiskRowSet视为较大文件中的一个区间，用于存储特定集合的数据。

如果Kudu稍后接收到对同一张表的另一个插入操作，但这次是针对“mko-vacev”行时，它会首先检查所有现有的DiskRowSet以及MemRowSet，确认该行不在其中（见图2-4）。

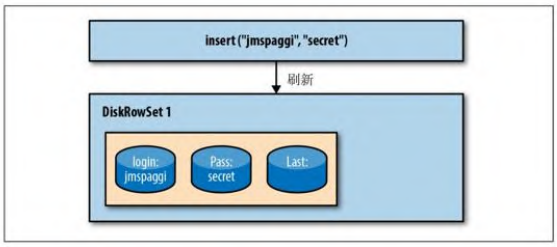

图2-4 对表的首次插入操作

因为它们都不包含这一行，所以这一行会被插入MemRowSet，然后这个MemRowSet过一会儿会被刷新为一个新的DiskRowSet（见图2-5）。Kudu需要执行这种对已有DiskRowSet的验证，以保证主键的唯一性。所有跟特定主键相关的数据都会被保存，并且仅保存在同一个DiskRowSet中。如果你尝试插入一个已经存在的主键，Kudu会返回一个错误（和你尝试更新一个不存在的行时返回错误一样）。

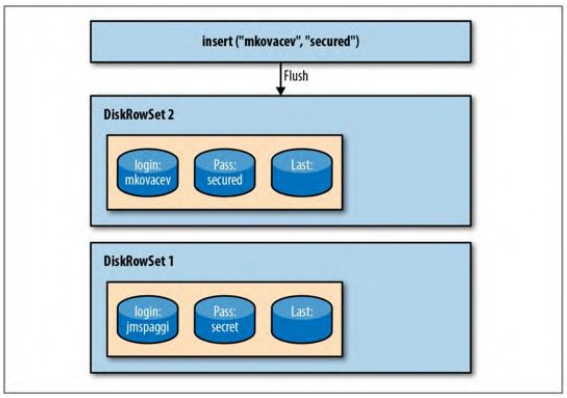

图2-5 向表中插入已经存在的记录

现在，我们考虑对jmspaggi这一行的更新，将password这个字段改为easy （见图2-6）。当Kudu收到这条更新命令时，它首先要确定这一行是保存在当前的MemRowSet中，还是在某个DiskRowSet中。Kudu会寻找第一个包含此行的DiskRowSet，在此过程中会使用布隆过滤器（bloom filter）来减少I/O操作。确认正确的目的DiskRowSet之后，更新后的列会保存在这个DiskRowSet特有的DeltaMemStore中。随着时间的推移，这个DeltaM-emStore可能会变大并且被刷新为DeltaRowSet中的一个RedoDeltaFile。

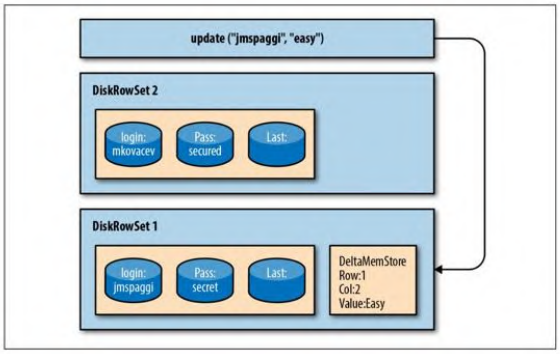

图2-6 对已有行的更新

现在，我们介绍几个重要的文件。首先是被称为基线数据（base data）的块文件，表中的每个列对应一个块，以单个文件的形式存储。然后是Re-doDeltaFile，每次DeltaMemStore做刷新操作都会生成一个RedoDeltaFile，分开保存。当一个客户端发起对某行的一次GET操作时，Kudu需要读取基线数据文件中的数据，然后读取RedoDeltaFile中更新操作的记录并执行这些操作，得到这一行的最新版本。RedoDeltaFile越多，读取操作就越慢。为了避免这种影响，

Kudu会执行名为major REDO delta compaction的操作。为了减少随后对这些文件的读取以提高性能，这个操作会对基线数据文件执行存储在RedoDeltaFile中的更新操作。

除了上文提到的major REDO delta compaction，Kudu还会执行另外两种compaction：minor REDO delta compaction和merging compaction[2]。就像

Kudu在线文档中描述的那样：

minor REDO delta compaction操作不包括基线数据。这种compaction得到的文件本身还是一个增量文件。

major REDO delta compaction操作包含基线数据和任意个REDO增量文件。[3]

虽然读取所有的RedoDeltaFile来执行读取操作是有成本的，但合并和重写所有的数据也是有成本的。为了确保达到最优性能，Kudu不会将所有的RedoDeltaFile合并到现有的数据文件中。事实上，这个数据文件可能会很大，具体大小由表的大小决定。如果只有几行需要更新，重写整个数据文件来合并这些行将会在系统中产生大量的I/O操作，而且最后也只能微小地提高操作的性能。所以，当执行compaction时，Kudu会生成以下3组数据（见图2-7）。

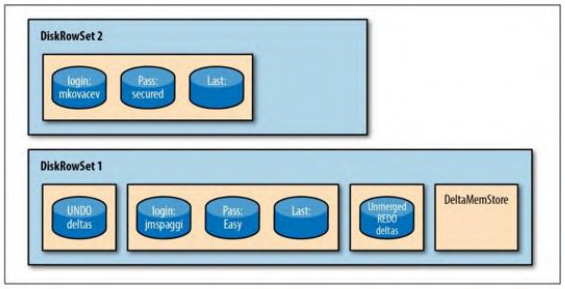

图2-7 执行compaction时生成的3组数据

更新过的基线数据

当某个特定列需要进行大量更新时，Kudu会把基线数据和这些更新合并起来，重新生成一个新的基线数据，通过这种方式来执行更新。

UNDO增量（UNDO deltas）

在对基线数据执行增量修改时，Kudu会跟踪所有的更新操作，并且把它们保存为UNDO增量文件。这个文件被Kudu用来实现获取过去“某个时间点”的值的功能。默认情况下，Kudu将这个文件保存15分钟。因此，可以在Kudu中查询前15分钟以内某个列的确切内容。15分钟后，在下一次compaction迭代中，那些超过15分钟的数据会过期，该文件会被删除。

未合并的REDO增量（unmerged REDO deltas）

正如我们刚才看到的，并不是所有的增量修改都会被执行。当这种修改太小而不能为合并带来好处时，它们会被保留等待进行后续的com-paction。未合并的REDO增量文件包含所有没有被执行到数据文件中的更新。

由于重建一行需要读取的数据和文件更少，对Kudu做完compaction之后，后续的读操作性能得到了提升，对磁盘的访问也减少了。

当Kudu需要重写DiskRowSet中的数据文件时，会确保其中的行按照主键的顺序排列。这样的话，读请求就不需要读取整个文件来查找其要查找的数据。实际上，如果所查找的行在文件中不存在时，一旦读取的主键大于请求的主键，Kudu就可以停止解析文件。

Kudu中的概念与机制

分布式系统的范式是将负载扩散到多个服务器上，允许所有服务器参与读/写操作，因此提高了吞吐量并减少了延迟。在本节中，我们将介绍Kudu如何将数据分散在各服务器上，以及如何配置你的表以帮助Kudu实现可能的最优数据分布。

Kudu中另一个非常重要的概念是主键。Kudu是基于主键来重组和索引数据的，目前Kudu还没有任何二级键或二级索引的概念。而糟糕的主键或分区设计通常会导致热点。

热点

在进一步讨论之前，我们先来讲一讲分布式系统的数据访问中最常见的问题，也就是所谓的热点问题（hotspotting）。当大多数的读或写查询（或者两者）都落到同一个服务器上时，我们就会看到一个热点。我们想要达到的效果是数据完美地分布，使所有的读/写操作分散到集群的大部分节点上。

下面用一个例子来帮助你更好地理解热点问题是什么。我们也将用这个例子来描述如何正确地设计表。假设有一个sensor（传感器）表，在这个表里存储了一大批传感器的所有指标。数据由传感器生成后输入集群，这样我们就能够查询某个特定传感器整个生命周期（趋势检测、故障检测等）的数据。数据流入Kudu，我们必须存储的信息是日期、传感器ID和它的值。因为我们希望能够快速查询指定日期的数据，所以假设将日期配置为sensor表的主键的一部分（主键必须是唯一的，因此，它不能仅仅是日期，除非每个日期只存储一个值）。

这个表会被分成几个tablet（或者分区<partition>），每个分区存储一个日期范围。当数据进入系统时，对于当天的数据，它会到达其日期范围包含当天的那个tablet所在的tablet服务器。因为单个主键存储在单个服务器上，所以集群中不会有其他服务器存储当天的数据。所有的写操作将会落到一个服务器上。尽管这个服务器很可能会过载，其他服务器却都是空闲的。当数据移动到日期范围的末尾时，所有的操作会移动到下一个tablet，而它很可能存储在另一个服务器上，那么这个服务器将会转而成为热点。

这种配置表和设计表的访问模式的方法是非常低效的。图2-8说明了这一点。

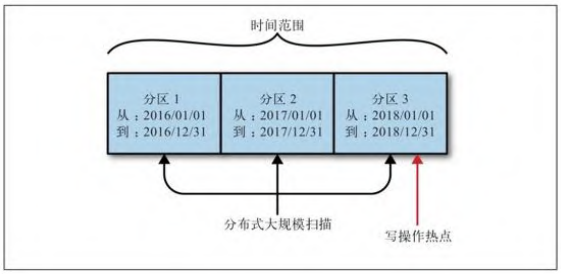

图2-8 基于日期范围的分区

上述问题的镜像问题——写操作被正确地分散到所有服务器，但是读操作集中到一个服务器上——也是类似的。

当考虑你的表时，你需要记住三件非常重要的事情：

- 读操作的访问模式（吞吐率和延迟）。
- 写操作的访问模式（吞吐率和延迟）。
- 存储开销（压缩比）。

分区

从集群中读取数据和写入数据的方式是非常重要的，它们决定了如何设计表的分区，因为当开发一个用例时，最后的结果是最重要的。我们故意把读取数据列在前面。事实上，即使你开发出可能最优的写分区方式，如果你的读操作（在延迟、访问模式等方面）不能满足应用场景的需求，你的项目也可能无法达到其最终目标。

当数据在服务器之间分散存储时，存储开销也会影响你的表。事实上，正如我们在之前的列存储格式的例子中看到的，把相似的数据存储在一起可以提高压缩率。因此，你可以考虑这样做，不过也要看你所面临的限制。

Kudu可以使用两种分区（可以组合使用）：范围分区和哈希分区。

范围分区

范围分区（range partitioning）很简单。请参考图2-8来看日期范围分区的例子。我们将这个表配置为包含三个日期范围，第一个tablet存储2016年的数据，第二个存储2017年的数据，第三个存储2018年的数据。正如我们之前看到的，这样的分区会导致写操作集中到一个热点服务器。然而，对单个传感器ID的数据请求将会并行地发送给所有的tablet服务器，被高效地执行。

根据传感器ID做范围分区将为写操作提供良好的分布。实际上，对于指定的一个日期，所有插入的数据会被发送到所有的服务器上，每个服务器都包含一部分传感器的数据。然而，即使所有的传感器都返回相同数量的指标，也有可能某个ID范围内的传感器个数比另一个

范围内的更多，这将造成分区不平衡并再次影响性能。图2-9说明了这个问题。

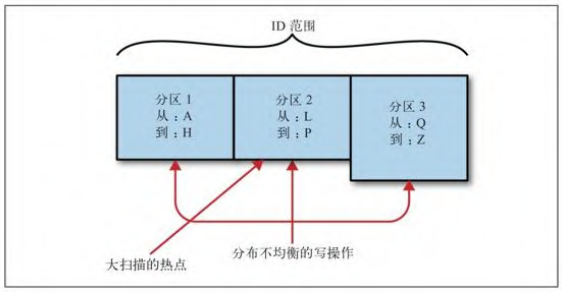

图2-9 根据传感器ID做范围分区

哈希分区

对传感器ID做哈希分区能够帮助解决这个难题。因为所有传感器将提供不同的哈希值，其分布将会是均匀分散在所有服务器上的，并且它们将以相同的速度增长（见图2-10）。然而，读取一个传感器生命周期的所有数值将只扫描一个分区，这样该分区所在的服务器会成为热点，导致性能比较差。

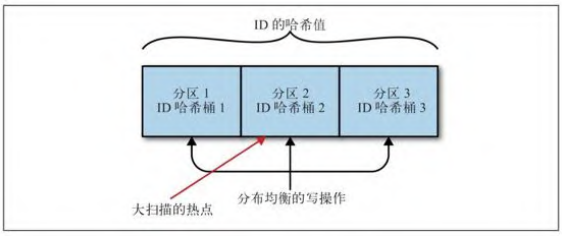

图2-10 根据传感器ID做哈希分区

正如你所看到的，哈希分区和范围分区各自都有优缺点，你需要仔细地考虑。组合使用这两种方式通常是最好的方法，但你还是需要确保在正确的维度使用正确的方法。在我们给出的示例中，我们希望将写操作分散到多个tablet，并且对于读操作也是这样。我们认为哈希分区将传感器ID分散开的效果最好，而范围分区对基于时间的查询效果比较好。将这两者结合起来可以获得良好的写性能（因为有哈希分区，所以写操作将分布在多个分区上）和良好的读性能（因为有分桶，所以大型扫描可以跨多个服务器并行运行）。

我们所说的“桶（bucket）”是哈希值的一个集合。例如，如果你的传感器I D的哈希值在00到99之间，那么一个桶可以包含00到32之间的所有哈希值，另一个桶可以包含33到65之间的所有哈希值，最后

一个桶包含66到99之间的哈希值。这样，所有的传感器将分散到三个桶。如果还有一个分区的维度，根据主键的其他成分，会有多个分区被用来存储特定桶的数据。

图2-11说明了如何使用哈希分区和范围分区来配置这个表。

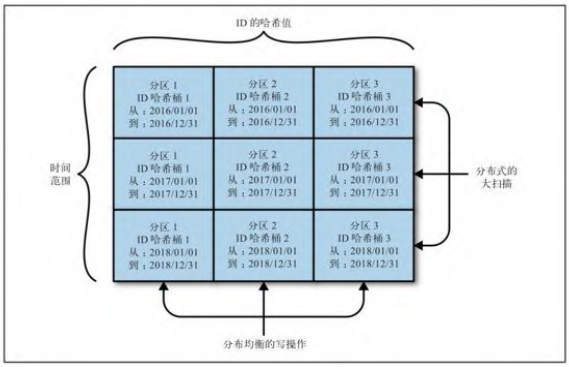

图2-11 根据时间范围和ID的哈希值来分区

请参阅第6章来了解更多细节。

---

[1] https://kudu.apache.org/2017/09/18/kudu-consistency-pt1.html

[2] 译者注：本书没有具体讲merging compaction。

[3] 译者注：上面几段原文省略了一些词语，导致很难理解，容易引起歧义，所以译者补充了一些内容和词语以尽量完整地描述。

第3章 安装与运行

要学习任何工具，最简单的方法就是动手实践，Kudu也是如此。在本章中，我们将介绍安装和配置Kudu时会遇到的各种选项，然后介绍使用RPM包来实际安装Kudu的基本过程。第4章将为你逐一介绍Kudu中的不同角色。

安装

要查看最新的安装选项信息，最好去Kudu的官方网站看一看（http：//kudu.apache.org/）。目前，主要有5种方式来安装和使用Kudu：

- 使用Kudu Quickstart VM（Kudu快速入门虚拟机）。
- 在已有集群上使用Cloudera Manager自动安装。
- 使用软件包手动安装。
- 从源代码构建。
- 使用Cloudera Quickstart VM（Cloudera快速入门虚拟机）。

使用Kudu Quickstart VM

Kudu Quickstart VM是学习Kudu最简单和成本最低的方法（见图3-1）。

使用Quickstart VM的好处是你不需要有一个完整的集群。万一安装时出现问题，也可以轻松地从头开始。我们可以使用Kudu Quickstart VM来熟悉Kudu的API以及与Kudu集成的一些工具和框架，比如Impala。当然，这种方式也有缺点：因为Kudu是在虚拟机上而不是在专用机器集群上运行的，所以它只能用来做与开发和演示相关的事情。

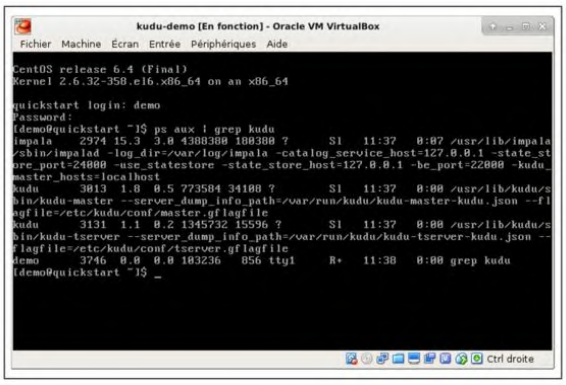

图3-1 Kudu Quickstart VM

Kudu的官网上提供了关于如何使用Kudu Quickstart VM的完整说明。建议你去网站上获取最新版的说明。Kudu Quickstart VM的安装有两个步骤：

1. 下载并运行Oracle VirtualBox。

2. 下载并运行引导脚本（bootstrap script），它会下载Kudu Quickstart VM的镜像并导入VirtualBox。

然后，你将会拥有一个单一的节点虚拟机（VM），上面运行着Kudu和Impala。

当结合本书中的例子使用虚拟机时，请注意，Kudu Quickstart VM并没有预装所有的Hadoop工具。因此，如果你想测试一些使用了Spark、Spark Streaming或Kafka的端到端示例，就需要手动安装Spark、Kafka和ZooKeeper（因为Kafka依赖ZooKeeper）或迁移到其他环境。

使用Cloudera Manager

如果你真的想尝试Kudu的强大功能和可扩展性，或者要将其部署到生产环境中，就需要在集群上部署Kudu。最常见的方法是使用Cloudera Manager，并且要使用Cloudera的Hadoop发行版。Cloudera Manager会自动执行安装前的集群验证、Kudu集群安装、配置以及监控等操作。

相较于使用传统的Linux包管理器或Red Hat包管理器（RPM），大多数Cloudera用户选择使用一种名为parcel的二进制发行版来安装Kudu。parcel是Cloudera用来简化打包和安装其发行版版本的各个组

件的。从CDH 5.10开始，Kudu已经包含在parcel中，并且通过使用“Add Service（添加服务）”选项可以将其简单地添加到集群中。

我们建议使用Cloudera Manager（见图3-2）和parcel在生产环境中管理和安装Kudu，它们称得上是最简单的方法。由于每个Cloudera发行版版本的安装步骤可能略有不同，具体细节请参阅Cloudera的文档（https：//www.cloudera.com/documentation.html）。

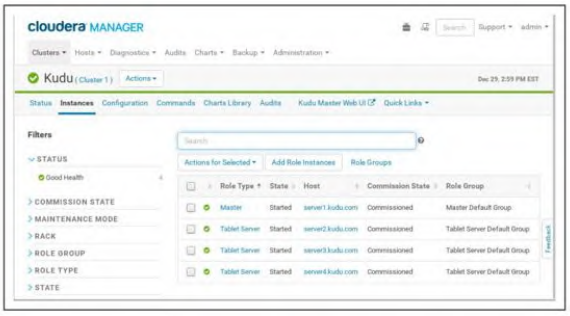

图3-2 Cloudera Manager中的Kudu

Cloudera Manager可以根据最佳实践准则来很好地配置Kudu，但无法为你的部署做规划。你仍然有许多细节需要考虑，包括关于硬件的选择、容量规划、主机和角色的选择（master服务器与tablet服务器），以及Kudu的tablet数据和日志（WAL）的存储位置。我们将在第4章讨论这些内容。

从源代码构建

如果你想开始学习Kudu开发，或希望拥有灵活性，得到最新和最优版本的Kudu上游代码，可以直接从源代码构建和安装Kudu。请查看Kudu的官网以获取详细说明。

从源代码构建使你有机会更接近Kudu源代码。但是，采用这种方式安装时需要更多的步骤，构建应用程序时遇到问题可能更难解决。而从源代码构建的一个好处是它允许你轻松选择要测试的Kudu版本。这种方式还有一个缺点，就是将Kudu与Hadoop生态系统的其他应用程序集成时会更麻烦。

软件包

为了帮助大家快速上手Kudu，我们将介绍使用Kudu软件包安装Kudu。软件包支持大多数主流Linux操作系统，如Red Hat、CentOS、

SLES、Ubuntu或Debian Linux。尽管基于软件包的安装肯定比Cloudera Manager自动安装的工作多一些，但你会发现它其实非常简单。此外，通过这些步骤你将更好地了解Kudu的不同组件。最后，因为软件包安装不需要运行任何其他应用程序（Cloudera Manager、Virtual Box等），所以它需要的资源更少。

Cloudera Quickstart VM

如果你想要在Hadoop生态系统中尝试Kudu，但无法将其部署在真实集群上，一个非常简单又不错的替代方案是在Cloudera Quickstart VM中运行Kudu（见图3-3）。实际上，这个虚拟机安装了整个CDH发行版。除了Kudu之外，它还包括HDFS、Impala、Hive、Spark等。你可以选择要运行的应用程序以及要停止的应用程序。这是一种尝试不同组件集成的好方法，但因为它是一个虚拟机并且它在单个环境中运行所有服务，所以需要大量的内存和CPU资源。因此，其性能并不会反映真实环境的性能。与Kudu Quickstart VM相比，Cloudera Quickstart VM需要更多的时间来启动而且需要更多的空间，但它允许你将所有的测试包含在一个封闭的容器中。

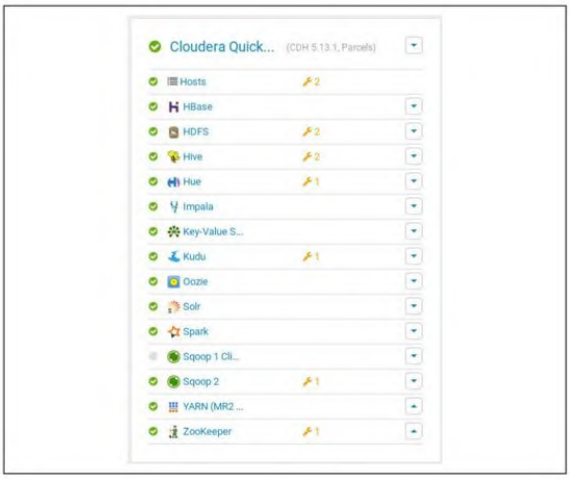

图3-3 Cloudera Quickstart VM中的Kudu

这个虚拟机支持多种环境。它可以在VirtualBox、VMWare、KVM中运行或者作为Docker的镜像。你可以从Cloudera的官网[1]直接下载最新版本。下载后，唯一需要做的就是将其加载到运行环境中并启动它。

快速安装：3分钟或者更短

如果你的手已经放在键盘上，准备开始安装了，那么这一章就是为你而写的。我们的目标是在3分钟内安装并运行Kudu。前提是你已经有一个正在运行的RHEL或者CentOS 7系统，并且你是作为拥有sudo权限的用户登录的。

以下就是安装Kudu的方法：

```
# 安装软件包源
$ sudo cat <<EOD >> kudu.repo
[cloudera-kudu]
# Cloudera 第 5 个发布版中的 Kudu 软件包
# 运行在 Redhat 或者 CentOS 7 x86_64 上
name=Cloudera's Distribution for kudu, Version 5
baseurl=http://archive.cloudera.com/kudu/redhat/7/x86_64/kudu/5/
gpgkey = http://archive.cloudera.com/kudu/redhat/7/x86_64/kudu/ \
  RPM-GPG-KEY-cloudera
gpgcheck = 1
EOD

# 复制 repo 文件到 yum.repos.d 文件夹下
sudo cp kudu.repo /etc/yum.repos.d/

# 安装软件包
sudo yum -y install kudu          # Kudu 基础文件（所有节点都需要）
sudo yum -y install kudu-master   # Kudu master 服务器（master 节点需要）
sudo yum -y install kudu-tserver  # Kudu tablet 服务器（tablet 节点需要）
sudo yum -y install kudu-client0          # Kudu C++ 客户端共享库
sudo yum -y install kudu-client-devel     # Kudu C++ 客户端 SDK

# 启动服务
sudo systemctl start kudu-master
sudo systemctl start kudu-tserver
```

如果你运行的是Debian之类的发行版，请记住Kudu软件包目前仅适用于Jessie（版本8），尚不适用于当前的Debian稳定版本（Stretch）。你还需要执行以下步骤安装存档密钥：

```
sudo wget http://archive.cloudera.com/kudu/debian/jessie/amd64/kudu/cloudera.list \
  -O /etc/apt/sources.list.d/cloudera.list
wget https://archive.cloudera.com/kudu/debian/jessie/amd64/kudu/archive.key \
  -O archive.key
sudo apt-key add archive.key
sudo apt-get update

# 安装包
sudo apt-get install kudu          # Kudu 基础文件（所有节点都需要）
sudo apt-get install kudu-master   # Kudu master 服务器（master 节点需要）
sudo apt-get install kudu-tserver  # Kudu tablet 服务器（tablet 节点需要）
sudo apt-get install libkuduclient0    # Kudu C++ 客户端共享库
sudo apt-get install libkuduclient-dev # Kudu C++ 客户端 SDK

# 开启服务
sudo systemctl start kudu-master
sudo systemctl start kudu-tserver
```

现在，我们已经在一台主机上做完了上述步骤，安装工作就完成了，Kudu已经搭建好并开始运行（见图3-4）。

图3-4 Kudu master服务器的Web用户界面

现在在浏览器中打开以下地址：

Master server:http://＜your-host＞:8051 Tablet server:http://＜your-host＞:8050

就仅安装Kudu而言，这就是全部要做的工作。通过这些指令，你可以将Kudu当作外部应用程序的独立存储管理器。例如，应用服务器可以使用Kudu的Java API作为时间序列应用程序的持久层。

但是，由于Kudu是Hadoop生态系统的一部分（虽然它不依赖于Hadoop生态系统），因此大多数实际应用场景需要的不仅仅是Kudu：为了输入数据，你可能需要 Kafka、StreamSets或Spark Streaming；对于机器学习和数据处理，你可能需要Spark；对于交互式SQL，你肯定想要Impala。实际上，Kudu和Hadoop生态系统紧密地集成正是其优势之一，因此你很少会单独使用它，因为你想要安装的很可能不仅仅是Kudu。

这就引出了一个常见的问题：“如果我只想要Kudu而不要Hadoop生态系统的其他组件怎么办？”在实践中，你很可能也是想拥有Hadoop的。尽管Kudu本身对Hadoop的任何其他组件并没有依赖性，但Kudu几乎总是与Impala一起使用，而这就是事情变得棘手的地方。Impala依赖于Hive，而Hive依赖于HDFS，这就意味着我们要将Kudu与Impala放在一起使用，而且还要配上Hive和HDFS。然而，好消息是Kudu和HDFS很容易和谐共存，甚至可以共享磁盘。但是你需要正确地配置它们，这个话题我们将在第4章中讨论，现在你只需要知道这两个存储层（存储引擎）可以共存，并且由你决定是否仅使用Kudu或HDFS。

小结

在本章中，我们学习了安装Kudu的各种方法，并介绍了基于RPM包的安装。正如本章所讨论的，当Kudu与Hadoop生态系统的其他技术（如Spark和Impala）结合使用时，Kudu的功能得到了增强。因此，为了使用Kudu的全部功能，我们建议使用像Cloudera这样的Hadoop发行版，以及使用像Cloudera Manager这样的管理工具来简化安装和管理。在以后的章节中，我们将深入探讨如何正确规划和配置Kudu，然后介绍Kudu数据管理的基础知识。我们会将Kudu与Impala结合起来使用，后者是Hadoop生态系统的快速分布式SQL查询引擎。最后，我们再来看如何使用Kudu的Java API。

---

[1] https://www.cloudera.com/downloads/quickstart_vms/5-13.html

第4章 Kudu的管理

管理员都有一个梦想，就是他所管理的系统是有弹性、容错和易于扩展的，而且能监控和适应需求的变化。最好这个系统还能自我修复，提前显示警告，让他能安心睡觉，知道系统将以可预测（predicable）的方式运行。

Kudu是从头开始设计和构建的，为的是能够容错、具有可扩展性和弹性，并且为管理员提供查看系统状态的方法，包括可用的API和可视化的Web用户界面（User Interface，UI），以及方便的命令行接口（Command-Line Interface，CLI）。

在本章中，我们假设你是管理员，介绍如何让你快速上手。Kudu的安装方法在前面的章节已经介绍过了，因此我们从为Kudu的部署做规划开始，然后介绍一些最常见和有用的管理员任务。

为Kudu做规划

让我们先花点时间了解如何为Kudu的部署做规划。

Kudu是完全开源的，提供了文档、下载包、概要介绍等，更多的内容可以通过访问http：//kudu.apache.org/来获得。不过，各种Hadoop发行版也可能包含Kudu。如果你使用发行版，请务必参考供应商提供的文档，因为其部署策略可能会有所不同。

Kudu可以作为独占的集群来安装，不依赖任何其他组件，但是我们预计很多情况是Kudu被安装在现有的Hadoop集群上。这样的话，Kudu和Hadoop的其他组件也会有关系，本书也会讲解在这种情况下如何规划、调整和放置Kudu服务。

master服务器和tablet服务器

在介绍硬件规划和Kudu服务的放置策略之前，作为管理员，我们需要了解Kudu里的两个主要组件：master服务器和tablet服务器。这些服务器是管理表的，表又是由tablet组成的，所以也可以说它们是管理tablet的。tablet被复制成多份，所以这些tablet也被称为副本（replica）。

通常，一个集群包含3个master服务器。这些master服务器之间需要达成共识，同意一个服务器作为其领导者（leader）。该决策过程必须由大多数服务器“投票”做出最终决定（例如，3个服务器中的2个，或5个服务器中的3个，依此类推），并且它们做出的决定就是最终决定。该过程使用了所谓的Raft一致性算法（https：//raft.github.io/），既容错又高效。Raft一致性算法是著名的Paxos一致性算法的一种变体。

比较好的规划是使用3个或者5个master服务器，7个master服务器就有点多了。master服务器的数目必须是奇数，最多可以有（n-1）/2个服务器发生故障而程序仍然能提供服务。当多于（n-1）/2个服务器发生故障时，master的服务将不能继续，这样服务对集群来说就不再可用。

表会将其数据分成多个分区来存储，因此就有了tablet这个概念，tablet会进行复制，默认的复制因子为3。因此，每个tablet通常是指包含了表的一部分数据的副本。

master服务器维护用户所创建的所有表的元数据或目录信息。Kudu创建了一个系统目录表以处理在Kudu中创建的各种对象的元数据。此表被设计为只有一个分区，因此也就只有一个tablet。

这些元数据存储在master服务器上的一个表里，这个表只有一个tablet，存储在master服务器上。

假设你决定创建表T，我们使用图4-1中所示的逻辑框图来描述它。该表的数据被分解为若干块，这些块被称为tablet。图4-1中显示了3个tablet：Tablet 1、Tablet 2和Tablet 3。表T还存储了你插入的数据。

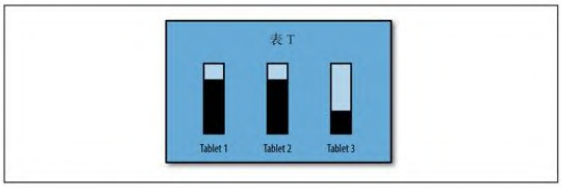

图4-1 表T的逻辑表示

表被分解成tablet后，每个tablet还会在多个tablet服务器上复制，这样的话，表就被进一步分散到各个节点上。在本例中有3个tablet服务器（见图4-2），我们将tablet编号为Tablet 1、Tablet 2、Tablet 3，但由于存在多个副本，因此其中的一个副本是领导者（Leader，用L表示），而另外两个是跟随者（Follower，用F表示）。

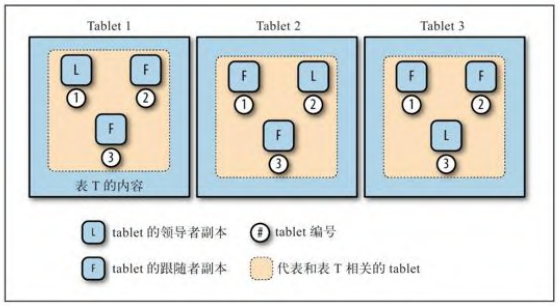

图4-2 复制后的表T

Raft一致性算法用于在tablet服务器上的tablet副本中选择领导者。

master服务器的tablet只包含元数据，所以实际上它不会像存储用户数据的tablet一样有很重的负载，因为我们可以预期你不会像将数据载入tablet那样频繁地执行DDL（Data Definition Language，数据定义语言）操作。同样，这些表也永远不会增长到像用户表那么大，因此master服务器对存储、内存和CPU的要求将比tablet服务器小得多。

系统目录表由entry_type、entry_id和metadata这三列组成。目录管理器将此表加载到内存中，并且构造3个哈希表以允许快速查找目录信息。目录信息是紧凑的，并且由于仅包含元数据而不包含真实的用户数据，它会一直比较小，也不需要占用大量的内存和CPU这些资源。

master服务器仅包含与用户表有关的元数据。因此，其对存储、内存和计算资源的要求比tablet服务器要小得多。具体细节将在本章后面讨论。

为了帮助你更形象地理解刚才介绍的一些高级概念，笔者绘制了图4-3。在本例中，有3个master服务器，每个master服务器管理一个tablet，这个tablet保存了系统目录表的元数据。在这些tablet中，其中一个将被选为领导者，而其他tablet则是为这些数据提供高可用性的追随者。最后，在每个master服务器上都有一个预写日志。

在图4-3的下半部分可以看到，有几个tablet服务器管理你在Kudu中创建的表。我们有N个tablet服务器，表数据（用虚线表示表）存储在tablet中，而这些tablet分散在各个tablet服务器上。我们再次看到其中的一个tablet被选为领导者（用L表示），还可以在其他tablet服务器上找到给定tablet副本的跟随者（用F表示）。图4-3还展示了，对于每个Kudu表，我们都会创建一个单独的预写日志（如图4-3中的WA L所示）并存放在每个tablet服务器上。

实际上，在这个高层架构图中，master服务器和tablet服务器是相同的，表明这两类服务器上存在相同的概念。

那么，应该规划多少个tablet服务器呢？当然，答案是视情况而定。Kudu使用各种编码和压缩策略，以类似于Parquet的列格式来存储数据。由于与Parquet类似，因此从存储容量的角度，你可以粗略地查看数据在HDFS（或其他文件系统中）以Parquet格式存储时占用的空间，并将其用作粗略测算Kudu存储容量的工具。

传统的Hadoop相关技术并不是为了有效地使用固态硬盘（SSD）而从头开始构建的，因为那时候的SSD太贵了。Kudu是一个现代的平台，它能有效利用SSD的性能，还可以使用NVM Express（NVMe，一种非常快速的PCIe闪存适配器）。SSD目前的存储密度仍然不如硬盘驱动器（HDD），这意味着通常在SSD的硬件配置情况下，整体的每节点存储密度将低于HDD。

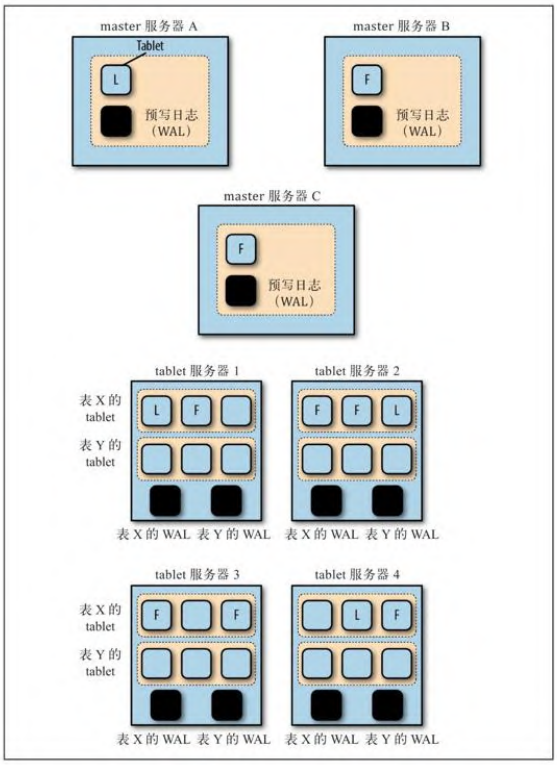

图4-3 高层架构

话虽如此，但较为合理的默认配置是，每个tablet服务器的磁盘容量应该规划为不多于10 TB（包括tablet的所有副本的存储），并且tablet服务器的数量最多为几千个。我们对这些数字的选择是比较保守的，其实随着Kudu的成熟，在一个tablet服务器上完全可以存储更多的数据。从tablet服务器数量的角度来看，以下公式可能有助于你计算服务器的初始数量。

```
d = 120 TB  ; 以 Parquet 格式存储的数据集的大小
k = 8 TB    ; 预计每个 tablet 服务器的最大磁盘容量
p = 25%     ; 预留的额外开销比例
r = 3       ; tablet 复制因子
```

现在从容量的角度看，我们可以进行以下简单的计算，以确定数据集所需的tablet服务器数量：

```
t = (d / (k * (1 - p))) * r
t = (120 / (8 * (1 - 0.25))) * 3
t = 20 个 tablet 服务器
```

此外，还有其他需要考虑的问题，例如规划每个tablet服务器存在多少副本，以确保其数量处于几千的范围内。这些计算取决于分区模式（partitioning schema）策略，以及你的环境中预期会定义的表的数量。我们会在第6章讲这些内容，因为其中会讲到表结构设计。了

解这些之后，确保你的配置满足以上最佳实践标准就只是一个简单的数学问题了。

预写日志

每次对表所做的修改，最后都会导致对tablet和其副本的修改，也会在tablet的预写日志（WAL）中写入一条条目（entry）。Kudu有一整套参数来控制写入WAL的日志段的各个属性，包括日志段的大小以及与预分配的存储相关的一些属性，比如是否异步地预分配存储。日志段的参数还包括其所用的压缩编解码器、日志段应该保存的最小和最大段数，其他副本参与者在落后时可以利用这些保留的日志赶上来。

WA L是一个仅支持追加写的日志，这样初看起来磁盘只需要能高速地执行顺序写入即可。尽管有这些仅支持追加写的日志，master服务器或tablet服务器上的每个tablet还有自己的日志。如果有多个表要写入，从磁盘的角度来看，对WAL的写操作则更像是一个随机写模式。

如果我们从随机写负载的角度来比较HDD与SSD的性能特征——因为我们对大量的低延迟I/O操作感兴趣（而不是高吞吐量的顺序写操作），那么SSD有巨大的优势。HDD的每秒I/O操作数（IOPS）只有几百，而SSD则有数千到数万的IOPS。2016年，三星展示了有100万IOPS的SSD，比HDD快几个数量级！仅仅看写入IOPS的话，也有几十万。很明显，这些都是巨大的差异。SSD的读/写延迟低于100μs，而HDD的延迟在10 ms范围内。你应该明白其意义。

所以，考虑到负载的类型，最好为WAL规划配置快速SSD NVMe的方案。

在容量方面，默认的日志段大小为8 M B，最少要有1个，最多为80个。这个最大值并非不可变的，因为在某些情况下，当其他节点的副本正在重启或已经下线时，leader tablet可能会继续接受写入。

尽管我们要基于这些最坏的情况来做规划，不过Kudu可能会在未来的版本中使用各种技术来改善WAL的磁盘使用率。如果所有的tablet都同时写入，并且tablet个数保持在当前推荐的几千个的范围内，比如每个tablet服务器包含2000个tablet，那么很容易计算出：

8 MB/日志段*最多80个日志段*2000个tablet=1，280，000 MB=约1.3 TB可扩展性和存储密度都在不断提高，而前面的计算是按照绝对最坏的情况来估计的，因此，准备1.3 TB的快速存储器是非常合理的。

在大数据环境中，作为工作节点的典型的2U服务器，其后部有两个S S D供操作系统（Operating System，OS）使用，并且在其前部有12个3.5英寸的LFF（Large Form Factor）或者24个2.5英寸的SFF（Small Form Factor）驱动器托架。可以采用下面几种方法来为WAL准备存储器：

- 在前面的驱动器托架中安装专用的SSD。
- 让后面的两个SSD中的一个专门用于WAL（这样会失去对OS的独立磁盘冗余阵列<Redundant Array of Independent Disks，RAID>的保护）。
- 在这对后置SSD上创建一个物理或逻辑分区，并让一个挂载点专门用于Kudu WAL。
- 安装一个基于NVMe PCIe SSD接口的闪存驱动器。

对于大型生产环境的部署，我们不建议将WAL设置到专用的HDD上（或者更糟，设置到与其他服务共享的HDD上），因为这样会影响写入性能和故障的恢复时间。

各种存储介质的性能差异很大，表4-1做了总结供参考。

表4-1 HDD、SSD和NVMe PCIe 闪存之间的对比

存储介质	IOPS	吞吐率（MB/s）
HDD	55~180	50~180
SSD	3,000~40,000	300~2,000（SAS 最大能达到 2,812 MB/s）
NVMe PCIe 闪存	150,000~1,000,000 以上	最大为 6,400（6.4 GB/s）

上面的这些数字有些差别很大，这是因为有许多因素在起作用，例如块大小、读负载与写负载等。但是，很明显，PCI接口的存储在IOPS和吞吐率方面有极大的优势，这对Kudu的负载非常有利。

数据服务器和存储

当涉及你的数据（也就是我们说的用户数据）时，事情就变得简单了。在服务器上提供尽可能多的可用的HDD（SSD更好！）用来做存

储。你可以随意对服务器进行水平扩展，但请注意，目前数据还不能自动分布到集群新添加的服务器中。

很可能你希望在已经部署了HDFS的大数据集群中启用Kudu。你可以把Kudu的存储目录配置在与HDFS完全相同的挂载点上。比如，如果你有一个已分区和已格式化文件系统的磁盘挂载在/disk1上，你可能会有一个名为/disk1/dfs的目录，用来安装分布式文件系统HDFS的DataNode。那么，你可以继续为Kudu数据创建一个名为/disk1/tserver的目录。

有些人会考虑是否要在服务器上专门为Kudu保留几个磁盘，然后将其他的磁盘用于HDFS。虽然这是可行的，甚至实际上还可以提高性能，但是我们不建议这么做，因为这样会带来管理上的开销，需要处理扩展问题以及检查哪些磁盘真正被有效地使用了。

HDFS当然知道每个磁盘还有多少容量是可用的，并且在2016年它甚至加入了平衡DataNode内部数据块的功能（见“HDFS-1312：Re-balance disks within a DataNode”，https：//issues.apache.org/jira/browse/HDFS-1312），根据轮流的顺序或者根据可用空间来确保对磁盘的均衡使用。

在撰写本书时，我们建议每个tablet服务器不要存储超过8 TB的Kudu数据。这是在考虑副本以及列存储格式的紧凑压缩之后得到的数字。随着Kudu继续被人采用及其自身的成熟，其单节点的密度肯定会提高，不过目前这仍然是一个需要考虑的重点问题，特别是你还担心与其他服务（如HDFS）共享磁盘时。

除了副本数据（即用户数据）之外，tablet服务器还存储了元数据。元数据的写入目录是可配置的，将它们放置在具有高吞吐率和低延迟的性能最高的驱动器（例如SSD）上是非常重要的。在之前的Kudu 1.7版本中，此元数据被写到tablet服务器上数据目录列表中的第一个目录。从Kudu 1.7版本开始，默认写入元数据的目录是由WAL目录指定的目录。但是，有一个新参数--fs_metadata_dir，它允许你控制并指定放置元数据的位置。

将元数据放在WAL或者非数据盘上是有好处的，因为这些数据会随着时间的推移而增长，如果放在数据盘上的话，你的数据盘将随着这些数据的增长而变得不均衡。

复制策略（replication strategy）

为给定的表指定复制因子后，tablet服务器将努力确保该表中的所有tablet采用该复制因子。

因此，如果一个tablet服务器出现故障，副本的数量可能会从三个减少到两个，Kudu将迅速修复这些tablet。

Kudu主要使用的是被称为“3-4-3”的复制策略，即：如果一个tablet服务器出现故障，在剔除该失败的副本之前，Kudu将先添加替换的副本，然后再剔除失败的副本。

另一种被称为“3-2-3”的策略，会先立即剔除失败的副本，然后添加新的副本。然而，对于那种定期下线然后重新上线的系统，这样做会造成节点重新上线后要经过很长一段时间才能重新成为集群的成员。

在可能出现频繁的服务器故障的环境中，复制策略对于系统整体的稳定性很重要。否则你会遇到这样一种情况，即：失败的tablet服务器的所有tablet都会被立即剔除，导致系统花大力气才能让新的副本上线。当tablet服务器短暂停机然后重新加入集群时，这种策略实际上可以实现非常快速的恢复。

在某些情况下，Kudu将执行3-2-3策略，但前提是其中的一个tablet服务器经历的故障是不可恢复的；否则，Kudu将在新的主机上创建新的tablet副本，如果失败的tablet服务器恢复，也不会造成任何损害，这时候新创建的副本将被取消。

部署时的注意事项：是采用新集群还是现有集群

配置Kudu时，通常会有三种不同的场景：

- 一个全新的仅有Kudu的集群
- 一个全新的包含Kudu的Hadoop集群

- 在一个现有的Hadoop集群中添加Kudu

我们将介绍每种场景的注意事项。

全新的仅有Kudu的集群

在开始创建适合Kudu负载的全新集群时，理想的情况是，所购买的服务器的所有数据盘都是SSD，并且用于WAL的驱动器是基于NVMe PCIe SSD接口的闪存驱动器。

实际上，对于仅有Kudu的集群来说，这些并不难做到。考虑到目前推荐的Kudu的整体存储密度大约为10 TB，我们很容易买到典型的1U服务器，其后部有两个SSD用于OS，其他8个SSD用于存储数据，每个大概有1.8 TB的容量，这样每台服务器将有14.4 TB的初始容量。再添加1 TB的NVMe PCIe闪存，你就拥有了一个非常强大的集群，可以满足Kudu的设计目标（见图4-4）。

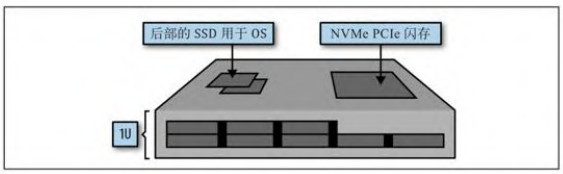

图4-4 仅有Kudu的集群

全新的包含Kudu的Hadoop集群

尽管Kudu可以作为一个独立的集群，但更常见的以及我们期望的情况是，Kudu是一个更大的Hadoop生态系统的一部分，其中包括HDFS以及顶层的其他各种处理服务，如Hive、Impala、HBase、YARN、Solr和Spark。

在这种情况下，Kudu可以无缝集成到Hadoop生态系统的其余部分，特别是对于有大量写入操作的工作负载，我们要确保WAL被放置在可使用的最快的存储介质上。

对于这种环境，我们提出两种不同的方法来做规划：一种是将WAL安装在NVMe PCIe闪存上，另一种是将数据盘中的某一个专门用于WAL。

典型的Kudu部署中有三个master服务器，这与常见的Hadoop环境中的master服务器搭配得很好。

通常在Hadoop中，我们会有一些节点运行“master”类型服务，例如以下服务：

- 用于支持HDFS高可用性的活动/备用NameNode，其中JournalNode分布在三个master服务器上，使用专用的磁盘。
- ZooKeeper分布在三个master服务器上，使用专用的磁盘。
- YARN的活动/备用ResourceManager，以及YARN的Job History Server。
- 一个或者多个HiveServer2实例（虽然它们也可以放在边缘节点上）。
- 一个或者多个HBase master服务器。
- Spark的Job History Server。
- Impala的StateStore和CatalogStore。
- Sentry的活动/备用节点。
- 多个Oozie服务。

可能还有其他服务分布在这三个master服务器上，这取决于你运行的Hadoop的发行版版本以及已部署的服务。

因此，即使在这里，Kudu也需要为WAL准备一个快速驱动器，这与JournalNode和ZooKeeper需要专用驱动器类似。我们希望Kudu也拥有自己的快速驱动器。

同样，我们还是以前面有8个驱动器的典型的1U服务器为例，建议所有这些驱动器都采用SSD。我们为Kudu master服务器的WAL准备一个NVMe PCIe闪存适配器，并为NameNode、JournalNode和ZooKeeper等各种服务提供专用的驱动器（使用RAID1配置的一对驱动器），如图4-5所示。

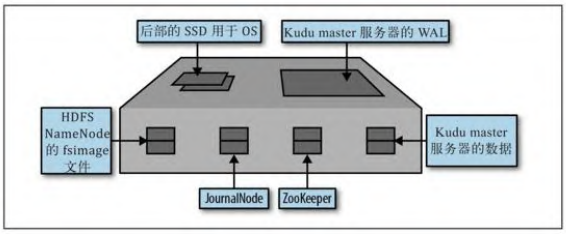

图4-5 包含Kudu的Hadoop集群中的master服务器（选项1）

这里我们还提供第二种选择：使用典型的2U服务器，其后部有两个S S D用于O S，前部有12个3.5英寸驱动器。其中驱动器成对配置为R A I D1，这样我们的操作系统就有了6个块设备。然后，每个设备格式化文件系统并挂载，每个文件系统分别专门用来存储下列数据：

- NameNode fsimage文件
- Hive Metastore数据库（通常是MySQL或者Postgres）
- HDFS JournalNode
- ZooKeeper日志
- Kudu master服务器的数据
- Kudu master服务器的WAL

虽然只有最后的那4个用于Kudu的驱动器会因为使用了SSD而得到更好的性能，但是其余服务也将受益于SSD。在这个例子中，我们没有配置NVMe PCIe闪存设备，以充分利用2U单元中提供的驱动器托架，不过还是为Kudu提供了足够的能力（见图4-6）。

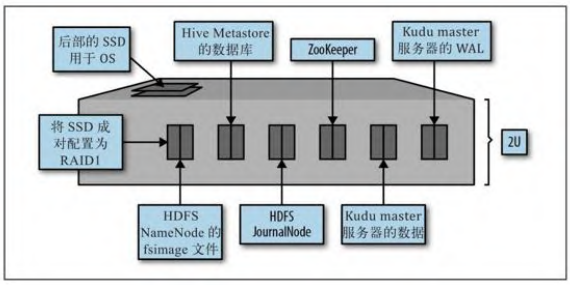

图4-6 包含Kudu的Hadoop集群中的master服务器（选项2）

下一个要考虑的因素是在Hadoop生态系统中规划Kudu的tablet服务器，即在每个HDFS DataNode上部署Kudu数据节点。这些能够水平扩展的服务器也用于Hadoop生态系统中其他各种服务，我们用一个更通用的术语来称呼它们：工作节点（worker node）。

在Hadoop场景下，工作节点通常使用2U的行业标准服务器，前面有12个3.5英寸驱动器托架或者24个2.5英寸驱动器托架，用于存储数据。这些元件的后部通常有两个2.5英寸驱动器托架，用于OS。

理想的策略是在2U服务器中为tablet服务器的WAL准备NVMe PCIe闪存。这些闪存专门供WAL使用，这样能够最大限度地提高Kudu的写性能。然后，在理想情况下，这些服务器单元中的数据驱动器最好都是

SSD，这对Kudu有好处。然而，即使对于新环境，目前采用HDD也很常见，这是符合预期的。因为它们仍然提供了更高的单节点密度，这可能是一项重要的需求，特别是对于在HDFS中存储数据的场景来说。

用户一旦了解到Kudu是独立的，完全独立于HDFS和Hadoop生态系统的其他组件之外，他们通常会有的想法是将Kudu和HDFS的驱动器隔离。虽然这是可行的，并且从存储的角度而言，工作负载肯定是要隔离的，但从长远来看，这种方式可能过于严格了。目前工作负载可能需要更大的HDFS容量；但是，将来可能有更多的负载转移到Kudu。发生故障之后对存储配置进行更改，可能代价高昂而且很烦琐。

我们建议使用相同的驱动器来存储Kudu数据和HDFS数据，但为每个服务提供不同的实际路径。

例如，假设/disk1是位于一个JBOD驱动器之上的ext4文件系统。通常你会为HDFS创建这样一个目录：

/disk1/dfs：HDFS数据目录

我们可以简单地为Kudu创建这样一个目录：

/disk1/tserver：Kudu数据目录

这样，Kudu和HDFS都有各自的目录，而且位于相同的文件系统和设备卷上。HDFS和Kudu都能很容易地知道这个驱动器的剩余总容量，HDFS会重新做均衡（rebalance），比如当这个节点成为Kudu数据的热点时。

有一种情况，我们会建议使用不同的驱动器，那就是需要对静态数据加密的情况。你应该使用HDFS Transparent Encryption（HDFS透明加密）来配置HDFS，而Kudu目前依赖于设备级别的全盘加密技术对静态数据加密。

如果你选择了所有驱动器都是SSD的服务器，那么HDFS和Kudu都将受益。如果所有驱动器都是HDD，HDFS和Kudu也会使用它们，但Kudu的读/写能力可能会更容易受到影响。尽管如此，使用HDD是预料中的事，也很普遍，并且许多工作负载已经在这种环境下运行，所以不必认为这是不可取的。

如今的3.5英寸HDD驱动器的容量在6 TB～8 TB范围内，为Kudu的WA L保留一个或一对这样的驱动器，而不是用它们来保存数据，将导

致大量的存储空间被浪费。因此，我们当然要建议使用NVMe PCIe闪存。这就会产生如图4-7所示的硬件拓扑。

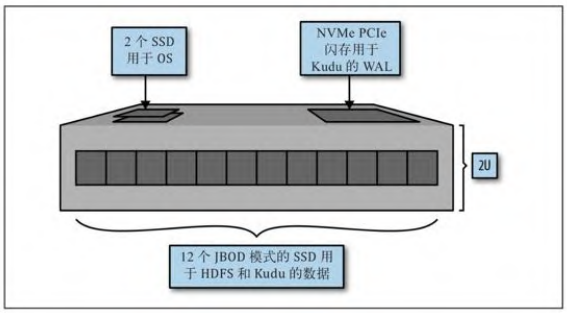

图4-7 包含Kudu的Hadoop集群中的tablet服务器（选项1）

另一种情况是不使用NVMe PCIe闪存的拓扑。在这种情况下，为了确保不为WA L浪费太多空间，我们可以采用带有24个2.5英寸驱动器托架的2U服务器。我们只需要让其中一个驱动器托架专用于Kudu的WAL，这个驱动器最好是SSD（如果其他驱动器不是SSD的话）。

图4-8更好地描述了这种情况。

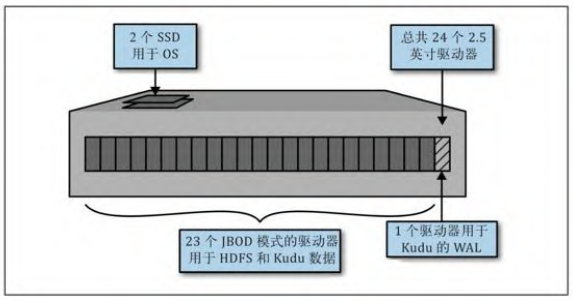

图4-8 包含Kudu的Hadoop集群中的tablet服务器（选项2）

在这种策略下，如果你从集群中删除了Kudu，那么专供WAL使用的驱动器可以快速地重新用于HDFS。所以，它也提供了一些灵活性。

在现有的Hadoop集群中添加Kudu

上一节讨论了在全新的Hadoop环境中Kudu的最佳配置。显然，我们的目标是尽可能接近这些拓扑。

要使Kudu达到最佳配置，最快的办法是购买并安装一个新的NVMe PCIe闪存。这样，你就可以更接近图4-9描述的配置。

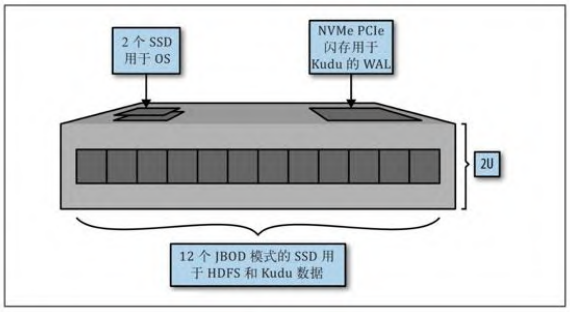

图4-9 为WAL添加NVMe PCIe闪存

新的NVMe PCIe闪存设备专门供WAL使用。同时，Kudu的tablet服务器上的Kudu数据目录将共享分配给HDFS的文件系统挂载点。因此，当有以下文件系统挂载点时：

```
/data1
/data2
...
/data12
```

HDFS 的目录很可能是下面这样的：

```
/data1/dfs
...
/data12/dfs
```

我们基本上只需像下面这样添加 Kudu 目录：

```
/data1/tserver
...
/data12/tserver
```

就可以清楚地分开Kudu和HDFS各自管理的目录，同时最大化磁盘利用率。

图4-10阐释了各种服务是如何占用磁盘空间的。操作系统，甚至是服务本身，都能检测到磁盘还有多少剩余空间。平衡功能和磁盘选择功能将自动使磁盘大致上被均匀地使用，并且最终自然地越来越均匀地被用完。

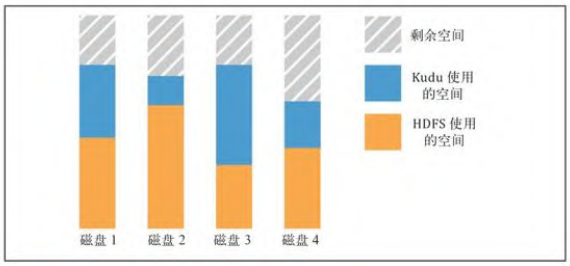

图4-10 推荐的做法——HDFS和Kudu使用相同的驱动器

通常，我们不希望禁止HDFS使用某些驱动器，以便Kudu能拥有自己专用的设备。如果我们这样做，那么文件系统布局将是如下这样

的：

```
/data1/dfs
...
/data6/dfs
/data7/tserver
...
/data12/tserver
```

这使得我们在未来很难调整需求，到最后的某个时刻，某个服务的一些磁盘可能会非常拥挤，而另一些磁盘则可能有很多剩余空间。在这种场景下，要实现得当的平衡非常困难。图4-11展示了磁盘最后可能的样子。

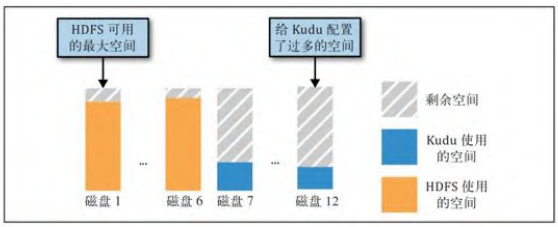

图4-11 不推荐的做法——将HDFS和Kudu服务隔离开

如果无法为WAL添加NVMe PCIe闪存设备，我们可以考虑将WAL放在其中一个现有的数据驱动器上。但是，在这种情况下，我们要确保该驱动器仅由Kudu的WAL使用。因此，建议先将该设备从HDFS使用的设备中删除。这个转换差不多是图4-12所示的那样。

记住，Hadoop中的磁盘不仅用于HDFS，还用作YARN和Impala等服务的临时目录。如果你的环境是这种情况，你需要在所有现有服务中禁用此磁盘，让这个磁盘是干净的且专用于服务器上Kudu的WAL。

当然，当节点是DataNode且服务器上只有12个磁盘时，这种更新（retrofit）会更加痛苦。你在每个工作节点上都会丢失容量，而且通常WAL不会用完磁盘的全部容量。在每个工作节点有24个磁盘的环境中，去掉一个磁盘可能要容易得多，因为它们的容量一般比较小。

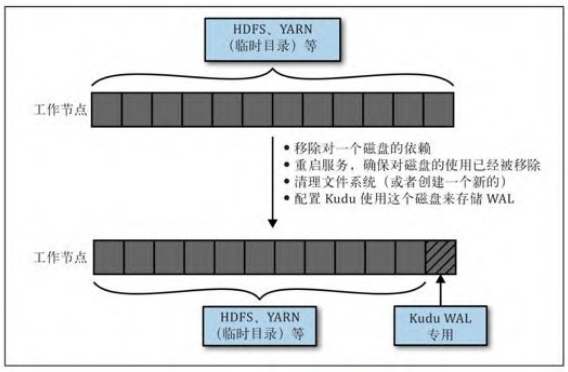

图4-12 对工作节点进行转换，使其也可以用作tablet服务器

通常，人们也会尝试将Kudu添加到现有环境的一部分工作节点中。虽然在技术上可行，但有一点需要注意，工作节点之间可能存在不同的配置（一些节点为HDFS/Kudu的数据提供23个驱动器，为Kudu的WAL提供一个驱动器，而另一些节点为HDFS提供24个驱动器），这样会导致Impala和Spark处理Kudu数据时会更多地从远程读取数据。因此，一般不鼓励这样做。

请记住，此处列出的许多建议，目的都是为了让生产集群能达到峰值性能。在开发环境中，为了尝试Kudu的功能，当然可以让磁盘身兼多职，例如将WAL和数据目录放在同一个驱动器上。

tablet服务器和master服务器的Web UI

对于管理员来说，如果希望高效地工作，那么能够迅速地可视化地查看Kudu部署的配置、有哪些表以及tablet所在的位置，是非常重要的。

Kudu有一个Web UI（下文为了表述上的简洁，将Web UI简称为UI），是查看这些信息时首先会访问的地方。

master服务器UI和tablet服务器UI

master服务器UI和tablet服务器UI具有相同的外观。它们有一些共同的功能，都能够查看以下内容：

Log（日志）

UI上显示了最后一部分日志，以及服务器上日志的存储目录。

Memory（内存）

包括分项列出的内存详细使用情况以及总的内存情况。查看该信息后，能全面地了解哪些地方使用了内存。

Metric（指标）

一个API端点，提供许多JSON格式的指标，很容易用JSON解析器解析它们以获得想要的指标。

RPC(Remote Procedure Call)

以JSON格式列出正在运行的和采样的远程过程调用。

Thread（线程）

一个线程的视图，展示了这些线程占用的CPU以及累积的I/O-wait量。线程是分成若干个线程组的，这样可以轻松地对其进行向下钻取（drill down）操作。

Flag（标志）

在服务器启动时指定的所有标志。这是一个验证你的更改是否生效以及查看默认设置的好方法。

当然，master服务器和tablet服务器有不同的角色。因此，这些UI元素与你所分析的服务器类型有关。

master服务器UI

master服务器UI有以下这些独特的内容：

Master

查看当前当选的领导者以及你的环境中master服务器的列表。

Table

一个所有表的列表。

Tablet Server

所有已经注册的tablet服务器列表、其通用标识符（或者uuid），以及RPC和HTTP地址信息等。

tablet服务器UI

tablet服务器UI有以下这些独特的内容：

Dashboard

提供一个视图，列出当前正在tablet服务器上运行的扫描、事务和维护操作。

Tablet

这个服务器管理的所有tablet（副本）的列表。

Kudu命令行接口

可视化信息对于管理员快速方便地了解他们的环境是很重要的。然而，管理员经常会对一组常见的详细信息和指标产生兴趣，因为他们关注的是对他们来说最重要的东西。因此，熟悉Kudu提供的命令行接口（CLI），对于管理员而言很有意义。这里总结了一些你可以使用CLI执行的关键操作，以便更好地了解集群及其状态，在进行维护和其他操作时做出正确的决策。

要使用CLI，需要以k u d u这个可执行程序的名称开头，后面跟着一条命令来指定要执行哪一组操作。命令可以细分为集群操作、文件系统操作、本地和远程副本操作、元数据文件操作、master服务器和tablet服务器操作以及表和WAL操作，还有检查数据本身的健康和完整性的操作。

下面我们依次介绍每一组操作，通过例子来了解从这些操作中能获得或者监控哪些信息。要想深入了解这些命令，请参考Kudu文档中的详细介绍，因为这些命令并不是一成不变的。

集群

在集群级别，你真正能用的唯一命令是执行健康检查：

kudu cluster ksck ＜comma-separate-master-server-addresses＞

其中的comma-separate-master-server-addresses可能会是这样：

master-server1,master-server2

注意，这里没有指定端口号。如果使用Kudu附带的默认端口，通常不需要为这些类型的命令指定端口号。如果更改了master服务器运行的端口，则需要在这些命令中指定端口号。

这些信息返回所有表的健康状态，以及从master服务器知道的所有tablet服务器收集来的信息。尤其是提供了以下信息：

- 集群健康
- 数据完整性
- 副本不足的tablet
- 联系不上的tablet服务器
- 没有领导者的tablet

你可能希望校验表的数据，以确保写入磁盘的数据符合预期。可以使用以下简单的命令来实现：

kudu cluster ksck ip-172-31-48-12,ip-172-31-59-149-checksum_scan-tables=python-example

对表进行校验是一个很好的做法，尤其是在执行维护工作时或数据中心出现意外而中断之后。它有助于你验证表及其中的数据是否被损坏。还有一些选项可以使用，比如限制只针对特定的tablet服务器扫描、提高校验的并发数等。

快照校验（snapshot checksum，默认情况下是开启的）会在你运行此命令时对数据进行快照。这项功能对于频繁变化的表特别有用，它允许这些任务1能够并行执行。

文件系统

此工具提供对Kudu文件系统的检查、格式化等功能。Kudu中的“文件系统”的概念实际上是一个虚拟概念，它位于磁盘定义的操作系统的文件系统（如ext4、xfs等）之上。探索、格式化和检查文件系统，有助于我们更好地理解Kudu本身。

.gflagfile文件包含了Kudu服务的启动选项，有时它可以帮助你深入了解Kudu进程是如何真正启动的，尤其是当你设置了某些参数但它们实际上似乎没有任何效果时。在此文件中，你可以查看你的参数在启动时是否真的做了设置。

check（检查）

检查文件系统是否存在错误，需要以root身份运行。你必须提供WAL目录，然后提供以逗号分隔的数据目录的列表。只要查看你的服务器上使用的.gflagfile文件，就能知道这些目录的位置。

可以通过ps命令找到.gflagfile的位置：

```
ps -ef | grep kudu
# 可以看到以下输出
kudu 10148 1 0 Aug15 ? 00:04:32
/usr/lib/kudu/sbin/kudu-tserver
--server_dump_info_path=/var/run/kudu/kudu-tserver-kudu.json
--flagfile=/etc/kudu/conf/tserver.gflagfile
```

注意 flagfile 这个选项：

```
--flagfile=/etc/kudu/conf/tserver.gflagfile
```

然后，通过 grep 命令找到 *.gflagfile* 文件中的 fs 参数：

```
grep "fs_" /etc/kudu/conf/tserver.gflagfile

    # 可以看到以下输出
    --fs_wal_dir=/var/lib/kudu/tserver
    --fs_data_dirs=/var/lib/kudu/tserver
```

在这里我们可以看到，WAL目录和数据目录是相同的。Kudu带有文件系统“检查”工具，它将提供一份完整的块管理器报告，说明Kudu文件系统的健康状况。如果发现有丢失的块或孤立的块之类的情况，这些都是警告信号，表明这个文件系统可能有点问题。

当我们对WAL目录执行fs check命令时，它（在本例中）也会自动查找数据目录并为你执行检查：

```
sudo kudu fs check-fs_wal_dir=/var/lib/kudu/tserver/wals
```

上面的命令的输出看起来会像这样：

```
uuid: "e60fc0618b824f6a994748c053f9f4c2"
format_stamp: "Formatted at 2017-08-16 04:36:00 on ip-172-31-59-149.
ec2.internal"
Block manager report
--------------------
1 data directories: /var/lib/kudu/tserver/data
Total live blocks: 0
Total live bytes: 0
Total live bytes (after alignment): 0
Total number of LBM containers: 0 (0 full)
Total missing blocks: 0
Total orphaned blocks: 0 (0 repaired)
Total orphaned block bytes: 0 (0 repaired)
Total full LBM containers with extra space: 0 (0 repaired)
Total full LBM container extra space in bytes: 0 (0 repaired)
Total incomplete LBM containers: 0 (0 repaired)
Total LBM partial records: 0 (0 repaired)
```

现在，对WAL和数据目录都进行检查：

sudo kudu fs check-fs_wal_dir=/var/lib/kudu/tserver-fs_data_dirs=/var/lib/kudu/tserver

在这个简单的例子中，我们实际上得到了和之前相同的输出。

format（格式化）

这条命令会准备好（格式化）一个新的Kudu文件系统。

请记住，这个格式化是针对Kudu文件系统的，而不是操作系统（OS）的文件系统。Kudu的文件系统是一组位于OS文件系统之上的用户文件和目录，这意味着你需要事先准备一个目录或挂载点，并使用支持的OS文件系统，比如ext3。在我们这个例子中，/这个挂载点已经有了一个基本的已格式化的xfs文件系统。

我们来看目前/这个文件系统是如何挂载的：

```
$ mount | grep -E '^/dev'
/dev/xvda2 on / type xfs (rw,relatime,seclabel,attr2,inode64,noquota)
```

可以看到挂载在/目录上的/dev/xvda2这个设备被格式化为xfs文件系统。我们还可以查看/etc/fstab文件来找到所有定义好的挂载点。

为了达到我们的目的，接下来为新的Kudu WAL和数据目录准备目录，我们将使用Kudu格式化命令进行格式化：

```
# 创建 WAL 目录，将 owner 设置为 kudu 用户
$ sudo mkdir /kudu-wal
$ sudo chown kudu:kudu /kudu-wal
# 创建 8 个数据目录
$ for i in `seq 1 8`; do sudo mkdir /kudu-data$i;sudo chown
kudu:kudu /kudu-data$i; done
```

在这个简单的示例中，在同一个底层磁盘上创建8个数据目录没有任何实际的好处。而将这些Kudu目录（包括WAL目录）中的每一个都写在它们各自磁盘的挂载点上，这么做才有意义。

现在，让我们格式化这些目录以供Kudu使用：

```
sudo kudu fs format -fs_wal_dir=/kudu-wal
-fs_data_dirs=/kudu-data1,/kudu-data2,/kudu-data3,/kudu-data4,/kudu-
data5,/kudu-data6,/kudu-data7,/kudu-data8
```

你将看到以下几行输出内容：

```
I0821 22:33:10.068599 3199 env_posix.cc:1455] Raising process file
  limit from 1024 to 4096
I0821 22:33:10.068696 3199 file_cache.cc:463] Constructed file cache
  lbm with capacity 1638
I0821 22:33:10.078719 3199 fs_manager.cc:377] Generated new instance
  metadata in path /kudu-data1/instance:
uuid: "a7bc320ed46b47719da6c3b0073c74cc"
format_stamp: "Formatted at 2017-08-22 02:33:10 on ip-172-31-48-12.
  ec2.internal"
I0821 22:33:10.083366 3199 fs_manager.cc:377] Generated new instance
  metadata in path /kudu-data2/instance:
uuid: "a7bc320ed46b47719da6c3b0073c74cc"
...
I0821 22:33:10.141870 3199 fs_manager.cc:377] Generated new instance
  metadata in path /kudu-wal/instance:
uuid: "a7bc320ed46b47719da6c3b0073c74cc"
format_stamp: "Formatted at 2017-08-22 02:33:10 on ip-172-31-48-12.
  ec2.internal"
```

请注意，这个文件系统是根据给定的uuid注册的。在这个例子中，uuid是a7bc320ed46b47719da6c3b0073c74cc。你还可以在创建格式化的数据目录时指定此uuid。

dump（转储）

用dump命令可以转储文件系统各个部分的内容。

首先转储文件系统的uuid，我们通过指定WAL和数据目录来得到它：

sudo kudu fs dump uuid-fs_wal_dir=/var/lib/kudu/tserver

这条命令将转储一些信息，最后一行显示了uuid：

....

uuid: " 3505a1efac6b4bdc93a5129bd7cf624e "

format_stamp: " Formatted at 2017-08-15 17:08:52 on ip-172-31-48-12.ec2.internal "

3505a1efac6b4bdc93a5129bd7cf624e

接下来，我们希望获得有关此文件系统的更多信息。我们可以在块级别做转储，并查看更多信息：

$ sudo kudu fs dump tree-fs_wal_dir=/var/lib/kudu/tserver

这条命令的输出为我们提供了以下信息：

```
1 data directories: /var/lib/kudu/tserver/data
Total live blocks: 6
Total live bytes: 8947
Total live bytes (after alignment): 32768
Total number of LBM containers: 5 (0 full)
...
uuid: "3505a1efac6b4bdc93a5129bd7cf624e"
format_stamp: "Formatted at 2017-08-15 17:08:52 on ip-172-31-48-12.
ec2.internal"
File-System Root: /var/lib/kudu/tserver
|-instance
|-wals/
|----4322392e8d3b49538a959be9d37d6dc0/
|-------wal-000000001
|-------index.000000000
|----15346a340a0841798232f2a3a991c35d/
|-------wal-000000001
|-------index.000000000
|----7eea0cba0b854d28bf9c4c7377633373/
        |-------wal-000000001
        |-------index.000000000
        |-tablet-meta/
        |----4322392e8d3b49538a959be9d37d6dc0
        |----15346a340a0841798232f2a3a991c35d
        |----7eea0cba0b854d28bf9c4c7377633373
        |-consensus-meta/
        |----4322392e8d3b49538a959be9d37d6dc0
        |----15346a340a0841798232f2a3a991c35d
        |----7eea0cba0b854d28bf9c4c7377633373
        |-data/
        |----block_manager_instance
        |----35b032f3488e4db390ee3fc3b90249c7.metadata
        |----35b032f3488e4db390ee3fc3b90249c7.data
        |----eb44ed9bbcce479c89908fbe939ccf49.metadata
        |----eb44ed9bbcce479c89908fbe939ccf49.data
        |----9e4d11c4b6ac4e1ea140364d171e5e7b.metadata
        |----9e4d11c4b6ac4e1ea140364d171e5e7b.data
        |----80a281a920fc4c10b79ea7d2b87b8ded.metadata
        |----80a281a920fc4c10b79ea7d2b87b8ded.data
        |----2f5bc79e814f4a6a98b12a79d5994e99.metadata
        |----2f5bc79e814f4a6a98b12a79d5994e99.data
```

你可以在本地或远程的副本上使用以下dump命令找到block_id。列数据被分解为块，每个块都有自己的块ID。在后面有关转储tablet信息的内容中，你可以找到获取块ID的方法，然后就可以使用以下命令以人类可读的格式转储块的内容：

```
$ sudo kudu fs dump cfile 0000000000000007 -fs_wal_dir=/var/lib/kudu/
tserver
Header:

Footer:
data_type: INT64
encoding: BIT_SHUFFLE
num_values: 1
posidx_info {
```

```
  root_block {
    offset: 51
    size: 19
  }
}
validx_info {
  root_block {
   offset: 70
   size: 16
  }
}
compression: NO_COMPRESSION
metadata {
  key: "min_key"
  value: "\200\000\000\000\000\000\000\001"
}
metadata {
  key: "max_key"
  value: "\200\000\000\000\000\000\000\001"
}
is_type_nullable: false
incompatible_features: 0

1
```

作为管理员，这个命令可以用来查看实际的块，显示数据类型、编码策略、压缩设置、最小/最大键等各种详细信息。

tablet副本

你可以分别使用local_replica或remote_replica命令在本地或远程执行对tablet副本的操作。与在上一节末尾讲的观察块本身相比，这些操作可以使你获得稍高层次的视图。这里remote_replica命令很有用，因为你不必登录每一台机器来运行命令。

首先列出我们的服务器上的副本：

```
sudo kudu local_replica list -fs_wal_dir=/var/lib/kudu/tserver
...
4322392e8d3b49538a959be9d37d6dc0
15346a340a0841798232f2a3a991c35d
7eea0cba0b854d28bf9c4c7377633373
```

可以通过转储副本信息的方式来检查某个本地副本。让我们先从转储block_ids本身开始：

```
sudo kudu local_replica dump block_ids 15346a340a0841798232f2a3a991c35d
-fs_wal_dir=/var/lib/kudu/tserver
Rowset 0
Column block for column ID 0 (key[int64 NOT NULL]): 0000000000000007
Column block for column ID 1 (ts_val[unixtime_micros NOT NULL]):
0000000000000008
```

我们还可以进一步转储有关表副本的元数据信息：

```
$ sudo kudu local_replica dump meta 15346a340a0841798232f2a3a991c35d
-fs_wal_dir=/var/lib/kudu/tserver
Partition: HASH (key) PARTITION 1, RANGE (key) PARTITION UNBOUNDED
Table name: python-example Table id: d8f1c3b488c1433294c3daf0af403
8c7
Schema (version=0): Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Superblock:
table_id: "d8f1c3b488c1433294c3daf0af4038c7"
tablet_id: "15346a340a0841798232f2a3a991c35d"
last_durable_mrs_id: 0
rowsets {
  id: 0
  last_durable_dms_id: -1
  columns {
                      block {
                        id: 7
                      }
                      column_id: 0
                    }
                    columns {
                      block {
                        id: 8
                      }
                      column_id: 1
                    }
                    bloom_block {
                      id: 9
                    }
                  }
                  table_name: "python-example"
                  schema {
                    columns {
                      id: 0
                      name: "key"
                      type: INT64
                      is_key: true
                      is_nullable: false
                      encoding: AUTO_ENCODING
                      compression: DEFAULT_COMPRESSION
                      cfile_block_size: 0
                    }
                    columns {
                      id: 1
                      name: "ts_val"
                      type: UNIXTIME_MICROS
                      is_key: false
                      is_nullable: false
                      encoding: AUTO_ENCODING
                      compression: LZ4
                      cfile_block_size: 0
                    }
                  }
```

```
schema_version: 0
tablet_data_state: TABLET_DATA_READY
orphaned_blocks {
  id: 10
}
partition {
  hash_buckets: 1
  partition_key_start: "\000\000\000\001"
  partition_key_end: ""
}
partition_schema {
  hash_bucket_schemas {
    columns {
      id: 0
    }
    num_buckets: 2
    seed: 0
  }
  range_schema {
    columns {
      id: 0
    }
  }
}
```

在这个块的副本元数据的输出中，包含了描述表本身的信息的前导内容，例如其名称和模式，接着是RowSet（行集）信息，具体到每一列的级别，然后是表模式的详细描述，也具体到每一列。转储信息最后的那部分显示了这个tablet的状态，以及此副本使用的分区模式的详细信息。

接下来，让我们更深入一点，查看行集本身：

```
$ sudo kudu local_replica dump rowset 15346a340a0841798232f2a3a991c35d
-fs_wal_dir=/var/lib/kudu/tserver
Dumping rowset 0
------------------------------------------------------------------------
  RowSet metadata: id: 0
last_durable_dms_id: -1
columns {
  block {
    id: 7
  }
  column_id: 0
}
columns {
  block {
    id: 8
  }
  column_id: 1
}
bloom_block {
  id: 9
}
 Dumping column block 0000000000000007 for column id 0( key[int64
                                                          NOT NULL]):
 ------------------------------------------------------------------------
 CFile Header:

 Dumping column block 0000000000000008 for column id 1( ts_val[unixtime_micros
                                                      NOT NULL]):
 ------------------------------------------------------------------------
 CFile Header:
```

我们会得到与副本转储相似的信息：一开始是行集中各列的信息，然后是每一列的列块的详细信息。

最后，我们转储出这个副本的WA L信息，可以看到WA L中的序列号，然后是模式信息、压缩细节、一些发生的操作，以及有关最小和最大复本索引的信息：

```
$ sudo kudu local_replica dump wals 15346a340a0841798232f2a3a991c35d
-fs_wal_dir=/var/lib/kudu/tserver
Header:

 tablet_id: "15346a340a0841798232f2a3a991c35d"
 sequence_number: 1
 schema {
   columns {
     id: 0
     name: "key"
     type: INT64
     is_key: true
     is_nullable: false
     encoding: AUTO_ENCODING
     compression: DEFAULT_COMPRESSION
     cfile_block_size: 0
  }
  columns {
    id: 1
    name: "ts_val"
    type: UNIXTIME_MICROS
    is_key: false
    is_nullable: false
    encoding: AUTO_ENCODING
    compression: LZ4
    cfile_block_size: 0
   }
 }
 schema_version: 0
 compression_codec: LZ4
 1.1@6155710131387969536 REPLICATE NO_OP
         id { term: 1 index: 1 } timestamp: 6155710131387969536 op_type:
           NO_OP noop_request { }
 COMMIT 1.1
         op_type: NO_OP commited_op_id { term: 1 index: 1 }
 1.2@6155710131454771200 REPLICATE WRITE_OP
         Tablet: 15346a340a0841798232f2a3a991c35d
         RequestId: client_id: "1ffdc96b901545468e37569a262d3bd1" seq_no:
           1 first_incomplete_seq_no: 0 attempt_no: 0
         Consistency: CLIENT_PROPAGATED
         op 0: INSERT (int64 key=1, unixtime_micros
                       ts_val=2017-08-16T04:48:38.803992Z)

        op 1: MUTATE (int64 key=1) SET ts_val=2017-01-01T00:00:00.000000Z
COMMIT 1.2
        op_type: WRITE_OP commited_op_id { term: 1 index: 2 } result
{ ops
          { skip_on_replay: true mutated_stores { mrs_id: 0 } } ops
          { skip_on_replay: true mutated_stores { mrs_id: 0 } } }
2.3@6157789003386167296 REPLICATE NO_OP
        id { term: 2 index: 3 } timestamp: 6157789003386167296 op_type:
          NO_OP noop_request { }
COMMIT 2.3
        op_type: NO_OP commited_op_id { term: 2 index: 3 }
Footer:
num_entries: 6
min_replicate_index: 1
max_replicate_index: 3
```

接下来，看一看remote_replica命令，我们要做的是将其中一个远程副本复制到此节点上。

首先，我们来看tablet服务器列表，以便可以将其中某个服务器的某个副本移到本地服务器：

```
$ kudu tserver list ip-172-31-48-12
            uuid                 |          rpc-addresses
---------------------------------+---------------------------------
 bb50e454938b4cce8a6d0df3a252cd44 | ip-172-31-54-170.ec2.
internal:7050
 060a83dd48e9422ea996a8712b21cae4 | ip-172-31-56-0.ec2.internal:7050
 3505a1efac6b4bdc93a5129bd7cf624e | ip-172-31-48-12.ec2.internal:7050
 e60fc0618b824f6a994748c053f9f4c2 | ip-172-31-59-149.ec2.internal:7050
```

现在，让我们进行一次远程副本列表操作（见图4-13）。这与local_replica列表操作不同，因为我们不仅会看到tablet ID的列表，还会看到状态、表名、分区、估计的磁盘大小和表模式：

```
  $ sudo kudu remote_replica list ip-172-31-56-0.ec2.internal:7050

Tablet id: 7eea0cba0b854d28bf9c4c7377633373
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 3, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 310B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Tablet id: ca921612aeac4516886f36bab0415749
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 1, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 341B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Tablet id: 4e2dc9f0cbde4597a7dca2fd1a05f995
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 2, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 0B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Tablet id: 1807d376c9a044daac9b574e5243145b
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 0, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 310B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
```

为了使下面的几张图显得更简洁，我们用tablet ID的最后三个字符来标记上面的代码示例中出现的tablet ID，这样更容易识别所描述的tablet。

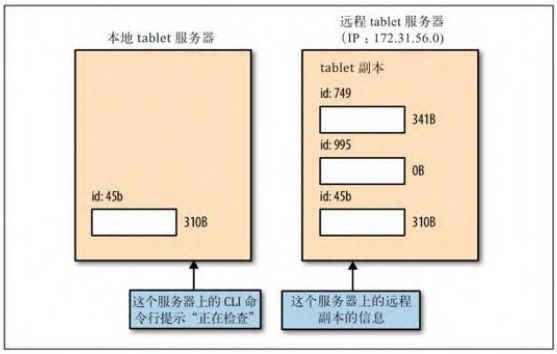

图4-13 列出远程副本

把远程副本复制到本地服务器

要从远程服务器中复制副本，你需要先停止本地正在运行的服务器。然后，copy命令会连接到远程正在运行的服务器，并在本地制作该副本的拷贝。

登录到本地服务器后，使用以下命令停止服务器：

```
sudo systemctl stop kudu-tserver
```

假设我们想要复制IP为172.31.56.0上的tablet ID为1807d376c9a044daac9b574e 5243145b的副本，而在本地已经有这个副本，让我们看看这样做的话会发生什么。

```
$ su - kudu
$ kudu local_replica copy_from_remote 1807d376c9a044daac9b574e5243145b
ip-172-31-56-0.ec2.internal:7050 -fs_wal_dir=/var/lib/kudu/tserver

I0821 23:43:38.586378 3078 tablet_copy_client.cc:166] T
1807d376c9a044daac9b574e5243145b P 3505a1efac6b4bdc93a5129bd7cf624e:
Tablet Copy client: Beginning tablet copy session from remote peer
at address
  ip-172-31-56-0.ec2.internal:7050
Already present: Tablet already exists: 1807d376c9a044daac9b574e5243145b
```

你必须以Kudu用户的身份运行local_replica的copy_from_remote命令，以便拥有合适的权限。不能以root或者sudo用户的身份做这个操作。

错误消息说得很清楚，如果你在远程服务器上有给定tablet的副本，在本地服务器上也已经存在这个副本，那么Kudu是不允许复制这个副本的，因为本地服务器已经有这个副本了（见图4-14）。

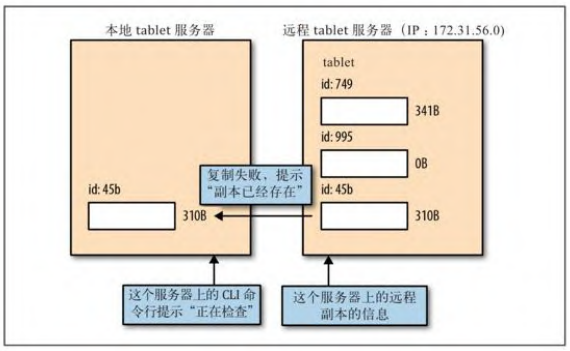

图4-14 复制失败，提示该tablet已经存在

现在，让我们选择一个本地服务器上没有的副本。在我们的例子中，PARTITION 1只能在远程服务器上找到，而在本地没有。它由ID ca921612aeac4516886f36bab0415749表示。我们试着复制它：

```
$ kudu local_replica copy_from_remote ca921612aeac4516886f36b
  ab0415749 ip-172-31-56-0.ec2.internal:7050 -fs_wal_dir=/var/lib/
  kudu/tserver
I0821 23:55:19.530689 3971 tablet_copy_client.cc:166] T
  ca921612aeac4516886f36bab0415749 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet Copy client: Beginning tablet copy session from remote peer at
  address ip-172-31-56-0.ec2.internal:7050
I0821 23:55:19.542532 3971 tablet_copy_client.cc:422] T
  ca921612aeac4516886f36bab0415749 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet Copy client: Starting download of 3 data blocks...
I0821 23:55:19.562101 3971 tablet_copy_client.cc:385] T
  ca921612aeac4516886f36bab0415749 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet Copy client: Starting download of 1 WAL segments...
I0821 23:55:19.566256 3971 tablet_copy_client.cc:292] T
  ca921612aeac4516886f36bab0415749 P 3505a1efac6b4bdc93a5129bd7cf624e:

  Tablet Copy client: Tablet Copy complete. Replacing tablet superblock.
```

我们现在已成功复制了数据。因为之前已经有三个活动状态的副本，所以此时我们有此数据集的4个副本。

虽然这个tablet已被复制，但它还未立即加入Raft共识中。因此，它实际上还没有任何角色，它既不是这个本地副本的追随者也不是领导者。

再看一下我们的表及其tablet的集合。T旁边的第2列是tablet的ID，而行中的条目包含了副本的uuid。

在我们的shell中，我们已经将变量KMASTER设置为Kudu的master服务器列表。

```
[ec2-user@ip-172-31-48-12 ~]$ kudu table list $KMASTER -list_tablets
python-example-repl2
T 1807d376c9a044daac9b574e5243145b
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050)
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050)
T ca921612aeac4516886f36bab0415749
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050)
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050)
T 4e2dc9f0cbde4597a7dca2fd1a05f995
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050)
  P   060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P   bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050)
T 7eea0cba0b854d28bf9c4c7377633373
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050)
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050)
```

禁用ID为ca921612aeac4516886f36bab0415749的tablet的一个副本，该副本的uuid是060a83dd48e9422ea996a8712b21cae4，这样我们就可以启用刚才复制到新系统上的副本了。

```
#
$ kudu tablet change_config remove_replica $KMASTER
 ca921612aeac4516886f36bab0415749 060a83dd48e9422ea996a8712b21cae4

# 现在在 172-31-48-12 服务器上添加副本
kudu tablet change_config change_replica_type $KMASTER
ca921612aeac4516886f36bab0415749 060a83dd48e9422ea996a8712b21cae4
NON-VOTER

kudu tablet change_config add_replica $KMASTER ca921612aeac4516886f36
bab0415749
```

图4-15描绘了刚才执行的操作。我们识别出想要在本地复制的tablet副本。首先停止本地tablet服务器，复制这个tablet，禁用旧副本，激活当前副本，然后重新启动tablet服务器。

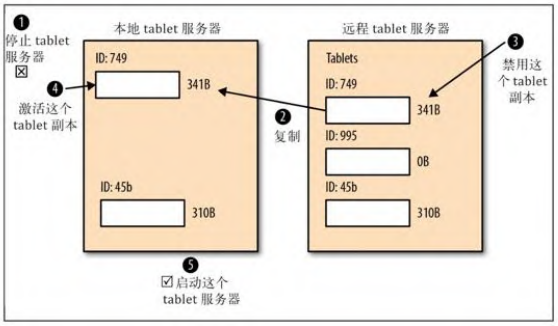

图4-15 把远程副本复制到本地

删除一个副本

使用local_replica delete选项，可以手动删除指定tablet服务器上的副本。但是，你首先需要停止该tablet服务器，再执行删除操作：

```
# 停止 tablet 服务器
sudo service kudu-tserver stop

# 列出 tablet 服务器上的副本
sudo kudu local_replica list --fs_wal_dir=/var/lib/kudu/tserver
4322392e8d3b49538a959be9d37d6dc0
15346a340a0841798232f2a3a991c35d
ca921612aeac4516886f36bab0415749

# 删除你想删除的那个副本
sudo kudu local_replica delete 4322392e8d3b49538a959be9d37d6dc0
--fs_wal_dir=/var/lib/kudu/tserver
uuid: "3505a1efac6b4bdc93a5129bd7cf624e"
format_stamp: "Formatted at 2017-08-15 17:08:52 on ip-172-31-48-12.
ec2.internal"
I0915 17:26:11.767314 2780 ts_tablet_manager.cc:1063]
  T 4322392e8d3b49538a959be9d37d6dc0
  P 3505a1efac6b4bdc93a5129bd7cf624e:
  Deleting tablet data with delete state TABLET_DATA_TOMBSTONED
I0915 17:26:11.773490 2780 ts_tablet_manager.cc:1075]
  T 4322392e8d3b49538a959be9d37d6dc0
  P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet deleted. Last logged OpId: 7.8
I0915 17:26:11.773535 2780 log.cc:965]
  T 4322392e8d3b49538a959be9d37d6dc0
  P 3505a1efac6b4bdc93a5129bd7cf624e: Deleting WAL directory at
 /var/lib/kudu/tserver/wals/4322392e8d3b49538a959be9d37d6dc0
```

请注意，这个tablet的数据实际上被设置为TABLET_DATA_TOMBSTONED状态。我们可以从日志消息中看到WAL已被完全删除，副本的数据也被清理干净，但是默认情况下与Raft一致性相关的数据和tablet元数据信息仍会被保留，这样做是为了确保Raft投票的持久性不受影响。

要完全删除所有的元数据（一般认为这是不安全的操作，因为如果操作不当可能会产生有害的副作用），可以在删除操作中添加--clean_unsafe=true选项，这样所有的副本元数据信息都将被删除。

与Raft一致性相关的元数据

你还可以尝试查看与副本的Raft一致性相关的元数据。

我们可以先打印出副本的uuid，借助这个信息就可以转储某个tablet副本的所有参与者（peer）的uuid，以便深入了解它的Raft配置。

让我们先列出该主机上的所有副本：

```
# 列出这个 tablet 服务器上的副本
sudo kudu local_replica list --fs_wal_dir=/var/lib/kudu/tserver
15346a340a0841798232f2a3a991c35d
ca921612aeac4516886f36bab0415749
```

现在我们可以挑选一个副本，查看它的Raft配置中所有参与者的uuid。

```
# Raft 配置中所有参与者的 uuid
sudo kudu local_replica cmeta print_replica_uuids
ca921612aeac4516886f36bab0415749 --fs_wal_dir=/var/lib/kudu/tserver
e60fc0618b824f6a994748c053f9f4c2 060a83dd48e9422ea996a8712b21cae4
bb50e454938b4cce8a6d0df3a252cd44
```

这样，我们就可以看到3个副本参与者的uuid。在本例中，这个副本实际上是处在墓碑状态（tombstoned）的。因此，我们仍然可以看到该副本的Raft一致性信息还存储在此tablet服务器上。可以使用--clean_unsafe执行删除操作，以便完全清除此信息。

你还可以对cmeta数据使用rewrite_raft_config操作来重写tablet的Raft配置。你可以查看文档来了解详细信息。做这个操作时要谨慎，此类操作应该仅由高级用户在系统可能已停止的情况下执行。

同样，你也可以更改远程tablet服务器上的Raft配置。要更改Raft一致性配置，你需要传入远程tablet服务器、tablet ID和一组参与者的uuid等参数。在出错时，比如丢失了多数副本，你可以使用这种方法进行恢复。但请注意，这样做可能会丢失对数据所做的修改。更多相关的详细信息，请参考Kudu的remote_replica unsafe_change_config操作。

添加和删除tablet服务器

作为一名管理员，往Kudu集群添加更多节点应该是你希望自己能妥善处理的常见操作，你肯定也会希望自己能处理好从集群中删除节点的情况。本节我们将介绍根据需要添加和删除tablet服务器的策略。

添加tablet服务器

要添加新的tablet服务器，通常只需要为服务器安装必要的可执行文件并更改配置，以便这些tablet服务器知道要注册到哪些master服务器上。

以下是安装可执行文件的示例。请按照第3章的内容来设置Kudu软件包仓库。然后，仅安装tablet服务器所需的软件包：

```
sudo yum -y install kudu          # Kudu 基础文件（所有节点）
sudo yum -y install kudu-tserver # Kudu tablet服务器(仅tablet服务器节点)
```

在第3章中，我们没有花心思对配置文件进行任何调整，因为所有内容都安装在localhost上。这一次，我们只安装tablet服务器，该主

机上没有安装master服务器，因此要在这个tablet服务器上设置一些参数。

你可以在/etc/kudu/conf/tserver.gflagfile这个位置找到包含各种标志和选项的主文件。

使用你喜欢的编辑器打开这个文件，然后追加写入下面这一行：

--tserver_master_addrs=ip-172-31-48-12.ec2.internal:7051

然后启动tablet服务器的服务：

sudo systemctl start kudu-tserver

从浏览器中打开tablet服务器的Web UI来查看日志，以便确认该服务正确地连接到了master服务器。你也可以直接在命令行界面中查看日志：

sudo vi/var/log/kudu/kudu-tserver.INFO

你应该会看到类似下面的内容：

sudo grep Register/var/log/kudu/kudu-tserver.INFO

I0816 00:41:36.482281 10669 heartbeater.cc:361] Registering TS with master...

在日志最开始的部分，你会看到你启动服务时指定的参数。

删除tablet服务器

删除tablet服务器，主要是要decommission（解除运行），然后删除这个节点上和Kudu相关的软件包。

要得体地decommission一个tablet服务器，最安全的方式是遵循以下步骤：

1.将这个tablet服务器上所有的副本复制到另一个活动的tablet服务器，并确保所有被复制的副本都加入了Raft一致性协议。

2.删除这个tablet服务器上的所有副本。

3.停止这个tablet服务器。

4.删除tablet服务器的可执行文件

为了确保在执行decommission操作时tablet服务器上没有创建新的tablet，你需要确保在删除时不执行新的DDL操作。

你可以在https：//kudu.apache.org/docs/configuration.html＃_configuring_tablet_servers中找到配置tablet服务器的所有选项。

安全

凡是正式地做部署，无论企业规模如何，都不会事后才会去考虑安全性。安全性一般是通过以下三个方面来考量的：

认证（authentication）

保证试图访问Kudu的人身份的真实性。

授权（authorization）

给指定的人提供访问权限的一种模式，可以访问指定的技术或者数据。

加密（encryption）

对在网络中传输的或在静态存储中的数据加密。

在开始对Kudu做安全配置之前，我们先使用一个简单的类比来讨论其中涉及的一些概念。

一个简单的类比

在日常生活中，进行身份认证的一个简单例子是在通过机场安检时出示护照。边防警卫将护照上的照片与站在他们面前的你进行比对，以此确认你就是护照持有人。如果你通过了这个检查点，意味着你已经通过“认证”，或者换句话说，他们已经验证了你的身份。

在这个机场安检的类比中，当你前往登机口时，登机口的工作人员会确认你确实可以登机。你的登机牌只允许你登上购买了机票的飞机。这是一种授权形式，因为你是经过授权的，并且登机口的工作人员也检查过。

现在，假设你有一封给母亲的信，而且你想保密，只有在你见到母亲时她才能读到这封信。如果你还在家的时候就打乱信件的内容，

即使在去机场的途中有人抢劫你，或者有人设法在你通过海关或登机时拿到你的信，也无法知道或窃取信的内容。

在数据管理领域，“登机”与访问数据库或数据库中的表或文件系统中的文件，代表相同的意思。同时，边防警卫对你进行认证，类似于行业标准的大数据强认证机制Kerberos的工作。最后，你把信件中的内容打乱代表传输中的加密，或者存储时的加密，比如当你把信留在母亲家里时。

我们可以进一步思考这个类比，认识到就像你需要随身携带护照一样，认证是作为客户的你的责任。此外，作为客户，你还需要做出判断，边防警卫确实是这个国家边境巡逻的代表。我们根据他们穿的制服或徽章可以直观地了解这一点。然后，我们可以从先验知识中了解给边防警卫颁发徽章的机构。

因此，从“乘客”或“客户端”的角度来看，我们这一端拥有两个信息：

- 用来证明我们是谁的护照。
- 官方会给边防警卫颁发徽章和制服的先验知识。

从“机场/航空公司”或“服务器端”的角度来看，我们需要做到下面这些：

- 同意将检查护照作为验证你的身份的方式。
- 让边防警卫佩戴合适的徽章，穿着合适的制服。
- 知道你可以乘坐哪一趟航班。

在为Kudu配置安全性时，我们理解这些概念就基本够了。

对于客户端，需要执行以下操作：

- 启用Kerberos，并在访问Kudu时拥有Kerberos Ticket Granting Ticket（TGT，即有“护照”）
- 了解Certificate Authority（CA，证书颁发机构），以验证在Kudu端配置的Transport Layer Security（TLS，传输层安全性）证书（即“知道警卫的徽章是官方颁布的”）。

对于服务器端，需要执行以下操作：

● 启用Kerberos认证（即“同意护照是验证和认证身份的模式”）。

● 拥有由适当的CA签署的TLS证书（即“佩戴了正确的徽章和穿着制服的边防警卫”）。

● 拥有一个授权数据库，保存谁可以访问哪个表这样的信息，通常由类似Sentry的服务完成（即“有关你被允许乘坐哪趟航班的信息，都存储在航空公司的数据库中”）。

有了这些入门知识，我们就可以了解今天Kudu在安全方面的能力以及它在不久的未来的发展方向。

Kudu的安全功能

Kudu支持的安全功能还在不断地增加，截至撰写本章时，它支持以下内容：

● 使用TLS对线路加密。

● 静态数据的加密。

● Kerberos认证。

● 自身的粗粒度授权。

● 通过Impala进行细粒度授权。

● 日志修订（redcation）。

● Web UI安全。

线路加密

需要从两个不同的角度来看待传输线路加密。首先是集群中的tablet服务器和master服务器之间的内部通信。Kudu有一个内置机制，可以为集群中的所有服务器构建和发布内部X.509证书。因此，TLS是对集群内部线路流量进行加密的主要机制。

集群内通信只是这些证书的次要作用。事实上，它们首先被用来提供强大的身份验证功能，以便集群中的每个服务器都可以信任连接到它的服务，无论是tablet服务器还是master服务器，实际上都可以信任。

通常，Kerberos用于身份验证，但在此特定情况下，使用证书策略进行集群内通信可以减少Kerberos Key Distribution Center（KDC，密钥分发中心）的负载，使集群可以非常容易地扩展，无须为集群中的每个服务到KDC进行身份验证；相反，它们依靠证书来验证集群内通信的服务器身份。

使用相同的机制，也可以实现Kudu客户端和服务器之间的加密。因此，启用服务器之间的线上通信加密和认证非常简单，只需要设置这些标志：

```
--rpc_authentication=required
--rpc_encryption=required
```

要设置这些标志很简单：可以在启动master服务器或者tablet服务器时，将其当作命令行参数传入，或者当作flagfile中的配置项写入。这些标志必须在所有节点上的Kudu服务中设置，才能正常生效。

当master服务器或tablet服务器启动时，会设置这些标志。下面的示例说明了如何直接在命令行选项中指定它们，你也可以在--flagfile选项指定的gflagfile文件中指定它们：

```
kudu-master --rpc_authentication=required --rpc_encryption=required
--master_addresses=... --flagfile=gflagfile
```

如果你在Cloudera中运行Kudu，那就很简单了，只需要在Cloudera Manager中为Kudu服务设置enable_security标志，其标签为“Enable Secure Authentication and Encryption（启用安全验证和加密）”，然后服务在启动过程中将自动设置这两个参数。

注意，--rpc_encryption还有其他的设置选项，例如disabled和optional，但我们不建议使用这些选项。optional标志将使Kudu尝试使用加密，如果失败，将仅允许来自可信子网的未加密流量通过。你可以通过Kudu文档来详细了解这些选项，但是一般来说，我们不鼓励使用它们。

静态数据加密

目前Kudu本身还不支持静态数据加密，但可以将数据存储在使用全盘加密工具（如dm-crypt）加密的磁盘上，这是Kudu完全支持的。

这样的话，任何读取或写入这些磁盘的内容都将被即时加密，而且对Kudu应用程序本身是透明的。

Kerberos认证

你可以在Cloudera Manager中选择“启用安全身份验证和加密”标志来启用Kerberos身份验证。不过在此之前，还需要执行一些操作。你需要在Kerberos KDC中创建主体（principal），比如kudu@ACME.COM。注意kudu这个名称目前是固定的，无法自定义。接下来，你需要为此主体创建一个keytab文件，然后在启动选项中引用此文件，如下所示：

```
--keytab_file=<path-to-keytab-file>
```

如果通过Cloudera Manager启用Kerberos，Cloudera Manager将会自动创建主体和keytab文件，并在Kudu服务的命令行启动选项上自动设置它们。

启用此功能后，Kudu服务现在只允许已通过Kerberos认证的客户端连接进来。但请注意，此时所有由Kerberos验证过身份的用户都可以访问集群。在下一节我们会讲解如何限制用户的访问权限。

用户授权

启用Kerberos后，我们可以确保经服务认证的客户端的真实身份，相当于前面的类比中所说的边防警卫检查我们的护照并验证了我们的身份。在撰写本书时，Kudu本身有两组用户：

Superuser（超级用户）

有管理功能，能使用kudu命令行工具诊断或修复问题。

User

访问和修改Kudu集群中的所有表和数据。

当使用Cloudera等发行版管理Kudu集群时，可以将Superuser参数留空。这意味着，实际启动Kudu master服务器和tablet服务器的用户将被允许执行管理功能。在这种情况下，它就是kudu这个用户，特别是这个名字也是Kerberos主体的名字。如果这个用户是由发行版管理的，那么你应保留名字的原样，除非有特别的原因导致用户必须使用

命令行工具修复Kudu集群，比如无法直接使用Cloudera Manager的时候。

接下来，拥有访问权限的用户的列表默认情况下包含所有人，用星号（*）表示，目前这是非常粗粒度的。典型的使用场景可能是外部报表工具发送过来Impala的查询。Kudu客户端的API访问可能会受到防火墙的限制，或者我们可能为ETL作业创建单个功能ID，以便允许在Kudu中创建、加载和操作内容。

因此，我们可能只希望有两个用户，每个用户可以作为一个整体来访问Kudu[2]，包括：

impala

etl_id

这些参数由以下命令行选项设置，我们将mko_adm这个ID和启动Kudu服务的主体ID kudu作为超级用户管理员：

--user_acl=impala,etl_id

--superuser-acl=kudu,mko_adm

在写本节时，这种授权方式肯定是比较粗粒度的。但是，考虑到各种报表和查询类应用主要是通过Impala进行的，所以Impala的所有授权控制都可以使用。Impala集成了用于授权的Sentry，因此它目前可以控制对Impala表、数据库和列的访问。

我们要强调重要的一点，即：Impala表是在Kudu之上的，它们是分离的。这意味着虽然我们被授予了通过Impala访问特定表的权限，但是如果没有控制好通过Kudu客户端API对Kudu的访问，则用户完全能绕过Impala来操纵和访问数据。

日志修订

默认情况下，任何Kudu服务器日志都会被修订，所有的行数据会被删除，以防止敏感数据泄露。谈到日志修订功能，我们还有两个选项：

- 在Web UI中禁用日志修订，但是服务器日志会被修订。
- 完全禁用日志修订。

可以通过以下调整来设置它们：

--redact=log：Web UI不做修订，但是服务器日志仍需修订，删除敏感数据

--redact=none：所有地方都不做修订（不推荐，除非在做调试）

Web UI安全

Web UI安全包括启用TLS以加密从客户端进出Web UI的所有流量的功能。你需要用PrivacyEnhanced Mail（PEM）格式来准备证书，以及私钥、私钥密码命令（运行此指定命令就输出密码）和CA文件。可以使用以下参数进行设置：

```
--webserver-certificate-file=<path-to-cert-pem>
--webserver-private-key-file=<path-to-key-pem>
--webserver-private-key-password-cmd=<password-cmd>
```

虽然Web UI的网络传输已经使用TLS加密，但是Kudu并没有通过Kerberos（通过SPNEGO实现）来控制谁能访问Web UI。

在对Web UI的访问能够被控制之前，你可能会决定先完全禁用Web服务器。可以使用下面这个命令：

--webserver-enabled=false

要注意到，执行此操作后，还会禁用生成Kudu指标的REST API端点，这可能会对你用来监控Kudu的监控系统产生影响。

基本的性能调优

与所有存储系统一样，Kudu有许多需要注意的深入的性能调优策略。Kudu仍然处于起步阶段，但是作为管理员，有一些性能调优的领域你需要理解。

到目前为止，我们已经讨论了很多用于WAL和数据目录的底层存储的种类。在接下来的章节中，我们将详细介绍表的模式设计，这对于你的工作负载获得不错的性能至关重要。

从管理员的角度，我们在此处列出了一些可调的参数，但这个列表不是详尽的。另外，比较重要的一点是，要知道实际上性能瓶颈是在Kudu的服务器端而不是客户端。有时客户端没有足够的资源（C P U或内存）或足够的并行度来驱动Kudu服务器达到性能极限。

从高层次的角度来看，在性能调优时有三个需要关注的方面：

- 分配给Kudu的内存量。
- 使用适当的分区策略（如上所述，这将在以后的章节中讨论）。
- 维护管理器的线程数量。

Kudu的内存限制

Kudu是用C++从头开始构建的，使用了高效的内存管理技术。虽然这意味着精益，但是为Kudu提供集群允许的尽可能多的内存仍然是有意义的。

在多租户集群中——在这种情况下，多租户指的是所有部署在大数据环境中的各种服务，例如HDFS、HBase、Impala、YARN、Spark——你可能需要仔细设计，为每个服务划分适当的内存。

以下是要调整的参数：

```
--memory_limit_hard_bytes
```

在生产环境中，刚开始时你应该考虑将这个值设置在24 GB到32 GB之间。当然，你也可以设置更高的内存。

维护管理器的线程

维护管理器的线程（maintenance manager thread）是执行各种任务的后台线程。它们会做各种工作，比如将数据从内存刷新到磁盘，以此进行内存管理（把记录从行存储格式的内存切换到列存储格式的磁盘中），还有提高整体的读性能或释放磁盘空间。只要有线程可用，Kudu就会将工作安排给这些线程。

基于行的数据很容易写，因为写一行只需要做很少的处理。基于行意味着该行中的数据与另一行中的数据无关。因此，它是一种适合写操作的非常快速的格式。基于列的数据更难快速地写入，因为它需要做更多的处理来将某一行转化成列格式。因此，基于列的数据集在读取时更快。当Kudu将数据保存在内存中等待将其刷新到磁盘时，数

据是以写优化的行格式存储的。但是，在维护管理器的线程将其刷新到磁盘后，就会转换为列格式存储，以便适合聚合读取类型的查询。

通常，我们会根据给定tablet服务器上分配的数据驱动器的数量设置这些线程的数量。如果tablet服务器上有12个驱动器，我们应该设置4个左右的线程，或者设置为机械磁盘数量的1/3。但是，磁盘速度越快，就可以设置越多的线程。从1/3开始，可以放心地将此数字增加到磁盘数量的范围内，也就是1。这只是一般性建议，你还需要在要调优的工作负载上测试这个配置。这里介绍这个参数的主要目的是，不要不管这个参数而默认只用一个线程，也不要为此参数设置数十个线程。

以下就是刚才讨论的那个参数：

--maintenance_manager_num_threads

监控性能

Kudu的Web UI提供了一个JSON指标页面，可帮助你轻松快速地了解表明可能存在性能瓶颈的某些性能指标。

JSON格式的指标与各种仪表板实用程序和工具紧密关联，使管理员能够深入了解系统的情况。

未雨绸缪，远离麻烦

作为管理员，尝试提前解决集群问题通常是一个好主意。本节会提出一些建议，供你参考。

避免耗尽磁盘空间

如果你想避免出现磁盘空间不足的情况，一定要记住两个参数。任何软件平台都可能出现不希望出现的行为，你可以通过设置一些参数来预先保留一些存储空间：

--fs_data_dirs_reserved_bytes

--fs_wal_dir_reserved_bytes

这些参数的默认值是1%，也就是在数据目录和WA L目录的文件系统上保留磁盘空间的1%，用于非Kudu的使用。一开始时，将这个值设置为接近5%的范围比较合适（这个参数以字节数为单位，而不是百分比，因此你需要根据百分比计算出字节数）。

容忍磁盘故障

Kudu是一个快速发展的产品，现在它可以应对tablet服务器上出现的持久磁盘故障（sustaining disk failure）。不过Kudu master服务器还不具备应对持久磁盘故障的能力。因此，当一个磁盘发生故障时，整个Kudu master服务器都会不可用。

tablet服务器可以容忍磁盘故障，但是如果存储WAL或tablet元数据的磁盘发生故障，则tablet服务器仍会挂掉。由于Kudu也在不断成熟，你可以跟踪了解Kudu故障容忍能力的发展情况。

tablet的数据实际上会被分开存储到tablet服务器的多个磁盘上，默认为3个磁盘，如果这个值被设置为数据目录的个数或比它更大，则数据会分散到所有可用的磁盘上。如果tablet分配的数据目录较少，则当磁盘发生故障时，tablet服务器上的所有tablet都受影响的可能性就比较小（假设没有出现tablet服务器上所有的tablet的数据都在分散到这个磁盘的情况）。

以下参数用于控制特定tablet副本的目标数据目录数：

```
--fs_target_data_dirs_per_tablet
```

备份

Kudu还没有核心备份功能，即使很快就会出现这种功能，你最好还是考虑一下各种数据备份的策略。

使用MapReduce、Spark或Impala，你可以读取Kudu表并将其写入HDFS，保留表模式，但是以Parquet格式写入。在数据保存到HDFS之后，可以将数据发送到灾难恢复区域中的其他集群，或者使用distcp工具将数据放入云中，甚至使用Cloudera的Backup Disaster Recovery（BDR，备份灾难恢复）工具进行备份，它会维护表的元数据，而且还有其他的好处。

另一个可考虑的方案是，即时将数据导入到两个独立的数据中心，特别是在数据是以流的方式写入时。当然，你必须留意正在进行写入操作的Kudu集群中有集群出现问题的情况，甚至还要留意数据写入表的速度，因为这两个集群可能因为距离的原因而在写数据时出现不同步的情况。不过，数据不太同步可能也能够满足需求，这取决于你的应用场景。

小结

本章介绍了Kudu中一些常见的管理员任务。我们帮助管理员规划他们的环境，讨论了Web UI，并介绍了一系列管理员需要熟悉的CLI命令。我们介绍了启用安全性的相关知识、管理员需要执行的常见操作，例如添加和删除节点，最后讨论了一些基本的性能调优和避免陷阱的方法。接下来，我们会关注开发人员的需求，即：作为Kudu的应用开发者，如何使用你选择的编程语言和框架来快速开始开发。

---

[1] 译者注：原文中并未指明“这些任务”是指哪些任务，我的理解是正常的对表的修改（使得表频繁变化）的任务和校验任务可以并行执行。

[2] 译者注：系统只有两个用户，但是实际使用者有很多，这些使用者可以共享这两个用户其中的一个，所以是每个用户可以作为一个整体（代表多个使用者）。

第5章 Kudu常用的开发接口

从本质上来说，Kudu是一个高度灵活、容错的分布式存储引擎，可以很好地管理结构化数据。你可以通过简单易懂的API将数据高效地写入或者读出Kudu。

对于开发人员而言，有多种方式可以与存储在Kudu中的数据进行交互。Kudu为以下编程语言提供了客户端API：

- C++
- Java
- Python

在与Kudu 交互时，也可以使用MapReduce 和Spark 等计算框架。MapReduce支持原生的Kudu输入格式，可以通过Java客户端来使用，而Spark的API提供了专门的Kudu Context，可以与Spark SQL深度集成。

由于Kudu以结构化、强类型的方式存储数据，因此自然而然也提供对Kudu的SQL访问。目前你不仅可以使用Spark SQL来访问和操作数据，还可以使用Impala。Impala是Hadoop生态系统中的开源原生分析型数据库，多个Hadoop发行版都包含它。Impala提供了对各种数据源的表的抽象，这些表可以存在于Kudu、HDFS、HBase或基于云的对象存储（比如Amazon S3）中。

本章我们将深入介绍各种客户端API，包括Spark，最后讨论Impala与Kudu的集成如何应用于各种类型的场景。本章所有的代码段都可以在我们的GitHub存储库中找到（http：//bit.ly/gswk-ch5）。

客户端API

Kudu的各种客户端API都有一个通用的工作流和一组相同的对象，用来设计客户端应用程序。无论你使用的是何种编程语言，都可以获取这些对象的实例，以便定义和操作存储在Kudu中的数据，并与之交互。

Kudu Client（客户端）

Kudu Client可以通过提供Kudu master服务器的列表来创建。这个对象可以用来执行以下操作：

- 检查表是否存在。
- 提交数据定义语言（DDL）操作，比如创建（create）、删除（delete）和修改（alter）表。
- 获得对一个Kudu Table对象的引用。

因此，你可以将其视为与Kudu交互的主入口。获取对Kudu Table对象的引用后，就可以开始操作表中的内容。

Kudu Table

对Kudu Table对象的引用一般可以让我们执行以下操作：

- insert/delete/update/upsert行。
- 扫描行。

表是有特定的表模式的，它具有强类型的列和分区策略，这些策略也需要由特定的对象来指定。

Kudu DDL

要定义新的Kudu表，就需要定义一个表模式（schema）。这个Kudu Schema对象包括所有列的名称、类型、是否可为空值的信息（nullability）和默认值。

接下来，需要定义分区方式。Kudu支持多种分区方法，包括按范围或哈希值分区。为此，我们定义了一个Kudu Partial Row（部分行）对象。

最后，还有一个Kudu Table Creator（表创建者）对象，它使用建造者（builder）模式，能更方便地提供定义表所需的所有方法。

总而言之，要想真正地创建表，你需要以下对象：

Kudu Schema

定义表的列。

Kudu Partial Row

定义用来做分区的列。

Kudu Table Creator

定义表的建造者的模式实现。

Kudu扫描器（Scanner）读取模式

扫描数据是一种常见操作，打开一个扫描操作之前，你需要熟悉许多重要的读取模式：

- READ_LATEST
- READ_AT_SNAPSHOT
- READ_YOUR_WRITES

READ_LATEST模式始终会返回在收到请求时已经被提交的写操作。如果用ACID（Atomicity，Consistency，Isolation，Durability）中的术语，这个模式和“Read Committed”隔离模式的效果相同。READ_LATEST模式是默认模式。它并不是可重复读取模式，因为从前一次执行扫描后数据会一直持续地被插入或提交，所以每次扫描返回的数据可能都不一样。

READ_AT_SNAPSHOT允许用户提供一个时间戳，并且尝试读取该时刻的数据。如果未提供任何时间戳，则将当前时间作为“快照”。因此，未来读取该数据集时应该会返回完全相同的行数据集。这种特别的读取模式可能会导致延迟增加，因为在此扫描返回之前，需要等待在这个时间戳之前发生的事务完成。在ACID的术语中，这种模式更接近于“Repeatable Read”隔离模式。同时，如果所有写入操作都是一致的（通过外部机制来保证），这就是“Strict Serializable”的隔离模式。

在撰写本书时，Kudu新添了READ_YOUR_WRITES模式，其效果是本次读取操作能够看到此客户端会话中已经进行的所有写入和读取操作。这种模式确保在会话中能够读取自己的写操作（read-your-writes）以及读取自己已经读过的操作（read-your-reads）的同时，最大限度地减少因等待未完成的写入事务完成所导致的延迟。在此模式下每次运行扫描时，可能都会返回不同的结果，因为在此期间可能发生了额外的写入。目前这是一个实验性功能，但请注意它在未来的Kudu版本中将成为一个稳定的功能。

C++API

尽管大多数传统的Hadoop生态系统组件都是基于Java虚拟机的服务，但Kudu是用C++编写的，所以Kudu能够做更多优化，不必依赖垃圾收集，并且对内存和CPU的使用也很高效。

使用C++编写客户端应用程序的好处是，你会是头等公民，随时可以使用Kudu最新和最强大的功能。相应地，你的应用程序也可以在资源利用率及其他方面做到精益求精。

可以在https：//kudu.apache.org/cpp-client-api/中查找最新的C++API。

让我们来看几个例子（代码片段可以在https：//github.com/kudu-book/getting-started-kudu/tree/master/chapter5/src/kudu/c中找到）。在开始之前，我们需要包含许多关键的API头文件：

```
#include "kudu/client/callbacks.h"
#include "kudu/client/client.h"
#include "kudu/client/row_result.h"
#include "kudu/client/stubs.h"
#include "kudu/client/value.h"
#include "kudu/common/partial_row.h"
```

如果要定义表，相关的关键文件包括client.h和partial_row.h。我们首先使用API提供的建造者模式创建并连接KuduClient对象：

```
// 创建并连接一个 Kuduclient 对象
shared_ptr<KuduClient> client;
KuduClientBuilder()
    .add_master_server_addr(masterHost)
    .default_admin_operation_timeout(MonoDelta::FromSeconds(10))
    .Build(&client);
```

可以看到，我们传入了master服务器的列表，定义在masterHost变量中，该变量应该是一个以逗号分隔的master服务器列表。如果你没有使用master服务器的默认端口号，那么还需要在每个master服务器的主机名后加上端口号。

我们可以使用KuduClient引用来检查一个表是否存在，以及删除表（如果这个表存在的话）。下面给出了一个例子：

```
shared_ptr<KuduTable> table;
Status s = client->OpenTable(kTableName, &table);
if (s.ok()) {
  client->DeleteTable(kTableName);
}
else if (s.IsNotFound()) {
  // 如果没有发现这个表，该做什么处理
}
else {
  // 处理错误
}
```

注意，在这里我们引入了Status对象。你最好了解一下这个类，这样在客户端代码中检查来自Kudu服务器的响应时，你就不会对用到的各种返回类型和代码感到陌生了。

我们开始使用建造者模式（Builder pattern）来定义表模式：

```
KuduSchema schema;
KuduSchemaBuilder sb;
sb.AddColumn("id") ->
Type(KuduColumnSchema::INT32)->NotNull()->PrimaryKey();
sb.AddColumn("lastname")->Type(KuduColumnSchema::STRING)->NotNull();
sb.AddColumn("firstname")->Type(KuduColumnSchema::STRING)->NotNull();
sb.AddColumn("city") ->
Type(KuduColumnSchema::STRING)->NotNull()->
  Default(KuduValue::CopyString("Toronto"));
```

采用建造者模式，我们就能够以编程的方式不断添加新列，指定列名、列类型、其是否可为空、默认值以及表的主键。

现在，将KuduSchema指针传递给建造者的Build函数以真正生成我们的对象：

```
sb.Build(&schema)
```

接下来，我们创建一组KuduPartialRow对象（表示行的一部分）的数组。每个KuduPartialRow对象都是通过前面定义的schema对象创建得到的。它表示具有这个表模式的行，你可以为它的一个或多个列提供值。在定义表时，我们要为那些分区键的列提供值，这些列值将决定分区表的切分点的位置：

```
vector<const KuduPartialRow *> splits;
KuduPartialRow *row = schema.NewRow();
row->SetInt32("id", 1000);
splits.push_back(row);
row = schema.NewRow();
row->SetInt32("id", 2000);
splits.push_back(row);
row = schema.NewRow();
row->SetInt32("id", 3000);
splits.push_back(row);
```

当每个KuduPartialRow对象都被添加到splits数组中后，我们就为例子中需要的分区模式定义了分区范围。

接下来，我们需要用一个列名称的数组来表示范围分区的列。此列表必须与我们刚才指定的切分点中定义的列匹配：

```
vector<string> columnNames;
columnNames.push_back("id");
```

使用到目前为止所构建的所有对象，我们终于可以调用KuduClient的表创建器来创建我们的表了。

我们可以看到这个建造者模式接收了表名、我们定义的表模式（包括主键）、范围分区的列名和行切分点、我们想要的副本数，最后调用Create方法在Kudu中创建了这个表。

Python API

Python API实际上只是C++客户端API的接口，安装PyPI包后即可轻松入门。如果要通过源代码安装Python，则需要先安装Cython，稍后会详细介绍。我们的GitHub代码库中提供了本节的完整代码示例（https://github.com/kudu-book/getting-started-kudu/tree/master/chapter5/src/kudu/python）。

```
KuduTableCreator *tableCreator = client->NewTableCreator();
s = tableCreator->table_name(kTableName)
    .schema(&schema)
    .set_range_partition_columns(columnNames)
    .split_rows(splits)
    .num_replicas(1)
    .Create();
```

准备Python开发环境

只需要两个简单的步骤，我们就可以开始开发了：

1.安装C++客户端的库和头文件。

2.安装kudu-python的PyPI库。

要安装C++客户端库和头文件，你只需要安装以下两个软件包：

```
sudo yum-y install kudu-client-devel kudu-client0
```

更多相关的信息，请参阅第3章的内容。

接下来，安装pip，请自行查找最新的pip文档（https://pip.pypa.io/en/stable/installing/）。

使用pip安装kudu-python软件包：

```
sudo pip install kudu-python
```

我们的示例用的是Python 3.5。我们采用从源代码编译的方式来安装Python 3.5。请记住，在编译Python 3.5之前，要先安装以下附加库，确保pip3能成功地安装：

yum-y install zlib-devel bzip2-devel sqlite sqlite-devel openssl-devel

安装Python 3.5之后，再安装kudu-python软件包：

sudo pip3 install kudu-python

使用Python开发Kudu应用

要使用Python开发Kudu应用，首先要导入常用的库：

```
import kudu
from kudu.client import Partitioning
```

所有使用Python开发的 Kudu应用程序都需要与Kudu服务器建立连接以创建client对象。连接Kudu服务器时只需要提供一个Kudu master服务器列表，并且你也可以在API中指定端口号，如下所示：

```
client=kudu.connect(host=kuduMaster,port=7051)
```

接下来，我们将调用Kudu的schema_builder API，得到一个建造者，然后用它来创建一个表。使用建造者的第一件事就是定义我们的列，稍后会举几个例子来说明在这些列中可以定义的内容。值得注意的内容包括提供列名、列类型、默认值和是否可为空的信息，不过更有趣的是，在每个列的级别指定压缩和编码方式以及块的大小。

```
builder = kudu.schema_builder()
builder.add_column('lastname').type(
          'string').default('doe').compression('snappy').encoding(
          'plain').nullable(False)
builder.add_column('firstname').type(
          'string').default('jane').compression('zlib').encoding(
          'plain').nullable(False).block_size(20971520)
```

你还可以直接在add_column方法中指定参数来定义新列，而不用前面的示例中所用的.xxx的方法：

```
builder.add_column('ts_val',
                   type_=kudu.unixtime_micros,
                   nullable=False, compression='lz4')
```

在定义列之后，我们定义主键的列，并实际调用build函数以获取kudu.schema对象：

```
builder.set_primary_keys(['lastname', 'state_prov', 'key'])
schema = builder.build()
```

考虑分区策略时，对于本例，我们将定义一个哈希分区列和一组范围分区的列。做哈希分区时需要知道将数据分区为多少桶，而做范围分区时则需要定义分区的范围。默认情况下，范围是包含下限而不包含上限的：

```
partitioning = Partitioning().add_hash_partitions(
            column_names=['state_prov'], num_buckets=3, seed=13)
partitioning.set_range_partition_columns('lastname')
partitioning.add_range_partition(['A'], ['E'])
partitioning.add_range_partition(['E'], ['Z'],
            upper_bound_type='inclusive')
```

现在创建表的准备工作已经做完了，传入表名、表模式、分区模式和tablet的副本数等这些参数：

```
client.create_table(tableName,schema,partitioning,1)
```

为了在表中插入行，我们再次使用client对象来获取表的句柄以及要用来做写操作的会话。我们的会话有一些可用的属性，例如超时值、刷新策略等：

```
table = client.table(tableName)
session = client.new_session()
session.set_flush_mode(kudu.FLUSH_MANUAL)
session.set timeout ms(3000)
```

在我们为表准备插入值时，可以在会话上为该操作发起apply方法。但是，这取决于不同的刷新模式，在Kudu服务器上可能并不会触发任何操作。因为在这个例子中我们使用了手动刷新模式，持续apply大量的插入操作，但实际上只用一次操作来完成它们：

```
op = table.new_insert({'lastname'  : 'Smith',
                       'state_prov': 'ON',
                       'firstname' : 'Mike',
    'key'       : 1,
    'ts_val'    : datetime.utcnow()})
    session.apply(op)
```

我们用一组键值对来表示要插入的行，然后在会话上apply这个操作，但是这个操作还没有被刷新出去（flush out）。我们可以接着执行多次之前的插入操作，如果在键值对列表中并没有出现某个列，它们将被自动设置为默认值或空值。如果这一列不允许为空值且没有指定的默认值，那么你可能必须指定这一列的值，否则插入操作都会失败。

如果准备好要刷新内容时，可以直接在会话上调用flush方法：

```
session.flush()
```

这里可能会抛出异常，也很适合处理错误：

与之前通过调用new_insert来执行插入操作类似，你还可以执行许多其他操作，都是每次操作指定一个键值对的方式。

以下就是所有的操作：

insert

使用new_insert API。

upsert

使用new_upsert API。如果该记录不存在就插入它，如果该记录已经存在，则用提供的新值来更新这条记录。

```
except kudu.KuduBadStatus as e:
        (errorResult, overflowed) = session.get_pending_errors()
        print("Insert row failed: {} (more pending errors? {})".
        format(errorResult, overflowed))
```

update

使用new_update API。

delete

使用new_delete API。

因此，如果你要对一些行进行insert、upsert、update或者delete操作，需要有这些要修改的行的键的列表。对于delete操作，不需要指定不属于键的其他列的值。同时，对行做更新时会使用所提供的值更新整个行。假设你要更新的行有以下内容：

keycol	col1	col2	col3
id1	3	hi	go

然后，你像下面这样更新这一行：

table.new_update（{ 'keycol ': 'id1 ', 'col1 ':5}）

Kudu会用你指定的值来更新整个行。因为你没有指定列col2或者col3的值，所以它们最终的值就是空值。因此，这一行的最终状态就会是下面这样：

keycol	col1	col2	col3
id1	5	-	-

如果你想要的效果是保留这两列更新之前的值，你需要先执行读操作获取i d1对应的行，这样会得到一个行对象，修改这个行对象中相应的列，然后用这个对象执行更新操作即可。

Java

Java是Hadoop生态系统中的头等公民，所以我们在与Kudu交互时很可能也会用到Java。在这里的示例中，我们会首先讲如何设置必要的依赖项。关于构建Java JAR文件有两种思路，即uber JAR和non-uber JAR。我们的GitHub代码库中有完整的示例代码（https://github.com/kudu-book/getting-started-kudu/tree/master/chapter5/src/kudu/java）。

uber JAR的优势是完全自包含，不管你在什么环境中运行，它们通常使用shade方法（把依赖包中的一些类重命名并打包到应用中以避免冲突）来确保应用程序中捆绑的库不会与运行环境中的任何库有冲突。

non-uber JAR的优势是它能使用环境中已经有的库，这样的话，JAR包本身就非常轻便，只包含应用自己的代码。只要你调用的API属于你运行的Kudu发行版，你的应用就能随着Kudu的升级而持续升级。

我们的示例演示的是如何使用non-uber JAR，因为它们很容易被传送到各种集群，做编译时简单而快速，并且更容易集成到现有环境中。

请查看我们提供的pom.xml示例文件以获取详细信息。我们首先要设置一些应该注意的关键依赖项。

我们在Kudu的标准Java应用中会用到以下插件：

- maven-compiler-plugin
- maven-shade-plugin
- maven-surefire-plugin

这些插件让我们能够编译代码并且在需要时对任何库做shade，还能为代码添加单元测试。

接下来，我们只需要添加对Kudu客户端库的依赖。目前，有两种方式可以提供Kudu客户端库：Apache提供的库和Cloudera提供的库。如果你是在Cloudera发行版Hadoop环境中运行Kudu，请指定你所运行的CDH版本对应的依赖项：

```
<dependency>
  <groupId>org.apache.kudu</groupId>
  <artifactId>kudu-client</artifactId>
  <version>1.5.0-cdh5.13.1</version>
  <scope>provided</scope>
</dependency>
```

你可以在本书的代码库提供的POM（Project Object Model）文件中查看与日志和单元测试相关的其他依赖项。

Java应用

我们在Java应用的开始导入以下关键的类：

```
import org.apache.kudu.ColumnSchema;
import org.apache.kudu.Schema;
import org.apache.kudu.Type;
import org.apache.kudu.client.*;
```

然后，利用由逗号分隔的master服务器列表，我们得到一个客户端连接：

KuduClient client=new KuduClient.KuduClientBuilder(KUDU_MASTER).build();

在本例中，我们将首先使用ColumnSchemaBuilder API定义一组列模式：

```
List<ColumnSchema> columns = new ArrayList(2);
     columns.add(new ColumnSchema.ColumnSchemaBuilder("key", Type.
INT32)
         .key(true)
         .build());
     columns.add(new ColumnSchema.ColumnSchemaBuilder("value",
Type.STRING)
```

.build());

这样我们就有了两列，一列是主键（key），另一列是存储值的（value）。然后，用这两列来创建我们的表模式对象：

Schema schema=new Schema(columns);

我们为范围分区的列定义了一个分区键的列表，并插入需要的键值：

List＜String＞ rangeKeys=new ArrayList＜＞();

rangeKeys.add(＂key＂);

到这里，我们就拥有了创建表所需的所有信息，包括表名、表模式，以及范围分区的信息。下面就开始创建表：

client.createTable(tableName,schema,

new CreateTableOptions().setRangePartitionColumns

(rangeKeys));

创建表之后，开启会话，为插入操作做准备。首先，使用客户端连接来打开表。接下来，创建一个你将使用的新会话，创建会话方法来接受超时设置和刷新策略等参数：

KuduTable table=client.openTable(tableName);

KuduSession session=client.newSession();

会话就位以后，我们为插入操作准备对象，使用Java中的newInsert API：

Insert insert=table.newInsert();

一般的思路是，有了Insert对象以后，从其中得到一个行对象（PartialRow）。当然，这个对象目前还是一个空行，只包含和这一行相关联的表模式以及一个API，我们可以用感兴趣的值来填充这一行：

PartialRow row=insert.getRow();

row.addInt(0,i);

row.addString(1,"value "+i);session.apply(insert);

根据你的session对象的属性，该行可能会也可能不会刷新到Kudu服务器。你可以在session对象上调用flush方法来触发刷新，或者根据指定的策略自动刷新。

当使用Java API查询Kudu中的表时，我们要在这个表上构建一个特定的扫描器。扫描器同样使用了建造者模式，我们还能为其提供一组想要投影的列（即通常在SQL查询的select子句中指定的列）：

```
// 设置我们想要投影的列
List<String> projectColumns = new ArrayList<>(1);
     projectColumns.add("value");
// 在表上构建扫描器
KuduScanner scanner = client.newScannerBuilder(table)
         .setProjectedColumnNames(projectColumns)
         .build();
```

现在，我们有了一个扫描器。虽然有很多的行，但我们可以不断循环来得到行结果集迭代器（RowResultIterator），利用这个迭代器扫描所有的行：

你也可以像通常那样在Scala应用中调用这个Java API，不过在大数据的上下文中，我们一般会使用Spark来处理大规模数据。

运行你的Java应用，看起来类似下面这样：

```
    while (scanner.hasMoreRows()) {
          RowResultIterator results = scanner.nextRows();
          while (results.hasNext()) {
            RowResult result = results.next();
            System.out.println(result.getString(0));
          }
        }
java -cp "kudu-java-client-1.0.jar:/opt/cloudera/parcels/CDH/jars/ \
  kudu-client-1.5.0-cdh5.13.1.jar: \
  /opt/cloudera/parcels/CDH/jars/kudu-client-tools-1.5.0-cdh5.13.1.jar" \
  -DkuduMaster=mladen-secure-kudu-8,mladen-secure-kudu-9,mladen-secure-
kudu-10 \
  org.apache.gettingstartedkudu.examples.KuduJavaExample
```

这条命令的关键是，我们把应用编译好的.jar文件设置到classpath中，在本例中就是kudu-java-client-1.0.jar，后面是以冒号分隔的JAR文件：

- kudu-client
- kudu-client-tools

在本例中，我们使用的.jar文件的位置就是Cloudera发行版中Kudu库的位置。对于独立安装的Kudu情况也类似，你要找到相同的库，并确保它们在你的classpath中。

使用-D标志来指定master服务器的做法仅限于我们这个应用的例子，并不是通用的。接下来，我们指定main类，如有必要的话，后面还可以跟上需要的参数。

Spark

Kudu有Spark专用的库，可供开发人员直接与Spark的Dataset和Dataframe框架集成使用。如果需要，它还可以直接挂接到RDD（Resilient Distributed Dataset）框架。我们的代码库（https://github.com/kudu-book/getting-started-kudu/tree/master/chapter5/src/kudu/spark）中提供了完整的代码片段。

你可以使用Java或Scala编写Spark应用程序与Kudu集成，但在写本书时，PySpark还不支持与Kudu集成。

当Spark任务在扫描Kudu表时，你可以安排任务从领导者和非领导者副本中读取数据。这样做可以提升性能，因为任务可以更容易地被安排在本地读取数据（而不是只能从领导者副本中读取，这是Kudu 1.7版本之前的情况）。

我们使用maven作为示例来编译Spark应用程序，在pom.xml文件中使用以下三个插件来满足我们的需求：

- maven-compiler-plugin
- maven-shade-plugin
- maven-scala-plugin

前两个插件基本上所有Java应用都会用到，而我们专门添加了第三个插件来编译我们的Scala代码.

虽然我们的示例中使用的是Spark 2，但其实Kudu可以同时运行在Spark 1.6和Spark 2上。如果使用Spark 2的话，也必须使用基于Scala 2.11版的库，而不是较老的2.10版本的库。有关的详细信息，请参阅pom.xml示例文件，不过运行Spark SQL时，尤其是使用Kudu运行时，需要以下依赖项。

kudu-client

和Java API使用的客户端相同。

kudu-spark2_2.11

Kudu的Spark 2绑定。

其余的依赖项都是Spark应用程序常用的，包括：

scala-library

Scala编程所需要的库。

spark-core_2.11

必需的Spark核心库。

spark-sql_2.11

Spark SQL特有的库。

spark-hive_2.11

Spark Hive库，用来访问和操作Hive的metastore。

我们在Spark应用程序中要导入的Kudu特有的内容包括：

import org.apache.kudu.spark.kudu._import org.apache.kudu.client._

然后，设置一个Spark会话，启动我们的Spark应用程序：

valspark=SparkSession.builder.appName("SparkonKuduGettingStarted")

.enableHiveSupport().getOrCreate()

Spark操作Kudu的方式与其他API不同，其他API通常一开始就是通过某个Kudu client对象来得到客户端连接。在Spark中，我们设置一个KuduContext对象，它可以在集群全局级别自动为我们在内部处理客户端和会话。

KuduContext的参数包括一个由逗号分隔的带端口号的master服务器列表（如果使用默认值，就不需要指定端口号）和一个对SparkContext的引用：

```
val kuduMasters = Seq(master1, master2, master3).mkString(",")

// 创建一个 KuduContext 的实例
val kuduContext = new KuduContext(kuduMasters, spark.sparkContext)
```

接下来，我们创建一个Map，其中保存一组可在多个操作中重用的Kudu选项，即包含键值对的Map [String，String]，这些键值对可以将参数传递给不同的操作。首先，指定我们正在使用的Kudu表和master列表：

```
val kuduOptions: Map[String, String] = Map(
      "kudu.table" -> kuduTableName,
      "kudu.master" -> kuduMasters)
```

有了Kudu Context，我们就能够轻松检查某个表是否存在，还可以删除表：

```
if (kuduContext.tableExists(kuduTableName)) {
  kuduContext.deleteTable(kuduTableName)
}
```

要创建表，首先需要定义表模式。定义一个表模式就像定义一个常规的Spark DataFrame一样。指定一个StructType，后面跟一组StructField类型的字段，它们以列名、类型和是否可为空（nullability）作为参数：

```
val kuduTableSchema = StructType(
//           列名     类型        是否可为空
StructField("name", StringType , false) ::
StructField("age" , IntegerType, true ) ::
StructField("city", StringType , true ) :: Nil)
```

接下来，我们需要一个构成主键的列名列表。在我们的示例中，只有一个名为name的主键列：

val kuduPrimaryKey=Seq("name")

然后，在kuduTableOptions对象中指定分区信息和副本数：

```
val kuduTableOptions=new CreateTableOptions()
kuduTableOptions.
setRangePartitionColumns(List("name").asJava).setNumReplicas(3)
```

关于CreateTableOptions方法，需要特别注意一点：范围分区列的列表必须是基于Java的列表，而不是基于Scala的列表。因此，我们需要对列表用.asJava方法进行转换。

现在我们可以真正创建表了，再次使用Kudu Context的API来调用：

```
kuduContext.createTable(
// 表名、表模式、主键和选项
kuduTableName,kuduTableSchema,kuduPrimaryKey,kuduTableOptions)
```

我们还可以用Kudu Context直接进行更多的操作，不过需要提供Kudu表名称以及想要写出或者操作的DataFrame。以下是Kudu Context对象上的一些API调用：

- kuduContext.insertRows
- kuduContext.deleteRows
- kuduContext.upsertRows
- kuduContext.updateRows

对于insert、upsert和update API调用，你需要指定具有键和值的DataFrame和要操作的表。但是，对于delete API调用，你的DataFrame只需要指定键。

现在，大多数Spark应用和集成模式都不需要KuduContext这样额外的对象来执行Spark逻辑，而是在DataFrame对象内部就做好了集成。Kudu也不例外，通过指定前面提到的Kudu选项（Map [String，String] 格式的选项集），我们可以进行以下的调用：

```
customersAppendDF.write.options(kuduOptions).mode("append").kudu
```

以这种非常简洁、优雅的方式，我们指定了一个DataFrame，指定它要做写操作，指定需要的Kudu选项（包括Kudu集群的master服务器

和表名），

并指示它插入（追加）行到指定的Kudu表。

我们还可以通过Spark的read操作在Spark中注册表：

spark.read.options(kuduOptions).kudu.registerTempTable(kuduTableName)

做了这些工作以后，就可以使用Spark SQL语法往此表中直接插入数据：

spark.sql(s " INSERTINTOTABLE $kuduTableNameSELECT*FROMsource_table ")

要读取一个Kudu的表并获得其DataFrame对象，最简单的方法如下：

spark.read.options(kuduOptions).kudu

最后，Kudu还支持在where子句中对谓词的下推操作，例如在下面的语句中：

val customerNameAgeDF=spark.

sql(s " " " SELECT name,age FROM $kuduTableName WHERE age ＞=30 " " ")

Spark利用Kudu master服务器上的数据位置信息，尝试让运行在指定节点上的Spark任务读取本地的数据。这样做能够提高整体的读入性能。

在Impala中使用Kudu

通过客户端API访问Kudu当然是可行的，我们也建议你用这种方式在Kudu上构建应用程序。而另一方面，如果采用Spark和Impala等框架，你不仅可以使用Spark等可编程可扩展的框架处理Kudu中的数据，而且可以使用SQL语法与Kudu交互，充分利用Impala和Spark SQL强大的大规模并行处理（Massively Parallel Processing，MPP）能力。

Kudu与SQL框架能够很好地集成，因为它具有强类型字段和几乎固定的列结构，可以轻松映射到SQL引擎。Kudu有表的概念，但这个表与Impala中的表是不同的。

例如，Impala表是在Kudu表之上定义的，这意味着提供给Impala的表名不一定是Kudu表的表名。对于Impala来说，Kudu只是它所包含的另一个存储层，类似于S3或HDFS存储层。此外，Impala具有序列化/反序列化器（SerDes），而且记录输入格式，当数据位于S3或HDFS中时，它可以处理这些格式。然而，当谈到Kudu时，Impala是把Kudu作为一个单独的存储层与之交互的，因为Kudu能够做快速查找（lookup），特别是能对大量结构化数据做快速扫描。这样，我们就能够通过Impala快速轻松地驱动聚合或分析类查询。

例如，Kudu之上的Impala表可能是像下面这样定义的：

CREATE EXTERNAL TABLE ＇impala_kudu_table ＇ STORED AS KUDU TBLPROPERTIES(

```
 'kudu.table_name' = 'my_kudu_table',
 'kudu.master_addresses' = 'master1:7051,master2:7051,master3:7051'
);
```

从这个定义中，我们可以看到Kudu的表名与Impala的表名是完全分开的；还可以看到，我们不需要定义列和列的数据类型，因为这些定义是直接从底层Kudu表获得的。最后，我们需要指定这个表所属的Kudu集群的master服务器地址列表。因此，在一个Impala中，可能实际上存在着不同Kudu集群的表。当然，这并不理想，因为这样就丢失了Impala进程和Kudu tablet服务器之间的数据局部性。但是，在这种架构模型下你拥有这种灵活性。

Impala还在你执行的查询中提供谓词下推功能，其中谓词会在存储层，也就是Kudu中执行，而不是让记录数据流回Impala后执行。这能极大地提升性能。

你还可以将Impala用作数据迁移的工具，将数据从一个存储层传输到另一个存储层。例如，你可以查询HDFS中定义的表，然后将查询结果插入Kudu表。

实际上，Impala鼓励在Kudu、HDFS和其他存储引擎之间做join操作，Impala会根据查询计划各个部分的存储层生成不同的执行计划。

第6章 表和模式设计

本章将介绍Kudu中的模式设计（schema design），目的是解释一些有助于项目顺利进行的基本概念和原语。理想的模式能让读取和写入操作在集群中均匀分布，还能使执行查询时处理的数据最少。我们相信，理解本章所述的基础知识后，你就能设计出更接近理想结果的模式，为项目的成功助力。

Kudu项目本身有出色的模式设计文档（https：//kudu.apache.org/docs/schema_design.html），因此即使与文档内容存在一些重叠，我们还是会讨论那些特别重要的主题并提供额外的背景知识。

在所有的数据存储系统中，模式设计都非常重要，它也是许多令人头疼的问题的源头。在关系型数据库中，如果模式没有设计好可能会导致一系列问题，从消耗大量资源到数据损坏等，不一而足。如果你使用HBase和Cassandra，在设计一个模式之前需要广泛了解数据的访问方式，绝大部分项目发生阻塞的原因是查询速度慢，并且查询速度慢的原因几乎总是使用的资源过多。在Kudu中，模式设计同样重要，但Kudu提供了其他系统没有的一些功能，这些功能使其能被应用到更大范围的场景中。

模式设计基础

本节为那些没有阅读模式设计官方文档的读者介绍Kudu模式设计的基础知识：

- 表至少要有一个主键。
- 只有主键才会被索引（截至Kudu 1.6版本时如此）。
- 不能更新主键。
- 只有主键才能被用来做表分区（范围分区或哈希分区）。
- 大多数的表使用哈希分区，包括同时使用范围分区的时间序列类的应用。注意，其他一些应用也可以使用范围分区。

● 每一列都有类型、编码方式和压缩方式。编码方式有合适的默认值，而压缩方式的默认值是不压缩。

表6-1展示了每一种列类型可能的编码方式。

表6-1 列类型及其编码方式

列类型	编码方式	默认值
布尔	Plain, Run-Length	Run-Length
8 位有符号整数	Plain, BitShuffle, Run-Length	BitShuffle
16 位有符号整数	Plain, BitShuffle, Run-Length	BitShuffle
32 位有符号整数	Plain, BitShuffle, Run-Length	BitShuffle
64 位有符号整数	Plain, BitShuffle, Run-Length	BitShuffle
UNIXTIME_MICROS（从 UNIX epoch 开始的 64 位微秒数）	Plain, BitShuffle, Run-Length	BitShuffle
单精度（32 位）IEEE-754 浮点数	Plain, BitShuffle	BitShuffle
双精度（64 位）IEEE-754 浮点数	Plain, BitShuffle	BitShuffle
UTF-8 编码字符串（没有压缩时最大为 64 KB）	Plain, Prefix, Dictionary	Dictionary
二进制（没有压缩时最大为 64 KB）	Plain, Prefix, Dictionary	Dictionary

在线事务处理/在线分析处理混合的模式设计

正如之前详细讨论过的那样，Kudu是一个基于列的存储引擎，使用基于quorum的复制机制保证数据的持久性。它主要用于在线分析处理（OLAP）和混合事务/分析处理（Hybrid Transactional/Analytical Processing，HTAP）类型的工作负载。这一节旨在讨论HTAP环境下的模式设计。简而言之，HTAP系统面临的挑战是在处理运营工作（包括小型的事务、更新和分析）的同时处理分析型工作（包括对一部分列的大型扫描）。

首先，让我们想象一下HTAP的用例。比较合适的例子的是物联网（IoT）设备，例如联网的汽车。在这个用例中，有一些使用模式：

● 工程师和数据科学家希望跨设备地执行数据分析，以便他们能够了解缺陷部件的影响或训练机器学习模型。

● 客服人员需要访问某台设备的最新数据，以便实时为客户排除设备故障。

● 管理层希望能看到跨设备的汇总数据的报告。

● 客户希望以实时仪表板和报告的形式访问他们的数据。重要的是，客户可能会看到稍微久远一点的数据，但互联网用户已经习惯了

实时获取所有数据。

在过去，要实现这个用例需要构建一个高度复杂的系统。要么构建一个Lambda架构的系统，要么执行所谓的OLTP/OLAP拆分，这两种系统都将数据拆分到两个不同的系统中，因而导致许多问题和复杂性。

Lambda架构

Lambda架构将数据和工作负载分为实时层（speed layer）和批处理层（batch layer）。我们使用实时层进行检索以及分析最新的数据，使用批处理层分析历史数据。这样做会引发两个特别的问题：

- 有两个代码库，会在很多模块间移动代码，故障处理和运维工作都非常复杂。几乎没人或者很少有人愿意运维Lambda架构。
- 作为在HDFS和Kafka这些不可变存储上实现过多个系统、有多年经验的过来人，我们可以告诉你，在数据不会被更新的用例中，重跑数据（restatement）是正常现象。

OLTP/OLAP拆分

在这个系统中，最新的数据存储在O LT P关系型数据库中，然后被批量复制到OLAP系统以便稍后进行分析。虽然这是做数据分析的传统方法，但它存在许多问题，比如：

- 分析型应用无法访问最新的数据。当需要查询最新的数据来做分析时，这个问题就会产生现实的影响，例如“我们对软件所做更新是否就是导致我们目前看到的问题的原因？”此外，管理层和其他人一样，也越来越期望能够实时得到数据。
- 关系型数据库具有真正的可扩展性限制，通常会导致每个设备只能存储有限的生命周期的数据。这不能满足用户的期望，他们期望能够永远看到“他们的数据”。
- IT人员必须管理两个系统和复杂的数据复制过程。

有了Kudu，我们就能用一个系统解决这两个用例的问题。通过使用包含时间戳和设备标识符的主键，我们可以为客户和客服人员高效地查询每台设备的数据。此外，由于Kudu是为分析类负载而构建的，

因此我们可以轻松地为工程师、数据科学家和管理人员执行大量数据的全扫描。

使用固态驱动器（S S D）作为存储，将为这一类用例提供很大的便利，因为查询每台设备的数据时通常要检索大量的行，这会导致大量的磁盘搜索操作。但是，从长远来看，有一种设计可能会消减SSD带来的好处。如前所述，破碎镜像（fractured mirror）机制（http：//www.vldb.org/conf/2002/S12P03.pdf）可以让quorum中的一个或两个节点以行格式存储数据，而其余节点以列格式存储数据。OLAP类型的工作负载可以在列存储格式的节点上运行，而OLTP类型的工作负载可以在行存储格式的节点上运行。

主键和列的设计

设计表的模式时，最重要的一项决定是决定你的主键。你只能将主键用作分区模式的一部分，但是无法更新它们。其他列可以通过UPDATE或UPSERT选项更新（我们稍后讨论分区）。Kudu表必须有一个或多个列被定义为主键。这些列是不能为空、不可更改的，而且不能是浮点或布尔类型，还必须能够定义唯一的行。但也许更重要的是，因为Kudu缺少二级索引（直到Kudu 1.6版本时还是如此），而没有所有主键的谓词，扫描就会变成全表扫描。对于分析类查询，这是可以接受的，但是对于你想执行的类似小规模查找之类的查询，全表扫描则是不可接受的。

在下面的例子中，我们使用“连续（continuous）”这个词与“批量（batch）”做区分。

我们来看一个例子。我们为StreamSets编写了Kudu查找处理器的第一个版本。StreamSets是一个连续的数据输入工具，允许你在输入过程中执行记录的转换。查找处理器允许开发人员将其他字段添加到一个StreamSet数据流的一条记录中。由于StreamSets是一个连续的输入过程，对每条记录执行大量操作会导致输入时间过长。因此，在开

发处理器时，输入的记录中拥有与主键相关联的值是非常重要的，这样就可以利用主键索引完成查找，而且其查找速度足够快，能连续对单条记录进行处理。也由于这个原因，在StreamSet中，如果传入的记录中没有必需的主键值，则会将记录发送到错误处理路径中。

图6-1显示了StreamSet查找处理器的运行流程。

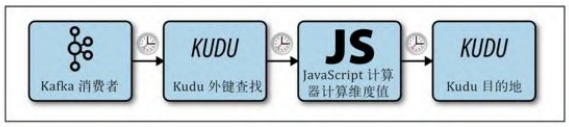

图6-1 StreamSet查找处理器

列模式的其他注意事项

对于所有的列，要特别考虑一下编码和压缩。这两个转换操作都能够减少数据的尺寸，但它们采用的是不同的折中方式，而且所减少的数据尺寸也是不同数量级的。你读到本书这里的时候，应该已经知道Kudu采用的是一种列式数据存储方式。列式数据存储具有更广泛的编码方式，因为相同类型的值会按顺序排列存储在一起。压缩是指采用一种通用算法来减少数据尺寸。在列式数据存储中这两种机制都存在，某些编码方式中包含了压缩算法。

在本节中，我们将详细讨论几种编码类型，以帮助你了解编码与压缩的区别。然后，我们会讨论每种编码类型的实际例子。Kudu目前支持Plain、Run-Length、BitShuffle、Dictionary和Prefix编码。

Plain编码

数据以其自然的方式编码。比如，32位整数就被编码为定长的32位格式。

Run-Length编码

如果某个值连续、重复地出现，就仅存储这个值及其出现的次数。以主键排序后如果列中含有很多重复值，这种编码方式会比较有效。

BitShuffle编码

BitShuffle编码是把一个转换操作和LZ4这种通用压缩算法结合起来，所有的值会被重新排列，先存储每个值的最高位，然后是次高

位，依此类推，对重新排序后的结果使用L Z4算法压缩。如果以主键顺序排列后，列中有很多重复的值或者值的差别很小，这种编码方式会比较有效。

Dictionary编码

将所有不重复的值构建成字典来对数据编码。数据被编码为字典中的索引。如果一列包含少量的唯一值，这种编码方式比较有效。如果该列存储了像uuid这样的唯一值，Kudu将会退回到Plain编码。

Prefix（前缀）编码

在存储一个值时，其与上一个值相同的前缀部分会被压缩。如果以主键顺序排列后其列值有公共前缀，那么这种编码方式就比较有效。

对于每一列，都需要选择编码方式或采用该列类型的默认编码方式。默认编码方式已经非常好了，但是对于字符串，你可以考虑更改默认编码方式。字符串类型的默认编码方式假定其列的值的基数（cardinality）很小。如果你的字符串没有公共前缀，比如像uuid这样的唯一字符串，则可以考虑使用Plain编码和LZ4压缩。如果字符串有公共前缀，那么可以考虑Prefix编码和LZ4压缩。请注意，如果相关的列是主键中的第一列，则Prefix编码可能是最佳选择，因为行在tablet中是按主键排序的。

例如，我们用一个字符串作为主键创建了三个表，并使用Python的uuid.uuid4（）函数生成主键，这样主键就没有公共前缀。其中，第一个表采用Plain编码、LZ4压缩，第二个表采用Dictionary编码、LZ4压缩，第三个表采用Prefix编码、LZ4压缩。前两个表最终的大小为102 MB，第三个表大小为70 MB。前两个表的大小相同，因为uuid是唯一的，因此Dictionary编码除了在编码时消耗了CPU之外没有其他影响。由于该表只有一列，并且行在tablet中按主键排序，因此经过Prefix编码后，表的性能最好。

作为第二个测试，我们创建了和上面相同的三个表，但是添加了一个整数列作为主键的第一列。这个整数是经过Plain编码和未压缩的，因此在几个测试表中它的大小不会发生变化。结果是，Plain编码和Dictionary编码产生了180 MB数据，而Prefix编码产生了178 MB数

据，或者说其压缩率高了约1%。为了测试编码方式的CPU开销，我们修改了Kudu中的src/kudu/cfile/encoding-test.cc文件来编码数百万个随机字符串，结果是Prefix编码的CPU开销大约为5%。很难说哪种编码方式会对特定用例的性能产生更大的影响，但考虑到计算机体系结构的发展趋势、I/O带宽的增长以及CPU性能相对有限的增长，我们不会为uuid添加Prefix编码，除非它是主键中的第一列，或该uuid具有公共前缀。

接下来，我们将帮助你可视化两种编码：Run-Length和BitShuffle。但请注意，这并不是Kudu实际使用的编码的实现方式，而是仅用于演示的经过大量简化的版本。

我们先讲Run-Length编码，以了解它与压缩的区别。首先，假设我们有一个包含100个布尔值的列表，前50个值为true，紧接着的49个值为false，最后一个值为true。如果用0表示false而用1表示true，我们可以将列表的值编码为111……000……01的二进制串，其中每一位为一个值，这会占用100 bit的存储空间。

或者，我们可以使用Run-Length编码，存储值和这个值重复出现的次数。例如，假设以8位（一个字节）来存储一个run。第一位表示值（true或false），其余7位代表true或false值的数目。用7位二进制数，我们可以存储的值的最大范围是0～2^7，或者说0～127。如果使用Run-Length编码来存储前面那个布尔序列，我们就用一个字节来编码第一串true值，再用一个字节来编码中间的那一串false值，然后再用一个字节来编码最后一个true，总共使用了24位或3个字节。与它相比，之前使用一个位存储true或者false值的方法，需要用100位。图6-2对Run-Length编码做了可视化的呈现。

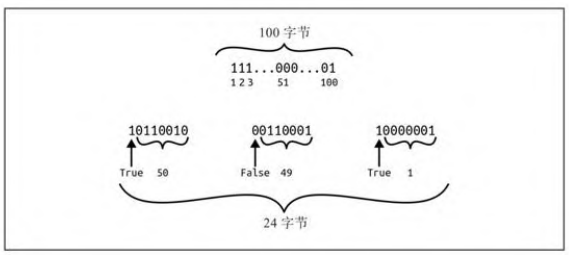

图6-2 Run-Length编码示例

现在，我们来讨论BitShuffle。一个有趣的事实是，BitShuffle于2015年发布，其原本是用来压缩加拿大氢强度测绘实验的数据的

（http：//bit.ly/2KSEwEH）。科学被应用到更广泛的公共领域，这是一件大好事。总的来说，BitShuffle方法就是重新排列给定数据集中的二进制位，以便更好地压缩。我们观察到，一个给定列中的数字通常是高度相关的，比如单调递增。鉴于此，我们可以重新排列位，以便相似的部分能放在一起，而其差异则放在最后。

例如，假设我们有三个无符号byte类型的数：1、2和5，用二进制表示分别为00000001、00000010 和00000101。将它们以二进制流的形式保存就是000000010000001000000101。现在我们把这些位转换一下，先保存最高位，然后是次高位，依此类推，最后我们会得到000000000000000001010101。这显然更易于压缩。事实上，在这个例子中也正是如此，BitShuffle能够使LZ4算法的压缩效率提升50%。当然，在读取这些数时，也需要转换位的位置。图6-3所示为BitShuffle编码的可视化表示。

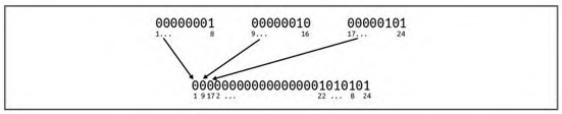

图6-3 BitShuffle编码示例

那么，哪一种编码方式更好呢？看情况而定。不过，一般Run-Length编码用于整数，因此很多人可能想知道，为什么Kudu中整数的默认编码方式是BitShuffle而不是Run-Length。因为我们认为BitShuffle是一种更好的通用的编码方式。

在下面的例子中，我们编码了两组数字：一组是从1到32，768的单调递增的整数列表，另一组是小于1，000，000的随机正整数。这两种类型的数在数据库中都很常见。例如，主键通常是单调递增的整数，因此采用Run-Length编码就很有效。相对随机的正数在维度表的外键中很常见。正如你所看到的，对于单调递增的数来说，Run-Length编码的效率比BitShuffle编码要高三倍。不过，它们都表现得相当不错，将32 KB的数据压缩到不到1 KB。对于随机的小正整数，情况就大不相同了，BitShuffle编码实现了大约1.6：1的压缩，而Run-Length编码实际上增加了数据的大小，不过我们验证过，数据大小的增加并不是由于采用了LZ4压缩算法。本练习的源代码位于本书代码库的chapter6文件夹下，在BitShuffleRunLengthComparison.java文件中。表6-2总结了测试的输出结果。

```
==== Monotonically Increasing
Plain Size: 32768
Plain LZ4 Compressed Size: 32875
Plain Zlib Compressed Size: 16513
Plain Snappy Compressed Size: 32791
BitShuffled LZ4 Compressed Size: 993
RunLength Encoded LZ4 Compressed Size: 297
==== Random Small Positive
Plain Size: 32768
Plain LZ4 Compressed Size: 32898
Plain Zlib Compressed Size: 25887
Plain Snappy Compressed Size: 32794
BitShuffled LZ4 Compressed Size: 20662
RunLength Encoded LZ4 Compressed Size: 41172
```

表6-2 对数据压缩和编码效果的总结

数据类型	编码	压缩	大小	原始大小的百分比
单调递增	Plain	无	32,768	100%
单调递增	Plain	LZ4	32,875	100%
单调递增	Plain	zlib	16,513	50%
单调递增	Plain	Snappy	32,791	100%
单调递增	BitShuffle	LZ4	993	1%
单调递增	Run-Length	LZ4	297	3%
随机小正整数	Plain	无	32,768	100%

续表

数据类型	编码	压缩	大小	原始大小的百分比
随机小正整数	Plain	LZ4	32,898	100%
随机小正整数	Plain	zlib	25,887	79%
随机小正整数	Plain	Snappy	32,794	100%
随机小正整数	BitShuffle	LZ4	20,662	63%
随机小正整数	Run Length	LZ4	41,172	126%

希望你现在明白了编码与压缩之间的区别。压缩可以以通用的方式，应用于任何类型的数据，并且它的影响通常比编码要小（由数据集决定）。在Kudu中，除了BitShuffle（其本身使用LZ4算法）之外，压缩是在对数据编码之后进行的。请注意，我们不建议在BitShuffle之上使用另一层压缩，但是，在写本书时，Kudu允许你对一个用BitShuffle编码过的列再进行压缩。Kudu支持的压缩格式有LZ4、Snappy和zlib。除非有特殊原因，我们坚持使用LZ4。它实现了与Snappy类似的压缩比，但速度更快，因此消耗的CPU更少。zlib消耗的CPU最多，应该避免使用它。总的来说，BitShuffle是一种很好的默认编码机制。从图6-4可以看到，Kudu允许我们在BitShuffle之上启用压缩。

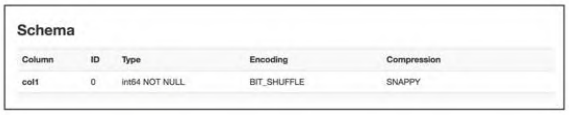

图6-4 Kudu中已经用BitShuffle编码过的列又启用了Snappy压缩

分区的基础知识

分区是把数据集划分为几个不重叠的集合的机制。我们会出于对性能、数据可用性和负载均衡等因素的考虑来做分区。有很多这样的机制，比如，轮询分区（round-robin partitioning）就是简单地轮换接收数据的分区，把数据分成n个分区。假设有3个分区（0、1、2）和5个元素（a、b、c、e、f），那么数据将按如下方式分配：（0=a，e；1=b，f；2=c）。这种方式的主要缺点是，如果不查询每个分区，就无法回答“元素c被分配给了哪个分区”这个问题。也就是说，在搜索c这个元素时，无法“排除”一些分区。

范围分区

范围分区相当直接。比如，假设我们尝试将0～9的整数分到两个分区，可以将0（含在内）到5（不含在内）的数分配给用 [0，5），表示分区0，而分区1可以存储 [5，10）。此模式也适用于对非整数型数据分区，比如二进制字节数组。

哈希分区

做哈希分区时，首先要选择需要哈希的列以及确定数据要分布在多少个桶中。然后，Kudu使用哈希函数来决定一个给定的行应存在哪个桶中。假设我们有一个带有单个整数主键的表，这个表是哈希分区的，而且我们已经选择4个哈希桶（b0、b1、b2、b3）。现在，假设我们有6行数据，其主键是（1、12、54、99）。再进一步假设我们有一个函数，如果给定一个整数，它就只返回这个整数。在数学中，这被称为恒等函数，它是一个既简单又有效的哈希函数。

我们来看这些行是如何分布的：

```
Key 1:  identity(1)  % 4 = 1 => Bucket b1
Key 12: identity(12) % 4 = 2 => Bucket b2
Key 54: identity(54) % 4 = 2 => Bucket b2

Key 99: identity(99) % 4 = 3 => Bucket b3
```

Kudu使用了一个名为Murmur2的哈希函数，它以速度快而著称，而且质量比较高，因为它几乎没有碰撞。

有HBase经验的读者会了解，创建新的主键，当到了键范围的末尾时，会产生热点问题。例如，在HBase中，如果你的键是一个时间戳，并且你将其存储为字符串（如20170910），则所有新的插入操作都将

被发送到单个服务器上的单个HBase Region中。要解决此问题，用户必须使用哈希桶之类的东西为键加前缀或者“加盐”。在Kudu中也可能发生同样的事情，但Kudu使这个问题的解决变得更加容易。你可以结合分区模式来解决这个问题。只需要在主键上添加一列，并在该列上添加哈希分区即可。例如，在时间序列应用中，你可以将时间戳和另一个列定义为主键，对时间戳做范围分区，对另一列做哈希分区。不需要手动“加盐”！

模式的更改

Kudu支持快速更改模式，这意味着它支持的更改不会导致表被锁定几分钟或几小时。这是因为数据不会在modification（修改）时被重写，而是在之后做compaction时被重写。目前Kudu（Kudu 1.6版）支持以下修改：

- 重命名表。
- 重命名主键的列。
- 重命名、添加、删除非主键的列。
- 添加和删除范围分区。

如果想了解模式更改的更多实现细节，请阅读文档https：//github.com/cloudera/kudu/blob/master/docs/design-docs/schema-change.md。

使用Impala访问Kudu时，你还应该了解一些事项。首先，要知道Hive/Impala有内部表和外部表的概念。对于内部表，Hive/Impala管理其存储，而对于外部表则不管理其存储。这意味着，如果你创建一个Hive/Impala内部表，它们将为你创建Kudu表，而当你删除Hive/Impala中的内部表时，它们将删除相应的Kudu表。你可以修改表的EXTERNAL属性（一个字符串布尔值，TRUE/FALSE），将Hive/Impala的Kudu表移入和移出内部表和外部表。更改Hive/Impala中Kudu表的名称并不会使其底层的Kudu表发生变化，因为其他客户端可能正在访问这个表。你可以通过表的kudu.table_name属性更改底层Kudu表的表名。更多详细信息可以参考

https：//kudu.apache.org/docs/kudu_impala_integration.html #_altering_table_properties。

最佳实践和提示

我们接下来介绍Kudu中模式设计的几个最佳实践和提示/窍门。值得注意的是，由于Kudu还处于起步阶段，随着社区成员学习在生产环境中高效地利用该技术，这些最佳实践将会发生变化。

分区

大多数表至少会有一个列是哈希分区的，并且可能还会有另一个列是范围分区的。一个表只有范围分区的情况很少见，因为哈希分区能避免许多范围分区的表会出现热点的问题。典型的例子是一个时间序列数据集，其新生成的记录一直插入到时间范围的一端，从而产生热点。

大对象

二进制/字符串类型的列的大小是有限制的，在未压缩的情况下最大为64 KB，这意味着Kudu在执行任何压缩之前都会检查数据的尺寸。虽然可以使用一个不安全的配置项来增加这个阈值，但目前这个值是合适的，因为Kudu尚未在使用了大于64 KB的对象的场景下进行测试。所以，如果你要存储较大的二进制/字符串对象，可以在存储它们之前先对其进行压缩，然后将Kudu的编码方式转换为Plain编码并关闭压缩。我发现64 KB的JSON、XML或者文本能够用Gzip压缩到很小，所以很难超过这个阈值。如果你的数据对象比64 KB大得多，可以把对象保存到HBase或者HDFS中，然后在Kudu中保存该对象的外键。

decimal（十进制数）

Kudu 1.7版本中添加了decimal类型。它是一种具有固定范围和精度的数值数据类型。你可以在主键中使用它（而其他类型比如float和double则不行），在C++、Java、Python、Impala和Spark中都可以使用这种类型。对于金融场景，或者在一些计算中，float和double的舍入行为不切实际甚至是错误的，这种数据类型就特别有用。请记住，

Kudu客户端1.6版或更早的版本不能以任何方式使用包含decimal类型的表。此外，如果你创建了带decimal类型的表，就无法降级到Kudu的早期版本了。

在decimal被引入之前的早期Kudu版本，会经常使用float和double类型，但是请记住它们与decimal是不同的类型，因此不适用于金融交易场景。我们将这些类型的值存储为字符串，然后在Impala或Spark SQL中将它们转换为decimal类型。由于对这些列的谓词求值不会被下推到Kudu中执行，因此性能会受到影响，但对于我们的数十个客户来说，这样做的效果很好。

不重复的字符串

如果一个表的主键是单个字符串，Prefix编码的效果最好；否则，对于这种类型，我们使用Plain编码和LZ4压缩。

压缩

BitShuffle编码的列会自动使用LZ4压缩。我们为所有其他的列打开了LZ4压缩选项。根据Percona的详尽测试（http：//bit.ly/2uguoLD），LZ4通常比Snappy快。

对象的命名

表名在Kudu中必须是唯一的。如果你通过Impala创建表，Impala将在表名前加上impala：：database.table这样的前缀。因为Impala中database.table的组合被强制为唯一的，所以你不会遇到麻烦。

表和列的名称都用小写。用户在Impala中查询表时是不区分大小写的，但通过API写入表时又区分大小写，所以都用小写的话可以避免混乱。

列的数量

请记住，在K u d u中，你应该将表的列数量保持在300以下。如果你的源系统中列的数量超过此数，把数据写入Kudu时，要将源系统拆分为多个300列的存储桶。每个桶都要保留主键，以便可以将它们通过视图合并在一起。

二进制类型

请记住，Impala没有二进制类型。如果你需要在Kudu表中存储一个二进制类型的值，首先要问自己，“我要用它来做什么？真的需要在Kudu中存储它吗？”如果确实需要这么做，请在Impala中将该列创建为字符串类型，将二进制数组值用base64编码，然后插入进来。注意，不要超过64 KB。在短期内，在Kudu中使用二进制类型可能没有什么意义。

网络包示例

NetFlow是一种数据格式，反映了与网络路由器或交换机交互的所有网络接口的IP地址的统计信息。我们可以接近实时地生成和收集NetFlow记录，用于网络安全、网络服务质量和容量规划等目的。对于对此类数据感兴趣的网络和安全分析师而言，能够快速、接近秒级别地分析数据，意味着能更快地检测威胁和提供更高质量的网络服务。

以下示例展示了哈希分区和范围分区的组合，是如何既减少带时间范围谓词的查询读取数据的量而提高读取性能，又通过在服务器之间分散写操作来提高写入性能的：

```
CREATE TABLE netflow (
id string,
packet_timestamp string,
srcaddr string,
dstas string,
dstaddr_s string,
dstport int32,
dstaddr string,
srcaddr_s string,
tcp_flags string,
dPkts string,
```

netflow表对id字段进行了哈希分区，id字段是唯一的，应该能够让所有行均匀分布到哈希桶和集群的节点中。哈希分区为写入操作提供了高吞吐率，因为提供足够的存储桶后，所有节点都将包含至少一个哈希分区。当扫描多个ID值时，哈希分区还为读取操作提供了并行性，因为所有包含哈希分区的节点都将参与扫描。

```
tos string,
engineid string,
enginetype string,
srcas string,
packetid string,
nexthop_s string,
samplingmode string,
dst_mask string,
snmponput string,
length string,
flowseq string,
samplingint string,
readerId string,
snmpinput string,
src_mask string,
version string,
nexthop string,
uptime string,
dOctets string,
sender string,
proto string,
srcport int32)
DISTRIBUTE BY HASH (id) INTO 4 buckets,
  RANGE (packet_timestamp)
  SPLIT ROWS ( ('2015-05-01'),
               ('2015-05-02'),
               ('2015-05-03'),
               ('2015-05-05')
             );
```

该表还按时间进行了范围分区，这样在做仅扫描特定时间段的查询时就可以排除不包含相关数据的tablet。这应该会增加大扫描（跨越几天的数据）的集群并行度，同时限制小扫描（单日的数据）的开销。范围分区还能确保分区的增长是受限制的，查询不会随着表中存储的数据的增长而变慢，因为我们只查询数据的某些部分，并且数据是通过哈希分区和范围分区分布在所有节点上的。

上面的建表语句创建了16个tablet。首先，它创建4个由id字段哈希分区的桶，然后为每个哈希桶创建4个范围分区的tablet。向Kudu写入数据时，被插入的行将首先基于id字段进行哈希分区，然后基于packet_timestamp字段进行范围分区。结果就是这些写入操作将被分散到4个tablet（或服务器）上。同时，读取操作（如果限制为单日数据）将仅查询包含给定日期数据的tablet。这一点很重要，因为我们不需要花太多力气就可以做到对写操作水平扩展的同时限制对时间序列的读取的数据量。

虽然我们在这里没有展开讨论，但是记住，这个分区方案除了在分析查询和写入数据上拥有上述优点之外，在之前讲过的物联网用例中，也可以高效地查找单个事件，因为数据包的id在主键中。

小结

与所有数据库一样，模式设计是一个高性能用例成败的关键。作为分布式数据库，Kudu在模式设计时有一些需要考虑的独特属性。此外，作为最先进的数据库，它还有一些新的调优选项，比如新的列编码和压缩算法。阅读本章后，你应该能更好地理解Kudu模式设计中一些最重要的内容。

第7章 Kudu用例

实时物联网分析

早上醒来，刷牙，喝咖啡，步行到车库，上车，开车上班，虽然这些是你每天都会做的事情，但今天你感觉哪里不太对。你试图不去管它，但这种感觉一直困扰着你。你好像忘记了什么。

你的注意力从电台DJ转移到周末的计划，再到工作中要完成的所有事情。但是令人不安的怀疑不断袭来：“我到底忘记了什么？”

让人不得安宁的怀疑被手机上的警报打断了。这是你的智能手机上的车库门应用！很显然，你忘记关车库了。你遇到了电影《小鬼当家》中的情况。虽然你不是在海外度假时把孩子忘在了家里，但是你忘了关车库的门。幸运的是，现在是互联网时代，你的车库门连接着手机上的一个应用。

所以，你靠边停车，在手机上点一下按钮，就关闭了车库。这一切都发生在离家30公里的地方。

如果是几年前，你遇到这种情况的话意味着要一路驾车回家，或者打电话给邻居，或者整天不停地在想有人在偷你妻子的公路自行车和儿子的高尔夫球杆。但是今天，你避免了这样的危机。你应该感谢数据，感谢数据分析和Hadoop生态系统。

现在，让我们换个角度，把自己想象成车库门制造商。

你那一家在中西部的工厂生产世界上最好的车库门已经几十年了。你可以找到并聘请最聪明的机械工程人才来制造制动器、马达、齿轮和安全设备，这些都是成为世界级制造商所必须提供的产品。你没有预料到自己还要搭建数据产品，但时代的变化逼着你走到了这一步。在消费者预期发生变化和竞争加剧的情况下，数据和分析是驱动差异化、提升效率和促进销售的关键。

你的数据基础设施必须支持以下内容：

一个新的面向消费者的智能手机应用

防止发生上文提到的电影《小鬼当家》中的情况。

改进的客户支持

能远程查看客户设备的当前状态以及远程诊断问题。

提高质量

分析部署在现场的数百万个车库门。它们在实际操作中是如何被使用的？如何识别零件的问题或使用中的问题？

消费者营销

当你了解某个用户在一段时间内的使用情况后，如何才能更好地向他推销相关的新产品？

作为一个制造商，这是你第一次尝试建立一个数据产品，而你想要赢。因此，拥有简单、可扩展、易于维护的数据基础设施，对于项目的成功至关重要。

美国有1亿个车库门，而你占据了美国约70%的市场。车库门每隔30s就传送一次信息。这样，每年总共约有73万亿行记录。所以，你要解决可扩展性的问题，并且希望使用有成本效益的、可扩展的大数据技术来设计解决方案。这些记录中的数据包括设备标识符、报告的时间戳（数据上报的时间）、设备的状态以及各种其他信息，如故障代码和内部温度，所有这些都可以在表7-1中看到。

表7-1 列的类型

设备 ID	数据上报的时间	状态	故障代码	马达状态	旅行模式	温度传感器
583134	1492924860	0	0	11	0	22
583134	1492924890	0	0	11	0	22
583134	1492924920	1	0	11	0	23

如果是在几年前，你会使用两个不同的存储层来设计这个系统。和Lambda架构不一样，Kudu能将两个单独的“实时层”和分析存储层合并为单个数据存储层（见图7-1）。

作为一个谨慎的专注工程化的制造商，你知道用一个系统肯定比同时用两个系统更简单。采用Kudu，你的数据基础设施开发起来会更快、更容易维护、成本更低，并且更适应未来的应用场景。

在设计模式和分区策略时，就像第6章所讨论的，我们将考虑读和写两方面的工作负载。这个应用的负载包括以下特点：

- 大量的读/写操作。

● 所有查询都包含时间谓词。例如，显示过去三个月中最常出现的故障的代码，或扫描所有设备最近20分钟的数据以确定哪些车库门是开着的。

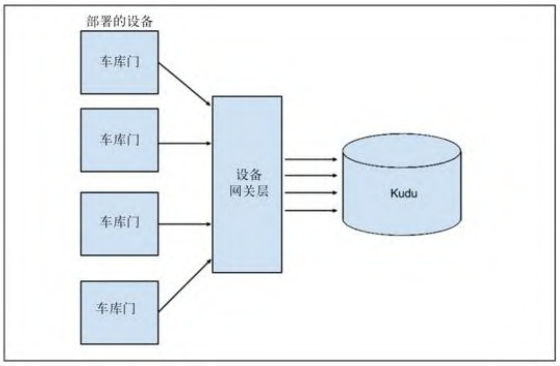

图7-1 基于Kudu的车库物联网架构图

● 许多查询包含对某个设备标识符的短扫描，客服代表和用户通常会在查询中以设备标识符为搜索目标。

因为我们的工作负载在大多数查询中使用时间和设备作为谓词，所以我们将数据上报的时间和设备ID设为主键。由于我们的写吞吐率很高，因此我们的分区策略中包含了对设备标识符进行哈希分区。如果有足够的分区，这样做可以确保写入操作被分散到许多tablet服务器上。

对于读操作负载，我们希望避免短扫描涉及太多的tablet服务器开销。例如，当客服代表或用户想了解设备的当前状态时，可能要扫描最近一小时的数据才能获得某个设备I D的数据，或者大约120行记录。对数据的上报时间进行范围分区后，行记录会按时间戳顺序分组，因而能够对短时间范围内的特定用户进行高性能的实时查找。范围分区还允许我们删除老旧的分区，解决分区没有界限的问题。

回顾之前的数据基础架构的目标，我们现在有一个单一的数据仓库，可以支持面向用户的客户服务、质量和消费者营销等各种应用的广泛需求。

预测建模

现在假设我们要从零开始做一家在线零售商。在当今市场，要做在线零售商，就必须向购物者提供相关的推荐。这里我们不会详细介

绍如何搭建一个推荐系统，这方面已经有很多著述了。重要的是，如果你现在正在搭建这样的推荐系统，很可能会用到Hadoop平台，其中的数据可能如表7-2所示。

表7-2 列的类型

用户 ID	商品 ID	得分
123	342352342	0.23434343
123	123128933	0.3439292
124	957472924	0.62329292

这种模型的输出将包含三列：用户ID、商品ID和得分。用户ID表示用户，商品ID表示商店中的商品，而得分表示的是对于给定用户喜欢此推荐商品的可能性的度量。

当你获得每个用户ID和商品ID的分数后，就可以开始进行推荐。这意味着你需要快速检索出对于给定用户而言得分最高的商品。为了让你对数据规模有基本的了解，我们来计算一下亚马逊网站数据集的大小。它目前有4.8亿件商品和3.04亿的活跃用户。这意味着我们有145，920，000，000，000，000或145万亿行记录。如果通过列式压缩，我们将行大小减少到每行4个字节，就将是583 PB的数据。这只是一个粗略的计算，还有许多技术可以减少数据的大小，但我们确实可以说推荐系统是典型的大数据系统。

我们了解了这个系统的基本情况，那么应该如何设计呢？首先，我们从在线检索端之外的批处理系统开始，这是必需的系统。这个系统需要一个批处理存储层来存储事务和页面浏览日志，以及一个批处理执行层来计算模型。我们使用HDFS或Amazon S3存储数据，然后使用Spark批处理执行。在训练模型之后，我们需要将它发布到某个地方，以便可以实时访问它。传统的做法是，将模型写入H D F S/S3，然后将其导出到关系型数据库。这是一种众所周知的方法，但其测试和运维都很复杂。我们都知道数据库导出作业失败最能破坏好心情了。该系统架构如图7-2所示。

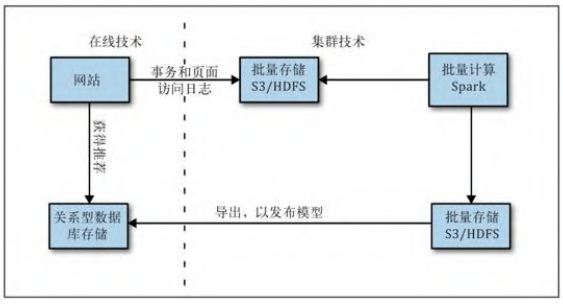

图7-2 推荐系统架构1.0

对该模型的第一项改进是，消除为了把数据导出到另一个系统所做的复杂编排。因为Kudu支持通过主键快速检索，所以我们可以去掉关系型数据库，并将数据导出到Kudu。这样可以消除流程中的活动部件，增加了稳定性。推荐系统架构2.0如图7-3所示。

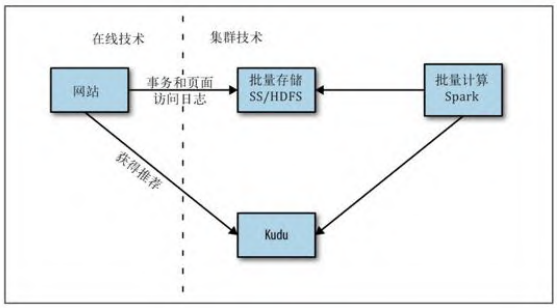

图7-3 推荐系统架构2.0

第一项改进减轻了数据运营团队的工作，第二项改进将侧重于改善业务成果以及管理层对我们团队创新能力的看法。此时，我们仍然只是在重算得分后才往系统里添加新数据，通常每天一次或每周一次。然而，多年前，研究人员就开始对在流式或实时条件下更新这些批量计算的结果展开研究。因此，现在实时更新模型是可能做到的。

为了实现这种方式，我们还需要一个系统来存储流数据。Kafka符合这个要求。这种新架构还有一个额外的好处，就是我们可以在Kudu中存储所有事务和页面查看历史记录（page-view history），并将此服务提供给网站。这样，网站团队就不需要配备自己的历史数据存储系统了。推荐系统架构3.0如图7-4所示。

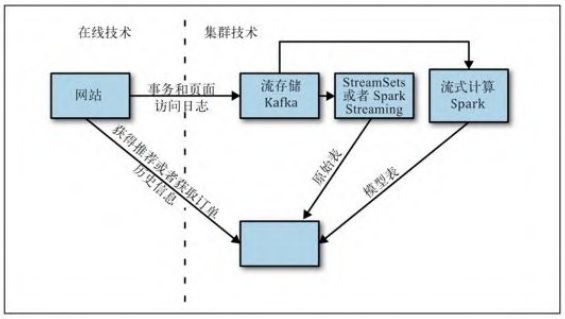

图7-4 推荐系统架构3.0

如果使用Kudu，就能够通过减少数据延迟来降低复杂性并改善我们的业务。

多平台混合方案

读到此处时，你很可能已经了解了基于Kudu的设计的好处：快速、简单、开放、结构化，等等。在本节中，我们将给出一个用例的概述，在这个用例中会结合另一个存储引擎来使用Kudu。

Kudu最大的好处之一在于，因为表是结构化的，并且都有主键，所以数据总是有序的。而对于像HDFS上的Parquet这样的存储系统来说，却并非如此。如果数据快速且无序地到达，需要对其进行分析，到目前为止Kudu就是最优的平台！然而，如果数据已经被排序，并且数据保留策略使得数据集增长到许多TB，这时把数据存储成另一种格式来降低Kudu的负载可能是有益的。

这个用例可以用于许多不同的垂直领域。例如，有大量数据来自许多不同的传感器或设备的各个组件，并且必须尽可能快地将它们聚合、排序和存储，以便立即可用于分析。你可以想象一家公共事业公司，它的电网设备会发送传感器信息，你也可以想象一家电信公司，它的天线会发送数据，或者一家石油公司，所有的配销点都发送实时的销售信息，等等。虽然它们属于不同的垂直领域，但是概念是相同的。

数据是从不同站点上的不同传感器发送过来的，所以其到达平台时可能是无序的。在过去的设计中，我们会把所有数据存储在一个临时区域中，等待所有站点发送完所有数据后，将整个数据集排序再插入表和分区。但由于Kudu是将数据按照基于主键的顺序存储的，因此无须把数据临时存储到其他地方。数据一到达，就可以把它插入到系统中。不仅如此，我们还可以没有任何迟延地检索数据。

尽管这里的表和主键设计与我们在前面章节中所看到的非常相似，但应用的设计存在差异。事实上，今天Kudu的可扩展性还达不到HDFS的水平，存储多年的历史数据时需要将“旧”的Kudu数据转存成另一种格式以降低Kudu的负载。我们下面要介绍的详细设计，其背后的思想就是利用多个存储工具的优点。这种设计既利用了Kudu非常快速的实时分析能力，又利用了HDFS几乎无限的可扩展性。

图7-5描述了数据如何在两个平台之间流动。

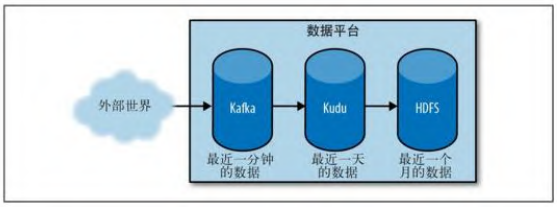

图7-5 多平台数据输入

大多数的分析请求通常是针对最近的数据的，很少会频繁地请求数月或数年的数据。频繁的请求一般是针对最近一天、最近一周、最近几周或最多最近一个月的数据的。对于数据分析而言，在几个可选项中Kudu是最快的，而且因为大部分的请求都针对上个月的数据，所以我们只在Kudu上保留最近一个月的数据。过去的31天，每24小时一次，最久远的一天的数据将从Kudu中提取出来，并导出到HDFS上一个Impala Parquet表的分区中。

所以，接下来的挑战将与我们查询数据的方式有关。因为有些数据存储在Kudu中，而另一些存储在HDFS中，所以我们需要找到一种方法来为查询引擎提供统一的视图，视图的一部分将检索最近的数据，另一部分将检索“冷”数据。当一个查询包含了最近的数据和旧数据时，我们应该能够同时访问两个存储引擎来处理这些数据。

这里，我们会利用到Impala的view功能。

因为我们的数据是按日期分区的，并且这两个平台分别存储各自日期范围的数据，所以所有要创建的表都将按此标准分区。这样的话，我们将在Kudu上创建一个按日期分区的表，并且在HDFS上也创建一个按日期分区的表。现在，为了能够联合这两个数据集，我们只需要在它们上面创建一个视图！对该数据集的所有请求只需要针对该视图执行。当仅查询最近的数据时，就只会涉及Kudu。然而，要分析更久远时间范围的数据时，这两个平台都会参与到查询中，但是对于用户是透明的。

图7-6显示了多个表如何使用不同的存储平台，以及如何联合这些表来构建视图，为终端用户服务。

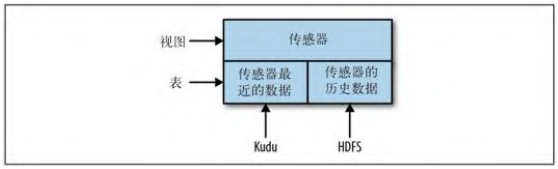

图7-6 多平台的表和视图

这种设计有很多好处：

● 它允许我们非常快速、有效地访问最近的（也是最近常被使用的）数据。

● 实现起来非常简单。它只需要一个从Kafka到Kudu的数据输入流，以及一个从Kudu转移数据到HDFS的ETL。我们可以使用非常简单的SQL请求来实现这个ETL。

● 它允许非常灵活的可扩展性。事实上，当Kudu的可扩展性增加时，更多的数据可以保存在Kudu集群中。当数据保留策略改变时，在HDFS侧做调整也是非常简单的。

关于作者

Jean-Marc Spaggiari，Kudu的早期用户，是Cloudera的首席解决方案架构师，为Hadoop、Kudu、HBase和其他工具提供技术支持及咨询服务。Jean-Marc对HBase和HDFS有深入的了解，这也使他能够更好地理解Kudu及其应用。

Jean-Marc的主要工作是支持HBase用户部署、升级、配置和优化HBase集群，以及支持他们进行与HBase相关的应用开发。他还是一位非常活跃的HBase社区成员，从性能和稳定性的角度对每一个发行版本进行测试。随着Kudu迅速地渗透进市场，Jean-Marc也开始推荐Kudu，搭建demo应用，部署基于Kudu的概念验证原型。

在加入Cloudera之前，Jean-Marc曾担任CGI和保险公司的项目经理及解决方案架构师。他有近20年的Java开发经验。除了定期参加Strata+Hadoop World和HBaseCon会议，他还在北美的各种Hadoop用户组会议和许多大会上发言，主要是HBase相关的演讲和演示。Jean-Marc也是Architecting HBase Applications（O'Reilly，2016）一书的作者。

Mladen Kovacevic有RDBMS技术的开发背景，将Kudu视为Hadoop生态系统中的变革者。他在几次当地的聚会上介绍了Kudu，介绍Kudu的beta版本对Spark的支持状态，并且提供了非常早的反馈，使得Kudu在发布时Spark对其有较好的支持。Mladen是Apache Kudu和Kite SDK项目的贡献者，在Cloudera担任解决方案架构师。Mladen拥有很多开发经验，他做过多年的RDBMS引擎开发、系统优化、性能和体系结构方面的工作，开发IBM的Big SQL技术时他在Power 8平台上优化过Hadoop。

Brock Noland一直在看Todd Lipcon的论文，他在Todd正式编写Kudu的第一行代码之前的几个月就开始关注Kudu了。Brock是phData的首席架构师，该公司专门提供Hadoop的托管服务。创建phData之前，Brock曾在Cloudera做了四年的培训师、解决方案架构师、工程师、销售工程师和工程经理。Brock是Apache Sentry的共同创始人，还是Apache Hive、Parquet、Crunch、Flume和Incubator的PMC成员。

Brock是Kudu在孵化器阶段的导师，并且目前是Apache Imapala（孵化阶段）的导师。此外，他还是Apache软件基金会的成员。

Brock经常在公共场合演讲，曾出席过很多大会，包括HBaseCon、Hadoop User Group和其他大会。

Ryan Bosshart是phDdata的首席执行官，也是Cloudera的前首席系统工程师。Ryan在过去10年中一直在搭建和设计分布式系统。在Cloudera，Ryan领导着领域存储专业化团队，主要关注HDFS、HBase和Kudu。他曾与许多Kudu早期用户合作，为他们搭建关系型架构、时间序列架构、物联网或实时架构。他见证了Kudu提高性能和简化架构的能力。Ryan是Spark和Hadoop双城用户组的联席主席，也是培训视频Getting Started with Kudu（O’Reilly）的作者。

封面图片

本书封面上的动物是扭角林羚（tragelaphus strepsiceros），非洲东部和南部的林地、山丘和山脉中都有扭角林羚，它是最大的羚羊物种之一。

扭角林羚的身躯比较窄，腿长，通体深灰色或棕色，两边有白色的垂直条纹，两眼之间有一条V形白色条纹。雄性扭角林羚的皮毛比雌性的厚，沿着脖子前部生长着短鬃毛，还有螺旋形的角，盘绕两圈半后向上延伸到一个点。

虽然雄性扭角林羚是独居的，但是雌性扭角林羚通常以6至10只为群体进行活动，有人见过一些较大的扭角林羚群。扭角林羚的交配发生在雨季结束时，雌性在8个月后生产。在幼崽出生后的第一个月，母亲把它藏在高高的草丛和其他植物中，但是幼崽成熟得很快，在6个月大的时候就可以独立了。

在过去10年中，尽管由于水井和灌溉系统的出现，一些以前太干燥、扭角林羚无法栖息的地区现在变得适合它们生存，但是由于狩猎、疾病和栖息地的丧失等原因，全球扭角林羚的数量略有减少。偷猎者垂涎它们独特的角，经常用其来制作羊角号（犹太人的祭祀中会用到，在犹太新年期间吹响）。

O’Reilly图书封面上的许多动物都濒临灭绝，它们对世界很重要。要了解更多关于如何帮助它们的信息，请访问animals.oreilly.com。

封面的图片来自Meyers Kleines Lexicon。封面字体是URW Typewriter和Guardian Sans。正文字体是Adobe Minion Pro，标题字体是Adobe Myriad Condensed，代码字体是Dalton Maag的Ubuntu Mono。

反侵权盗版声明