

O'REILLY®

Hive 编程指南

[美] Edward Capriolo

Dean Wampler

Jason Rutherglen 著

曹 坤 译

人 民 邮 电 出 版 社

北 京

图书在版编目 (C I P) 数据

Hive编程指南 / (美) 卡普廖洛 (Capriolo, E.),
(美) 万普勒 (Wampler, D.), (美) 卢森格林
(Rutherglen, J.) 著; 曹坤译. — 北京: 人民邮电出
版社, 2013. 12

ISBN 978-7-115-33383-4

I. ①H… II. ①卡… ②万… ③卢… ④曹… III. ①
数据库系统—程序设计—指南 IV. ①TP311.13-62

中国版本图书馆CIP数据核字(2013)第238550号

版权声明

Copyright © 2013 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2013. Authorized translation of the English edition, 2013 O'Reilly Media, Inc., the owner of all rights to publish and sell same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体字版由 **O'Reilly Media, Inc.** 授权人民邮电出版社出版。未经出版者书面许可, 对本书的任何部分不得以任何方式复制或抄袭。

版权所有, 侵权必究。

◆ 著 [美] Edward Capriolo Dean Wampler

Jason Rutherglen

译 曹 坤

责任编辑 汪 振

责任印制 程彦红 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

三河市海波印务有限公司印刷

◆ 开本: 787×1000 1/16

印张: 21.25

字数: 428 千字

2013 年 12 月第 1 版

印数: 1-3 000 册

2013 年 12 月河北第 1 次印刷

著作权合同登记号 图字: 01-2013-2654 号

定价: 69.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

内容提要

本书是一本 Apache Hive 的编程指南,旨在介绍如何使用 Hive 的 SQL 方法——HiveQL 来汇总、查询和分析存储在 Hadoop 分布式文件系统上的大数据集合。全书通过大量的实例,首先介绍如何在用户环境下安装和配置 Hive,并对 Hadoop 和 MapReduce 进行详尽阐述,最终演示 Hive 如何在 Hadoop 生态系统进行工作。

本书适合对大数据感兴趣的爱好者以及正在使用 Hadoop 系统的数据库管理员阅读使用。

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，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

作者简介

Edward Capriolo 目前是 Media6degrees 公司的系统管理员，在这里他为互联网广告企业提供设计和维护分布式数据存储系统的服务。

Edward 是 Apache 软件基金会的成员，同时也是 Hadoop/Hive 项目的贡献者。Edward 具有软件开发经验，同时也具有 Linux 和网络管理员的经历，而且对于开源软件世界充满了热情。

Dean Wampler 是 ThinkBigAnalytics 公司的首席顾问，其擅长“大数据”文件，以及像 Hadoop 这样的工具研究，还有机器学习相关的内容。除了擅长大数据，他还擅长 Scala、JVM 生态系统、JavaScript、Ruby、函数式以及面向对象编程，同时还擅长敏捷方法。Dean 经常性地工业和学术会议上就这些主题进行演讲。他还具有来自华盛顿大学的物理学博士学位。

Jason Rutherglen 是 Think Big Analytics 公司的一名软件架构师，其擅长大数据、Hadoop、搜索和安全领域。

作者序

Edward Capriolo

当我第一次参与到 Hadoop 里时，我看到了分布式文件系统和 MapReduce 计算框架可以以一种伟大的方式来解决计算密集型的问题。然而，使用 MapReduce 编程模型进行编程曾经对于我来说是件非常麻烦的事情。Hive 提供了一个类 SQL 的方式可以让我快速而又简单地利用到 MapReduce 计算的优势。这种方法也使得概念验证应用程序原型设计变得容易，同时在内部可以很好地使用 Hadoop 作为解决方案。尽管我现在非常熟悉 Hadoop 内核，Hive 仍然是我利用 Hadoop 进行工作的主要方法。

能够参与编写一本关于 Hive 的书，对我来说是一件非常荣耀的事情；同时能够作为一名 Hive 代码贡献者和 Apache 软件基金会的成员也是我最有价值的荣誉。

Dean Wampler

作为 Think Big Analytics 公司的一名“大数据”顾问，我经常和一群具有丰富经验的 SQL “数据人”一起工作。对他们来说，使用 Hive 是必要且充分的，这样才能使用 Hadoop 作为可行的工具，并利用他们的 SQL 知识来使用数据分析，开创新的机遇。

Hive 缺乏良好的文档。我向 O'Reilly 出版社的编辑 Mike Loukides 建议，社区确实需要一本 Hive 相关的书籍。于是，本书应运而生……

Jason Rutherglen

我是 Think Big Analytics 公司的一名软件架构师。我的职业生涯涉及一系列的技术，包括搜索、Hadoop、移动、密码学和自然语言处理。Hive 是使用开源技术，基于海量数据构建数据仓库的最终方式。我在很多不同的项目中使用了 Hive。

致 谢

感谢参与到 Hive 中的每一个人。包括代码贡献者、参与者以及最终用户。

Mark Grove 编写了 Hive 和亚马逊网络服务那一章的内容。他是一个 Apache Hive 项目的贡献者并在 Hive IRC 上非常积极地帮助他人。M6D 公司的 David Ha 和 Runit Pate 贡献了案例研究章节的内容和等级函数的代码。在 Hive 中可以进行排名是一个重要的特性。

M6D 公司的 Stitelman，贡献了案例研究章节中关于数据科学如何使用 Hive 和 R 的内容，其中演示了如何通过 Hive 对大数据集进行一次处理并提供了产生的结果，然后在之后的处理过程中使用 R 处理 Hive 产生的结果数据。

David Funk 贡献了 3 个用例，即：站内引用链接识别、会话化、计数独立用户访问量。David 的技术说明展示了如何重写和优化 Hive 查询可以使数据分析效率得到大幅度提高。Ian Robertson 审阅了整个书的初稿并提供非常有用的反馈信息。我们非常感谢他，在时间很紧的短时间内提供了这些反馈。

John Sichi 对本书进行了专业技术评审。John 同时也帮助开发了 Hive 中的一些新特性，例如 StorageHandlers 和索引支持。他一直积极帮助支持 Hive 社区的成长。

Alan Gates,《Pig 编程指南》的作者,贡献了关于 HCatalog 的那一章内容。Nanda Vijaydev 贡献了关于 Karmasphere 公司如何将 Hive 进行增强并提供产品化的那一章内容。Eric Lubow 提供了关于 SimpleReach 公司的案例研究。Chris A. Mattmann、Paul Zimdars、Cameron Goodale、Andrew F. Hart、Jinwon Kim、Duane Waliser 和 Peter Lean 共同贡献了美国宇航局喷气推进实验室 (NASA JPL) 的案例研究。

1. The first part of the document discusses the importance of maintaining accurate records of all transactions and activities. It emphasizes that proper record-keeping is essential for ensuring transparency and accountability in financial operations.

2. The second part of the document outlines the various methods and techniques used to collect and analyze data. It highlights the need for consistent and reliable data collection processes to ensure the validity of the results.

3. The third part of the document describes the different types of data that are collected and analyzed. It includes information on both quantitative and qualitative data, as well as the specific variables being measured.

4. The fourth part of the document discusses the various statistical methods and techniques used to analyze the data. It covers topics such as descriptive statistics, inferential statistics, and regression analysis.

5. The fifth part of the document discusses the various methods and techniques used to collect and analyze data. It highlights the need for consistent and reliable data collection processes to ensure the validity of the results.

6. The sixth part of the document discusses the various methods and techniques used to collect and analyze data. It highlights the need for consistent and reliable data collection processes to ensure the validity of the results.

前言

本书是一本Hive的编程指南。Hive是Hadoop生态系统中必不可少的一个工具，它提供了一种SQL（结构化查询语言）方言，可以查询存储在Hadoop分布式文件系统（HDFS）中的数据或其他和Hadoop集成的文件系统，如MapR-FS、Amazon的S3和像HBase（Hadoop数据库）和Cassandra这样的数据库中的数据。

大多数数据仓库应用程序都是使用关系数据库进行实现的，并使用SQL作为查询语言。Hive降低了将这些应用程序转移到Hadoop系统上的难度。凡是会使用SQL语言的开发人员都可以很轻松地学习并使用Hive。如果没有Hive，那么这些用户就必须学习新的语言和工具，然后才能应用到生产环境中。另外，相比其他工具，Hive更便于开发人员将基于SQL的应用程序转移到Hadoop中。如果没有Hive，那么开发者将面临一个艰巨的挑战，如何将他们的SQL应用程序移植到Hadoop上。

不过，Hive和其他基于SQL的环境还是有一些差异的。如今，可供Hive用户和Hadoop开发者使用的文档并不多，所以我们决定撰写这本书来填补这个缺口。我们将对Hive进行全面详实的介绍，主要适用于SQL专家，如数据库设计人员和业务分析师。我们也谈到了深入的技术细节，可以帮助Hadoop开发人员对Hive进行调优和定制。

用户可以在本书的目录页面了解到更多信息：http://oreil.ly/Programming_Hive。

本书中所使用的约定

本书中使用到了如下几种印刷字体。

斜体字

表明是新的术语、URL、电子邮件地址、文件名或者文件扩展名。

等宽字体

用于程序列表，同时段落中使用到的了程序片段，例如变量或者函数名称、数据库、数据类型、环境变量、语句和关键字。

等宽粗体

表示是命令或者其他需要用户进行输入的文本。

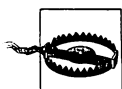
等宽斜体

表示这个文本需要用户提供对应的值或者需要通过上下文才能获取到的值。



提示

这个图标表明是一个小技巧、建议或者一般性的注释。



警告

这个图标表明是个警告或者警示。

使用的代码示例

本书的目的是帮助用户完成他们的任务。通常情况下，用户可以在他们的程序和文档中使用本书中的代码。如此不需要联系我们以获取许可，除非明显地复制了代码的大部分内容。例如，写程序用到了本书中几个代码片段是不需要获得许可的，但是如果销售或者传播包含了O'Reilly系列书籍中的例子的CD光盘，那么就一定要获得我们的许可才行。在回答问题时引用到本书或以本书中的例子为引证时不需要获得许可，将一定数量的样例代码复制到自己的产品文档中则一定需要获得我们的许可才可以。

虽然并非是必需的，但如果可以注明出处，我们将十分感激。出处一般包括标题，作者，出版商和ISBN。例如：“Programming Hive by Edward Capriolo, Dean Wampler, and Jason Rutherglen (O'Reilly). Copyright 2012 Edward Capriolo, Aspect Research Associates, and Jason Rutherglen, 978-1-449-31933-5.”

如果用户感觉自己没有合理地或者在如上所述的许可范围内使用本书中代码样例的话，请尽管通过 permissions@oreilly.com联系我们。

Safari®在线图书



Safari在线图书是一个按需服务的数字图书馆。使用它，用户可以轻松检索超过7 500本技术和创意参考书以及视频教程，快速获得想知道的答案。

通过订阅，用户可以从我们的在线图书馆中阅读每一篇文章或观看每一部视频。通过用户的手机和其他移动设备看书。在书还没有印刷前就可以事先看到书目，还可以看到正在进行中的草稿，并可以将意见反馈给作者。复制粘贴代码样例，组织用户的收藏夹，下载一些章节，对关键章节标记标签，创建笔记，打印书籍内容，并通过其他众多的省时功能而受益。

O'Reilly公司已经将本书上传到 Safari图书在线服务。想获得对这本书的完整数据版访问权限以及其他的来源于O'Reilly和其他出版商的对相同话题的讨论，请通过网址 <http://my.safaribooksonline.com>免费注册账户。

如何联系到我们

请将对于本书的评论和问题通过如下地址发送给出版商：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

在本书的官方页面中，我们列举了勘误表、例子和其他附加信息，用户可以通过以下链接访问：

http://oreil.ly/Programming_Hive

可以通过如下E-mail，将用户对本书的评论或技术方面的问题发送给我们：

bookquestions@oreilly.com

想获得更多关于我们的系列书籍、会议、资源中心和O'Reilly网络公司，请通过如下网站查看：

<http://www.oreilly.com>

我们的Facebook地址是：<http://facebook.com/oreilly>

我们的Twitter地址是：<http://twitter.com/oreillymedia>

我们的YouTube地址是：<http://www.youtube.com/oreillymedia>

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

目录

第 1 章	基础知识	1
1.1	Hadoop 和 MapReduce 综述	3
1.2	Hadoop 生态系统中的 Hive	6
1.2.1	Pig	8
1.2.2	HBase	8
1.2.3	Cascading、Crunch 及其他	9
1.3	Java 和 Hive：词频统计算法	11
1.4	后续事情	13
第 2 章	基础操作	14
2.1	安装预先配置好的虚拟机	14
2.2	安装详细步骤	15
2.2.1	装 Java	16
2.2.2	安装 Hadoop	17
2.2.3	本地模式、伪分布式模式和分布式模式	18
2.2.4	测试 Hadoop	19
2.2.5	安装 Hive	21
2.3	Hive 内部是什么	22
2.4	启动 Hive	22
2.5	配置 Hadoop 环境	24
2.5.1	本地模式配置	24
2.5.2	分布式模式和伪分布式模式配置	26
2.5.3	使用 JDBC 连接元数据	27
2.6	Hive 命令	29
2.7	命令行界面	30
2.7.1	CLI 选项	30
2.7.2	变量和属性	31
2.7.3	Hive 中“一次使用”命令	34
2.7.4	从文件中执行 Hive 查询	35
2.7.5	hiverc 文件	36

2.7.6	使用 Hive CLI 的更多介绍.....	36
2.7.7	查看操作命令历史.....	37
2.7.8	执行 shell 命令.....	37
2.7.9	在 Hive 内使用 Hadoop 的 dfs 命令.....	38
2.7.10	Hive 脚本中如何进行注释.....	38
2.7.11	显示字段名称.....	38
第 3 章	数据类型和文件格式.....	40
3.1	基本数据类型.....	40
3.2	集合数据类型.....	42
3.3	文本文件数据编码.....	44
3.4	读时模式.....	47
第 4 章	HiveQL: 数据定义.....	48
4.1	Hive 中的数据库.....	48
4.2	修改数据库.....	52
4.3	创建表.....	52
4.3.1	管理表.....	56
4.3.2	外部表.....	56
4.4	分区表、管理表.....	57
4.4.1	外部分区表.....	61
4.4.2	自定义表的存储格式.....	63
4.5	删除表.....	66
4.6	修改表.....	66
4.6.1	表重命名.....	67
4.6.2	增加、修改和删除表分区.....	67
4.6.3	修改列信息.....	68
4.6.4	增加列.....	68
4.6.5	删除或者替换列.....	68
4.6.6	修改表属性.....	69
4.6.7	修改存储属性.....	69
4.6.8	众多的修改表语句.....	70
第 5 章	HiveQL: 数据操作.....	71
5.1	向管理表中装载数据.....	71
5.2	通过查询语句向表中插入数据.....	73
5.3	单个查询语句中创建表并加载数据.....	76
5.4	导出数据.....	76
第 6 章	HiveQL: 查询.....	78
6.1	SELECT... FROM 语句.....	78

6.1.1	使用正则表达式来指定列.....	80
6.1.2	使用列值进行计算.....	81
6.1.3	算术运算符.....	81
6.1.4	使用函数.....	82
6.1.5	LIMIT 语句.....	93
6.1.6	列别名.....	93
6.1.7	嵌套 SELECT 语句.....	93
6.1.8	CASE ... WHEN ... THEN 句式.....	93
6.1.9	什么情况下 Hive 可以避免进行 MapReduce.....	94
6.2	WHERE 语句.....	95
6.2.1	谓词操作符.....	96
6.2.2	关于浮点数比较.....	97
6.2.3	LIKE 和 RLIKE.....	98
6.3	GROUP BY 语句.....	99
6.4	JOIN 语句.....	100
6.4.1	INNER JOIN.....	100
6.4.2	JOIN 优化.....	103
6.4.3	LEFT OUTER JOIN.....	104
6.4.4	OUTER JOIN.....	104
6.4.5	RIGHT OUTER JOIN.....	106
6.4.6	FULL OUTER JOIN.....	107
6.4.7	LEFT SEMI-JOIN.....	107
6.4.8	笛卡尔积 JOIN.....	108
6.4.9	map-side JOIN.....	108
6.5	ORDER BY 和 SORT BY.....	110
6.6	含有 SORT BY 的 DISTRIBUTE BY.....	111
6.7	CLUSTER BY.....	112
6.8	类型转换.....	112
6.9	抽样查询.....	113
6.9.1	数据块抽样.....	114
6.9.2	分桶表的输入裁剪.....	114
6.10	UNION ALL.....	115
第 7 章	HiveQL: 视图.....	117
7.1	使用视图来降低查询复杂度.....	117
7.2	使用视图来限制基于条件过滤的数据.....	118
7.3	动态分区中的视图和 map 类型.....	118
7.4	视图零零碎碎相关的事情.....	119

第 8 章	HiveQL: 索引	122
8.1	创建索引.....	122
8.2	重建索引.....	124
8.3	显示索引.....	124
8.4	删除索引.....	124
8.5	实现一个定制化的索引处理器.....	125
第 9 章	模式设计	126
9.1	按天划分的表.....	126
9.2	关于分区.....	127
9.3	唯一键和标准化.....	128
9.4	同一份数据多种处理.....	129
9.5	对于每个表的分区.....	130
9.6	分桶表数据存储.....	131
9.7	为表增加列.....	132
9.8	使用列存储表.....	133
9.8.1	重复数据.....	133
9.8.2	多列.....	133
9.9	(几乎)总是使用压缩.....	134
第 10 章	调优	135
10.1	使用 EXPLAIN.....	135
10.2	EXPLAIN EXTENDED.....	138
10.3	限制调整.....	139
10.4	JOIN 优化.....	140
10.5	本地模式.....	140
10.6	并行执行.....	141
10.7	严格模式.....	141
10.8	调整 mapper 和 reducer 个数.....	142
10.9	JVM 重用.....	144
10.10	索引.....	145
10.11	动态分区调整.....	145
10.12	推测执行.....	146
10.13	单个 MapReduce 中多个 GROUP BY.....	147
10.14	虚拟列.....	147
第 11 章	其他文件格式和压缩方法	149
11.1	确定安装编解码器.....	149
11.2	选择一种压缩编/解码器.....	150
11.3	开启中间压缩.....	151

11.4	最终输出结果压缩	152
11.5	sequence file 存储格式.....	152
11.6	使用压缩实践	153
11.7	存档分区	157
11.8	压缩：包扎	159
第 12 章	开发.....	160
12.1	修改 Log4J 属性	160
12.2	连接 Java 调试器到 Hive	161
12.3	从源码编译 Hive.....	161
12.3.1	执行 Hive 测试用例.....	162
12.3.2	执行 hook	163
12.4	配置 Hive 和 Eclipse.....	163
12.5	Maven 工程中使用 Hive	164
12.6	Hive 中使用 hive_test 进行单元测试	165
12.7	新增的插件开发工具箱 (PDK)	167
第 13 章	函数.....	168
13.1	发现和描述函数	168
13.2	调用函数	169
13.3	标准函数	169
13.4	聚合函数	169
13.5	表生成函数	170
13.6	一个通过日期计算其星座的 UDF	171
13.7	UDF 与 GenericUDF.....	174
13.8	不变函数	177
13.9	用户自定义聚合函数	177
13.10	用户自定义表生成函数	183
13.10.1	可以产生多行数据的 UDTF	183
13.10.2	可以产生具有多个字段的单行数据的 UDTF	185
13.10.3	可以模拟复杂数据类型的 UDTF	185
13.11	在 UDF 中访问分布式缓存.....	188
13.12	以函数的方式使用注解	190
13.12.1	确定性 (deterministic) 标注.....	191
13.12.2	状态性 (stateful) 标注.....	191
13.12.3	唯一性	191
13.13	宏命令	191
第 14 章	Streaming.....	193
14.1	恒等变换	194

14.2	改变类型.....	194
14.3	投影变换.....	194
14.4	操作转换.....	195
14.5	使用分布式内存.....	195
14.6	由一行产生多行.....	196
14.7	使用 streaming 进行聚合计算.....	197
14.8	CLUSTER BY、DISTRIBUTE BY、SORT BY.....	198
14.9	GenericMR Tools for Streaming to Java.....	201
14.10	计算 cogroup	203
第 15 章	自定义 Hive 文件和记录格式.....	204
15.1	文件和记录格式.....	204
15.2	阐明 CREATE TABLE 句式.....	204
15.3	文件格式.....	206
15.3.1	SequenceFile	207
15.3.2	RCfile	207
15.3.3	示例自定义输入格式: DualInputFormat	208
15.4	记录格式: SerDe.....	210
15.5	CSV 和 TSV SerDe	211
15.6	ObjectInspector	212
15.7	Thing Big Hive Reflection ObjectInspector	212
15.8	XML UDF.....	212
15.9	XPath 相关的函数.....	213
15.10	JSON SerDe.....	214
15.11	Avro Hive SerDe	215
15.11.1	使用表属性信息定义 Avro Schema.....	215
15.11.2	从指定 URL 中定义 Schema	216
15.11.3	进化的模式.....	216
15.12	二进制输出.....	217
第 16 章	Hive 的 Thrift 服务.....	218
16.1	启动 Thrift Server.....	218
16.2	配置 Groovy 使用 HiveServer	219
16.3	连接到 HiveServer	219
16.4	获取集群状态信息.....	220
16.5	结果集模式.....	220
16.6	获取结果.....	220
16.7	获取执行计划.....	221
16.8	元数据存储方法.....	221

16.9	管理 HiveServer	222
16.9.1	生产环境使用 HiveServer	223
16.9.2	清理	224
16.10	Hive ThriftMetastore	224
16.10.1	ThriftMetastore 配置	224
16.10.2	客户端配置	224
第 17 章	存储处理程序和 NoSQL	226
17.1	Storage Handler Background.....	226
17.2	HiveStorageHandler.....	227
17.3	HBase.....	227
17.4	Cassandra.....	229
17.4.1	静态列映射 (Static Column Mapping)	229
17.4.2	为动态列转置列映射.....	229
17.4.3	Cassandra SerDe Properties.....	229
17.5	DynamoDB.....	230
第 18 章	安全	232
18.1	和 Hadoop 安全功能相结合.....	233
18.2	使用 Hive 进行验证.....	233
18.3	Hive 中的权限管理.....	234
18.3.1	用户、组和角色	235
18.3.2	Grant 和 Revoke 权限	236
18.4	分区级别的权限	238
18.5	自动授权	239
第 19 章	锁	241
19.1	Hive 结合 Zookeeper 支持锁功能.....	241
19.2	显式锁和独占锁	244
第 20 章	Hive 和 Oozie 整合	245
20.1	Oozie 提供的多种动作 (Action)	245
20.2	一个只包含两个查询过程的工作流示例.....	247
20.3	Oozie 网页控制台	248
20.4	工作流中的变量	248
20.5	获取输出	249
20.6	获取输出到变量	250
第 21 章	Hive 和亚马逊网络服务系统 (AWS)	251
21.1	为什么要弹性 MapReduce	251
21.2	实例	251
21.3	开始前的注意事项	252

21.4	管理自有 EMR Hive 集群	252
21.5	EMR Hive 上的 Thrift Server 服务	253
21.6	EMR 上的实例组	253
21.7	配置 EMR 集群	254
21.7.1	部署 hive-site.xml 文件	254
21.7.2	部署.hiverc 脚本	255
21.7.3	建立一个内存密集型配置	255
21.8	EMR 上的持久层和元数据存储	256
21.9	EMR 集群上的 HDFS 和 S3	257
21.10	在 S3 上部署资源、配置和辅助程序脚本	258
21.11	S3 上的日志	258
21.12	现买现卖	258
21.13	安全组	260
21.14	EMR 和 EC2 以及 Apache Hive 的比较	260
21.15	包装	261
第 22 章	HCatalog	262
22.1	介绍	262
22.2	MapReduce	263
22.2.1	读数据	263
22.2.2	写数据	265
22.3	命令行	268
22.4	安全模型	269
22.5	架构	270
第 23 章	案例研究	271
23.1	m6d.com(Media6Degrees)	271
23.1.1	M 6D 的数据科学, 使用 Hive 和 R	271
23.1.2	M6D UDF 伪随机	275
23.1.3	M6D 如何管理多 MapReduce 集群间的 Hive 数据访问	280
23.2	Outbrain	284
23.2.1	站内线上身份识别	284
23.2.2	计算复杂度	287
23.2.3	会话化	288
23.3	NASA 喷气推进实验室	295
23.3.1	区域气候模型评价系统	295
23.3.2	我们的经验: 为什么使用 Hive	297
23.3.3	解决这些问题我们所面临的挑战	298
23.4	Photobucket	299

23.4.1	Photobucket 公司的大数据应用情况.....	299
23.4.2	Hive 所使用的硬件资源信息.....	300
23.4.3	Hive 提供了什么.....	300
23.4.4	Hive 支持的用户有哪些.....	300
23.5	SimpleReach.....	300
23.6	Experiences and Needs from the Customer Trenches.....	303
23.6.1	介绍.....	303
23.6.2	Customer Trenches 的用例.....	304
	术语词汇表.....	312

基础知识

从早期的互联网主流大爆发开始，主要的搜索引擎公司和电子商务公司就一直在和不断增长的数据进行较量。最近，社交网站也遇到了同样的问题。如今，许多组织已经意识到他们所收集的数据是让他们了解他们的用户，提高业务在市场上的表现以及提高基础架构效率的一个宝贵的资源。

Hadoop 生态系统就是为处理如此大数据集而产生的一个合乎成本效益的解决方案。*Hadoop* 实现了一个特别的计算模型，也就是 *MapReduce*，其可以将计算任务分割成多个处理单元然后分散到一群家用的或服务器级别的硬件机器上，从而降低成本并提供水平可伸缩性。这个计算模型的下面是一个被称为 *Hadoop 分布式文件系统 (HDFS)* 的分布式文件系统。这个文件系统是“可插拔的”，而且现在已经出现了几个商用的和开源的替代方案。

不过，仍然存在一个挑战，那就是用户如何从一个现有的数据基础架构转移到 *Hadoop* 上，而这个基础架构是基于传统关系型数据库和 *结构化查询语句 (SQL)* 的。对于大量的 *SQL* 用户（包括专业数据库设计师和管理员，也包括那些使用 *SQL* 从数据仓库中抽取信息的临时用户）来说，这个问题又将如何解决呢？

这就是 *Hive* 出现的原因。*Hive* 提供了一个被称为 *Hive 查询语言*（简称 *HiveQL* 或 *HQL*）的 *SQL* 方言，来查询存储在 *Hadoop* 集群中的数据。

SQL 知识分布广泛的一个原因是：它是一个可以有效地、合理地且直观地组织和使用数据的模型。即使对于经验丰富的 *Java* 开发工程师来说，将这些常见的数据运算对应到底层的 *MapReduce Java API* 也是令人畏缩的。*Hive* 可以帮助用户来做这些苦活，这样用户就可以集中精力关注于查询本身了。*Hive* 可以将大多数的查询转换为 *MapReduce* 任务 (job)，进而在介绍一个令人熟悉的 *SQL* 抽象的同时，拓宽 *Hadoop* 的可扩展性。如果用户对此存在疑惑，请参考稍后部分的第 1.3 节“*Java* 和 *Hive*：词频统计算法”中的相关介绍。

Hive 最适合于数据仓库应用程序，使用该应用程序进行相关的静态数据分析，不需要快速响应给出结果，而且数据本身不会频繁变化。

Hive 不是一个完整的数据库。Hadoop 以及 HDFS 的设计本身约束和局限性地限制了 Hive 所能胜任的工作。其中最大的限制就是 Hive 不支持记录级别的更新、插入或者删除操作。但是用户可以通过查询生成新表或者将查询结果导入到文件中。同时，因为 Hadoop 是一个面向批处理的系统，而 MapReduce 任务 (job) 的启动过程需要消耗较长的时间，所以 Hive 查询延时比较严重。传统数据库中在秒级别可以完成的查询，在 Hive 中，即使数据集相对较小，往往也需要执行更长的时间¹。最后需要说明的是，Hive 不支持事务。

因此，Hive 不支持 OLTP (联机事务处理) 所需的关键功能，而更接近成为一个 OLAP (联机分析技术) 工具。但是我们将会看到，由于 Hadoop 本身的时间开销很大，并且 Hadoop 所被设计用来处理的数据规模非常大，因此提交查询和返回结果是可能具有非常大的延时的，所以 Hive 并没有满足 OLAP 中的“联机”部分，至少目前并没有满足。

如果用户需要对大规模数据使用 OLTP 功能的话，那么应该选择使用一个 NoSQL 数据库，例如，和 Hadoop 结合使用的 HBase² 及 Cassandra³。如果用户使用的是 Amazon 弹性 MapReduce 计算系统 (EMR) 或者弹性计算云服务 (EC2) 的话，也可以使用 DynamoDB⁴。用户甚至可以和这些数据库 (还包括其他一些数据库) 结合来使用 Hive，这个我们会在第 17 章进行介绍。

因此，Hive 是最适合数据仓库应用程序的，其可以维护海量数据，而且可以对数据进行挖掘，然后形成意见和报告等。

因为大多数的数据仓库应用程序是使用基于 SQL 的关系型数据库实现的，所以 Hive 降低了将这些应用程序移植到 Hadoop 上的障碍。用户如果懂得 SQL，那么学习使用 Hive 将会很容易。如果没有 Hive，那么这些用户就需要去重新学习新的语言和新的工具后才能进行生产。

同样地，相对于其他 Hadoop 语言和工具来说，Hive 也使得开发者将基于 SQL 的应用程序移植到 Hadoop 变得更加容易。

不过，和大多数 SQL 方言一样，HiveQL 并不符合 ANSI SQL 标准，其和 Oracle, MySQL,

1. 不过，因为 Hive 是被设计用来处理的大数据集的，这个启动所消耗的时间和实际数据处理时间相比是微乎其微的。

2. 请访问 Apache HBase 的官方网站，<http://hbase.apache.org>，以及 Lars George(O' Reilly)所著的《HBase 权威指南》一书。

3. 请参考 Cassandra 的官方网站，<http://cassandra.apache.org/>，以及参考 Edward Capriolo (Packt)所著的《High Performance Cassandra Cookbook》一书。

4. 请参考 DynamoDB 的官方网站，<http://aws.amazon.com/dynamodb/>。

SQL Server 支持的常规 SQL 方言在很多方面存在差异（不过，HiveQL 和 MySQL 提供的 SQL 方言最接近）。

因此，本书共有两个目的。其一，本书提供了一个针对所有用户的介绍。这个介绍会比较综合，并且会使用例子来进行讲解。适用的用户包括开发者、数据库管理员和架构师，以及其他（如商业分析师等）非技术类用户。

其二，本书针对开发者和 Hadoop 管理员等需要深入了解 Hive 技术细节的用户提供了更详尽的讲述，以帮助这些用户学习如何优化 Hive 查询性能，如何通过用户自定义函数和自定义数据格式等，来个性化使用 Hive。

因为 Hive 缺少好的文档，所以我们经历了不少的挫折才完成了这本书。特别是对于那些非开发者以及不习惯通过查看项目 BUG 记录和功能数据库、源代码等途径来获取其所需信息的用户，Hive 并没有提供好的文档。Hive Wiki¹提供的信息价值很大，但是其中的解释有时太少了，而且常常没有进行及时的更新。我们希望本书可以弥补这些不足，可以提供一个对于 Hive 的所有基本功能以及如何高效使用这些功能的综合性的指南²。

1.1 Hadoop 和 MapReduce 综述

如果用户已经熟悉 Hadoop 和 MapReduce 计算模型的话，那么可以跳过本节。虽然用户无需精通 MapReduce 就可以使用 Hive，但是理解 MapReduce 的基本原理将有助于用户了解 Hive 在底层是如何运作的，以及了解如何才能更高效地使用 Hive。

我们在这里提供了一个关于 Hadoop 和 MapReduce 的简要描述。更多细节，请参考 Tom White (O'Reilly) 所著的《Hadoop 权威指南》一书。

MapReduce

MapReduce 是一种计算模型，该模型可将大型数据处理任务分解成很多单个的、可以在服务器集群中并行执行的任务。这些任务的计算结果可以合并在一起来计算最终的结果。

MapReduce 编程模型是由谷歌(Google)开发的。Google 通过一篇很有影响力的论文对这个计算模型进行了描述，本书附录部分可看到该论文，名为《MapReduce：大数据之上的简化数据处理》。一年后，另一篇名为《Google 文件系统》的论文介绍了 Google 文件系统。这两篇论文启发了道·卡丁 (Doug Cutting) 开发了 Hadoop。

MapReduce 这个术语来自于两个基本的数据转换操作：map 过程和 reduce 过程。一个

1. 参考链接 <https://cwiki.apache.org/Hive/>。

2. 不过，非常有必要将这个 wiki 链接加入到网址收藏夹中，因为 wiki 中包含了一些我们没有覆盖的、比较模糊的信息。

map 操作会将集合中的元素从一种形式转换成另一种形式。在这种情况下，输入的键-值对会被转换成零到多个键-值对输出。其中，输入和输出的键必须完全不同，而输入和输出的值则可能完全不同。

在 MapReduce 计算框架中，某个键的所有键-值对都会被分发到同一个 reduce 操作中。确切地说，这个键和这个键所对应的所有值都会被传递给同一个 Reducer。reduce 过程的目的是将值的集合转换成一个值（例如对一组数值求和或求平均值），或者转换成另一个集合。这个 Reducer 最终会产生一个键-值对。再次说明一下，输入和输出的键和值可能是不同的。需要说明的是，如果 job 不需要 reduce 过程的话，那么也是可以无 reduce 过程的。

Hadoop 提供了一套基础设施来处理大多数困难的工作以保证任务能够执行成功。例如，Hadoop 决定如果将提交的 job 分解成多个独立的 map 和 reduce 任务 (task) 来执行，它就会对这些 task 进行调度并为其分配合适的资源，决定将某个 task 分配到集群中哪个位置（如果可能，通常是这个 task 所要处理的数据所在的位置，这样可以最小化网络开销）。它会监控每一个 task 以确保其成功完成，并重启一些失败的 task。

Hadoop 分布式文件系统（也就是 HDFS），或者一个同类的分布式文件系统，管理着集群中的数据。每个数据块 (block) 都会被冗余多份（通常默认会冗余 3 份），这样可以保证不会因单个硬盘或服务器的损坏导致数据丢失。同时，因为其目标是优化处理非常大的数据集，所以 HDFS 以及类似的文件系统所使用的数据块都非常大，通常是 64MB 或是这个值的若干倍。这么大的数据块可以在硬盘上连续进行存储，这样可以保证以最少的磁盘寻址次数来进行写入和读取，从而最大化提高读写性能。

为了更清晰地介绍 MapReduce，让我们来看一个简单的例子。Word Count 算法已经被称为是 MapReduce 计算框架中的“Hello World”程序¹了。Word Count 会返回在语料库（单个或多个文件）中出现的所有单词以及单词出现的次数。输出内容会显示每个单词和它的频数，每行显示一条。按照通常的习惯，单词（输出的键）和频数（输出的值）通常使用制表符进行分割。

图 1-1 显示了在 MapReduce 计算框架中 Word Count 程序是如何运作的。

这里有很多内容要讲，所以我们会从左到右来讲解这个图。

图 1-1 中左边的每个 Input（输入）框内都表示一个单独的文件。例子中有 4 个文件，其中第 3 个文件是个空文件，为了便于简单描述，其他 3 个文件都仅仅包含有少量几个单词。

默认情况下，每个文档都会触发一个 Mapper 进程进行处理。而在实际场景下，大文件可能会被划分成多个部分，而每个部分都会被发送给一个 Mapper 进行处理。同时，也

1.对于不是开发者的用户，这里需要补充说明的是“Hello World”程序通常是学习一门新的语言或者工具集的第一个程序。

有将多个小文件合并成一个部分供某个 Mapper 进行处理的方法。不过，我们当前不必深究这些细节性问题。

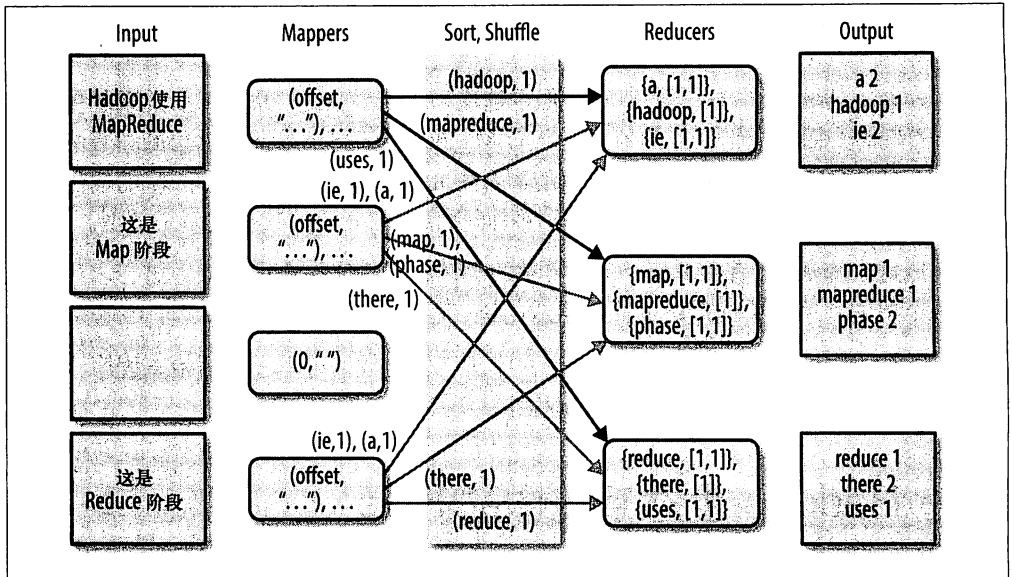


图 1-1 使用 MapReduce 执行 WordCount 算法

MapReduce 计算框架中的输入和输出的基本数据结构是键-值对。当 Mapper 进程启动后，其将会被频繁调用来处理文件中的每行文本。每次调用中，传递给 Mapper 的键是文档中这行的起始位置的字符偏移量。对应的值是这行对应的文本。

在 WordCount 程序中，没有使用字符偏移量（也就是没有使用键）。值（也就是这行文本）可以使用很多种方式进行分割（例如，按照空格分隔是最简单的方式，但是这样会遗留下不需要的标点符号），最终这行文本会被分解成多个单词。我们同时假定 Mapper 会将每个单词转换成小写，因此对于“FUN”和“fun”会被认为是同一个单词。

最后，对于这行文本中的每个单词，Mapper 都会输出一个键-值对，以单词作为键并以数字 1 作为值（这里 1 表示“出现 1 次”）。需要注意的是键和值的输出数据类型和输入数据类型是不同的。

Hadoop 神奇的地方一部分在于后面要进行的 Sort（排序）和 Shuffle（重新分发）过程。Hadoop 会按照键来对键-值对进行排序，然后“重新洗牌”，将所有具有相同键的键-值对分发到同一个 Reducer 中。这里有多种方式可以用于决定哪个 Reducer 获取哪个范围内的键对应的数据。这里我们先不必考虑这些问题。但是出于说明性目的，我们假设图中使用了一个特殊的按字母数字划分的过程（在实际执行中，会有所不同）。

对于 Mapper 而言如果只是简单地对每个单词输出计数 1 这样的处理的话，那么会在 Sort

和 Shuffle 过程中产生一定的网络和磁盘 I/O 浪费（不过，这样并不会减少 Mapper 的内存使用）。有一个优化就是跟踪每个单词的频数，然后在 Mapper 结束后只输出每个单词在这个 Mapper 中的总频数。对于这个优化有好几种实现方式，但是，最简单的方式应该是逻辑是正确的，而且对于这个讨论，理由是充足的。

每个 Reducer 的输入同样是键-值对，但是这次，每个键将是 Mapper 所发现的单词中的某一个单词，而这个键对应的值将是所有 Mapper 对于这个单词计算出的频数的一个集合。需要注意的是键的数据类型和值的集合中元素的数据类型和 Mapper 的输出是一致的。也就是说，键的类型是一个字符串，而集合中的元素的数据类型是整型。

为了完成这个算法，所有的 Reducer 需要做的事情就是将值集合中的频数进行求和然后写入每个单词和这个单词最终的频数组成的键-值对。

Word Count 不是一个虚构的例子。这个程序所产生的数据可用于拼写检查程序、计算机语言检测和翻译系统，以及其他应用程序。

1.2 Hadoop 生态系统中的 Hive

WordCount 算法，和基于 Hadoop 实现的大多数算法一样，有那么点复杂。当用户真正使用 Hadoop 的 API 来实现这种算法时，甚至有更多的底层细节需要用户自己来控制。这是一个只适用于有经验的 Java 开发人员的工作，因此也就将 Hadoop 潜在地放在了一个非程序员用户无法触及的位置，即使这些用户了解他们想使用的算法。

事实上，许多这些底层细节实际上进行的是从一个任务 (job) 到下一个任务 (job) 的重复性工作，例如，将 Mapper 和 Reducer 一同写入某些数据操作构造这样的底层的繁重的工作，通过过滤得到所需数据的操作，以及执行类似 SQL 中数据集键的连接 (JOIN) 操作等。不过幸运的是，存在一种方式，可以通过使用“高级”工具自动处理这些情况来重用这些通用的处理过程。

这也就是引入 Hive 的原因。Hive 不仅提供了一个熟悉 SQL 的用户所能熟悉的编程模型，还消除了大量的通用代码，甚至是那些有时是不得不使用 Java 编写的令人棘手的代码。

这就是为什么 Hive 对于 Hadoop 是如此重要的原因，无论用户是 DBA 还是 Java 开发工程师。Hive 可以让你花费相当少的精力就可以完成大量的工作。

图 1-2 显示了 Hive 的主要“模块”以及 Hive 是如何与 Hadoop 交互工作的。

有好几种方式可以与 Hive 进行交互。本书中，我们将主要关注于 CLI，也就是命令行界面。对于那些更喜欢图形用户界面的用户，可以使用现在逐步出现的商业和开源的解决方案，例如 Karmasphere 发布的一个商业产品 (<http://karmasphere.com>)，Cloudera 提供的开源的 Hue 项目 (<https://github.com/cloudera/hue>)，以及 Qubole 提供的“Hive 即服务”方式 (<http://qubole.com>)，等。

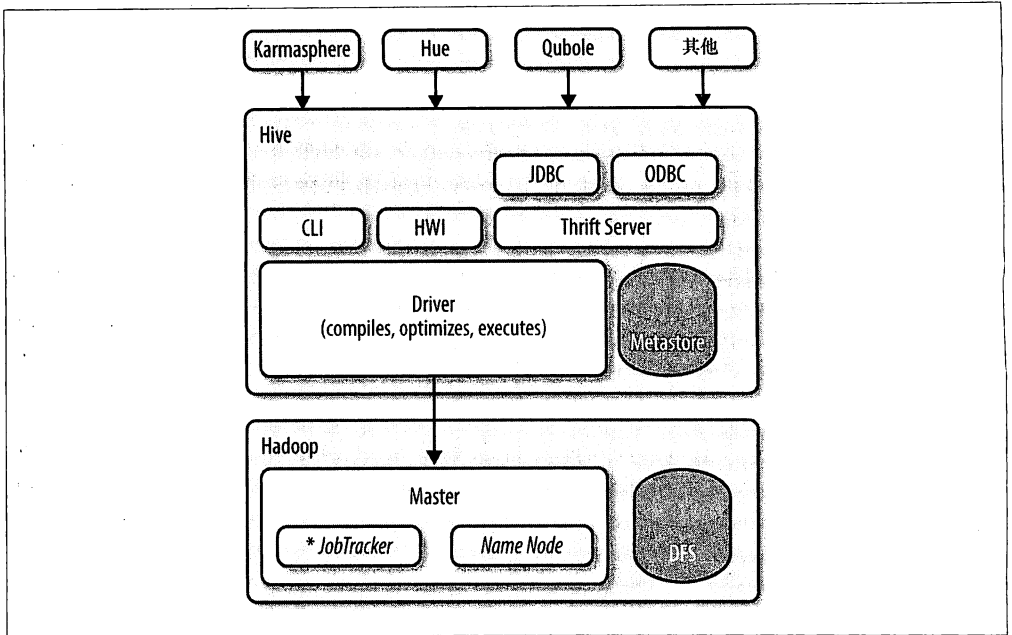


图 1-2 Hive 组成模块

Hive 发行版中附带的模块有 CLI，一个称为 Hive 网页界面 (HWI) 的简单网页界面，以及可通过 JDBC、ODBC 和一个 Thrift 服务器 (参考第 16 章) 进行编程访问的几个模块。

所有的命令和查询都会进入到 Driver (驱动模块)，通过该模块对输入进行解析编译，对需求的计算进行优化，然后按照指定的步骤执行 (通常是启动多个 MapReduce 任务 (job) 来执行)。当需要启动 MapReduce 任务 (job) 时，Hive 本身是不会生成 Java MapReduce 算法程序的。相反，Hive 通过一个表示“job 执行计划”的 XML 文件驱动执行内置的、原生的 Mapper 和 Reducer 模块。换句话说，这些通用的模块函数类似于微型的语言翻译程序，而这个驱动计算的“语言”是以 XML 形式编码的。

Hive 通过和 JobTracker 通信来初始化 MapReduce 任务 (job)，而不必部署在 JobTracker 所在的管理节点上执行。在大型集群中，通常会有网关机专门用于部署像 Hive 这样的工具。在这些网关机上可远程和管理节点上的 JobTracker 通信来执行任务 (job)。通常，要处理的数据文件是存储在 HDFS 中的，而 HDFS 是由 NameNode 进行管理的。

Metastore (元数据存储) 是一个独立的关系型数据库 (通常是一个 MySQL 实例)，Hive 会在其中保存表模式和其他系统元数据。在第 2 章我们将详细进行讨论。

尽管本书是关于 Hive 的，不过还是有必要提及其他的一些高级工具，这样用户可以根据需求进行选择。Hive 最适合于数据仓库程序，对于数据仓库程序不需要实时响应查询，不需要记录级别的插入、更新和删除。当然，Hive 也非常适合于有一定 SQL 知识

的用户。不过，用户的某些工作可能采用其他的工具会更容易进行处理。

1.2.1 Pig

Hive 的替代工具中最有名的就是 Pig 了（请参考 <http://pig.apache.org>）。Pig 是由 Yahoo! 开发完成的，而同时期 Facebook 正在开发 Hive。Pig 现在同样也是一个和 Hadoop 紧密联系的顶级 Apache 项目。

假设用户的输入数据具有一个或者多个源，而用户需要进行一组复杂的转换来生成一个或者多个输出数据集。如果使用 Hive，用户可能会使用嵌套查询（正如我们将看到的）来解决这个问题，但是在某些时刻会需要重新保存临时表（这个需要用户自己进行管理）来控制复杂度。

Pig 被描述成一种数据流语言，而不是一种查询语言。在 Pig 中，用户需要写一系列的声明语句来定义某些关系和其他一些关系之间的联系，这里每个新的关系都会执行新的数据转换过程。Pig 会查找这些声明，然后创建一系列有次序的 MapReduce 任务 (job)，来对这些数据进行转换，直到产生符合用户预期的计算方式所得到的最终结果。

这种步进式的数据“流”可以比一组复杂的查询更加直观。也因此，Pig 常用于 ETL（数据抽取，数据转换和数据装载）过程的一部分，也就是将外部数据装载到 Hadoop 集群中，然后转换成所期望的数据格式。

Pig 的一个缺点就是其所使用的定制语言不是基于 SQL 的。这是可以理解的，因为 Pig 本身就不是被设计为一种查询语言的，但是这也意味着不适合将 SQL 应用程序移植到 Pig 中，而经验丰富的 SQL 用户可能需要投入更高的学习成本来学习 Pig。

然而，Hadoop 团队通常会将 Hive 和 Pig 结合使用，对于特定的工作选择合适的工具。Alan Gates (O’ Reilly) 所编著的《Pig 编程指南》一书对于 Pig 进行了全面的介绍。

1.2.2 HBase

如果用户需要 Hive 无法提供的数据库特性（如行级别的更新，快速的查询响应时间，以及支持事务）的话，那么该怎么办呢？

HBase 是一个分布式的、可伸缩的数据存储，其支持行级别的数据更新、快速查询和行级事务（但不支持多行事务）。

HBase 的设计灵感来自于谷歌 (Google) 的 BigTable，不过 HBase 并没有实现 BigTable 的所有特性。HBase 支持的一个重要特性就是列存储，其中的列可以组织成列族。列族在分布式集群中物理上是存储在一起的。这就使得当查询场景涉及的列只是所有列的一个子集时，读写速度会快得多。因为不需要读取所有的行然后丢弃大部分的列，而是只需读取需要的列。

可以像键-值存储一样来使用 HBase，其每一行都使用了一个唯一键来提供非常快的速度读写这一行的列或者列族。HBase 还会对每个列保留多个版本的值（按照时间戳进行标记），版本数量是可以配置的，因此，如果需要，可以“时光倒流”回退到之前的某个版本的值。

最后，HBase 和 Hadoop 之间是什么关系？HBase 使用 HDFS（或其他某种分布式文件系统）来持久化存储数据。为了可以提供行级别的数据更新和快速查询，HBase 也使用了内存缓存技术对数据和本地文件进行追加数据更新操作日志。持久化文件将定期地使用附加日志更新进行更新等操作。

HBase 没有提供类似于 SQL 的查询语言，但是 Hive 现在已经可以和 HBase 结合使用了。在第 17.3 节“HBase”中我们将讨论这个结合。

关于 HBase 的更多信息，请参考 HBase 的官方网站，以及参阅 LarsGeorge 所著的《HBase 权威指南》一书。

1.2.3 Cascading、Crunch 及其他

Apache Hadoop 生态系统之外还有几个“高级”语言，它们也在 Hadoop 之上提供了不错的抽象来减少对于特定任务（job）的底层编码工作。为了叙述的完整性，下面我们列举其中的一些来进行介绍。所有这些都是 JVM（Java 虚拟机）库，可用于像 Java、Clojure、Scala、JRuby、Groovy 和 Jython，而不是像 Hive 和 Pig 一样使用自己的语言工具。

使用这些编程语言既有好处也有弊端。它使这些工具很难吸引熟悉 SQL 的非程序员用户。不过，对于开发工程师来说，这些工具提供了图灵完全的编程语言的完全控制。Hive 和 Pig 都是图灵完全性的（译者注：图灵完全性通常是指具有无限存储能力的通用物理机器或编程语言）。当我们需要 Hive 本身没有提供的额外功能时，我们需要学习如何用 Java 编码来扩展 Hive 功能(见表 1-1)。

表 1-1 其他可选的 Hadoop 之上的高级语言库

名称	URL	描述
Casading	http://cascading.org	提供数据处理抽象的 Java API。目前有很多支持 Casading 的特定领域语言（DSL），采用的是其他的编程语言，例如 Scala、Groovy、JRuby 和 Jython
Casalog	https://github.com/nathanmarz/casalog	Casading 的一个 Clojure DSL，其提供了源于 Datalog 处理和查询抽象过程灵感而产生的附属功能
Crunch	https://github.com/cloudera/crunch	提供了可定义数据流管道的 Java 和 Scala API

因为 Hadoop 是面向批处理系统的，所以存在更适合事件流处理的使用不同的分布式计算模式的工具。对事件流进行处理时，需要近乎“实时”响应。这里我们列举了其中一些项目（见表 1-2）。

表 1-2 没有使用 MapReduce 的分布式处理工具

名称	URL	描述
Spark	http://www.spark-project.org/	一个基于 Scala API 的分布式数据集的分布式计算框架。其可以使用 HDFS 文件，而且其对于 MapReduce 中多种计算可以提供显著的性能改进。同时还有一个将 Hive 指向 Spark 的项目，称作 Shark (http://shark.cs.berkeley.edu/)
Storm	https://github.com/nathanmarz/storm	一个实时事件流处理系统
Kafka	http://incubator.apache.org/kafka/index.html	一个分布式的发布-订阅消息传递系统

最后，当无需一个完整的集群时（例如，处理的数据集很小，或者对于执行某个计算的时间要求并不苛刻），还有很多可选的工具可以轻松地处理原型算法或者对数据集进行探索。表 1-3 列举了一些相对比较热门的工具。

表 1-3 其他数据处理语言和工具

名称	URL	描述
R	http://r-project.org/	一个用于统计分析和数据图形化展示的开源语言，通常在数据与统计学家、经济学家等人群中很受欢迎。R 不是一个分布式系统，所有其可以处理的数据大小是有限的。社区正努力将 R 和 Hadoop 结合起来
Matlab	http://www.mathworks.com/products/matlab/index.html	一个受工程师和科学家欢迎的商业数据分析和数值方法计算系统
Octave	http://www.gnu.org/software/octave/	Matlab 对应的一个开源版
Mathematica	http://www.wolfram.com/mathematica/	一个商业数据分析、人工智能和数值方法运算系统，这是个可以受科学家和工程师欢迎的工具
SciPy, NumPy	http://scipy.org	数据科学家广泛使用的、使用 Python 进行科学编程的软件

1.3 Java 和 Hive：词频统计算法

如果用户不是 Java 工程师，那么可以直接跳到下一节。

如果用户是名 Java 工程师，那么可能需要阅读本节，因为用户需要为其所在组织的 Hive 用户提供技术支持。你可能会质疑如何使用 Hive 解决自己的工作。如果是这样的话，那么可以先看看下面这个实现了之前我们所讨论的 Word Count 算法的例子，我们先学会使用 Java MapReduce API，然后再学习如何使用 Hive。

通常都会使用 Word Count 作为用户学习使用 Java 编写 MapReduce 程序的例子，因为这样用户可以关注于 API。因此，Word Count 已经成为 Hadoop 世界中的“Hello World”程序了。

Apache Hadoop 分支版本中包含有下面的这个 Java 实现¹。如果读者并不了解 Java（但是你若仍在读本节内容的话），也不必担心，我们提供这个代码只是为了方便用户进行大小对比。

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

    public static class Map extends Mapper<LongWritable, Text, Text,
IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }
}
```

1. Apache Hadoop word count: <http://wiki.apache.org/hadoop/WordCount>.

```

    }
}

public static class Reduce extends Reducer<Text, IntWritable, Text,
IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context
context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    Job job = new Job(conf, "wordcount");

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

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

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

    job.waitForCompletion(true);
}
}

```

上面是一个有 63 行的 Java 代码。我们不会详细解释其中的 API¹。如下是使用 HiveQL 进行的相同的运算,这时只有 8 行代码,而且不需要进行编译然后生成一个“JAR”(Java 压缩包)文件。

```

CREATE TABLE docs (line STRING);

LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE docs;

```

1. 详细信息请参考 Tom White 所著的《Hadoop 权威指南》一书。

```
CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
  (SELECT explode(split(line, '\s')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
```

我们稍后会解释所有这些 HiveQL 语法。

在上面两个例子中，都是使用尽可能简单的方法将文件中的内容分割成单词，也就是按照空格进行划分的。这个方法不能很好地处理标点，同时也不能识别同一个单词的单数和复数形式，等等。不过，这里这么使用已经可以达到我们的目的了。¹

借助 Java API 可以定制和调整一个算法实现的每个细节。不过，大多数情况下，用户都不需要这个级别的控制，而且当用户需要控制所有那些细节时也会相当地放慢用户的开发进度。

如果你不是一名程序员，那么也就用不着写 Java MapReduce 代码了。不过，如果你已经熟悉 SQL 了，那么学习 Hive 将会相当地容易，而且很多程序也都很容易快速实现。

1.4 后续事情

我们描述了 Hive 在 Hadoop 生态系统中所扮演的重要角色。现在我们开始！

¹还有一个微小的差异。Hive 查询硬编码指定一个指向数据的路径，而 Java 代码把这个路径作为一个输入参数处理。在第 2 章，我们将学习如何在 Hive 脚本中使用变量来避免这种硬编码。

让我们来在个人工作站上安装 Hadoop 和 Hive 吧。这是学习和体验 Hadoop 的一个便捷的方式。之后我们将讨论如何配置 Hive 以了解如何在 Hadoop 集群上使用 Hive。

如果用户已经在使用亚马逊网络服务 (AWS) 了, 那么建立一个供学习 Hive 的最快速途径是在亚马逊弹性 MapReduce 系统 (EMR) 中提交一个 Hive 任务。我们在第 21 章将会讨论这种方式。

如果用户已经会使用安装有 Hive 的 Hadoop 集群的话, 那么我们建议可以跳过本章的第一部分而直接从第 2.3 节 “Hive 内部是什么” 开始看。

2.1 安装预先配置好的虚拟机

用户可以通过多种方式来安装 Hadoop 和 Hive。安装一个完整的 Hadoop 系统 (包含有 Hive) 的一个最容易的方式就是下载一个预先配置好的 *虚拟机* (VM), 这个虚拟机可以在 *VMWare*¹ 或者 *VirtualBox*² 中执行。对于 VMWare, 可以使用 Windows 或 Linux (免费的) 的 *VMWare Player*, 也可以使用 Mac OS X (需付费但并不贵) 的 *VMWare Fusion*。VirtualBox 在这些平台 (包含有 Solaris 平台) 中都是免费的。

虚拟机使用 Linux 作为操作系统, 这也是在生产情况下运行 Hadoop 的唯一指定操作系统³。

1. <http://vmware.com>。

2. <https://www.virtualbox.org/>。

3. 然而, 一些厂商目前开始支持 Hadoop 在其他操作系统中使用。Hadoop 目前已经在生产中应用于各种各样的 UNIX 操作系统上, 其在 Mac OS X 中作开发用时效果很好。



提示

在 Windows 系统中目前使用 Hadoop 的唯一方式就是使用虚拟机，即使安装了 *Cygwin* 或其他类似的类 UNIX 软件。

目前提供的大多数预先配置好的虚拟机 (VM) 仅是为 VMWare 设计的，但是如果用户偏好使用 *VirtualBox*，那么也是可以在网站上找到如何将某个特定的 VM 导入到 *VirtualBox* 的指南的。

用户可以从表 2-1¹提供的网站中下载指定的预先配置好的虚拟机。根据这些网站上的指南可以下载并导入到 VMWare 的虚拟机。

表 2-1 为 VMWare 提供的预先配置好的 Hadoop 虚拟机

提供者	URL	注意事项
Cloudera.Inc	https://ccp.cloudera.com/display/SUPPORT/Cloudera's+Hadoop+Demo+VM	使用 Cloudera 自己的 Hadoop 分支，CDH3 或者 CDH4
MapR.Inc	http://www.mapr.com/doc/display/MapR/Quick+Start+-+Test+Drive+MapR+on+a+Virtual+Machine	MapR 的 Hadoop 分支，其将 HDFS 替换为了 MapR 文件系统 (MapR-F5)
Hortonworks.Inc	http://docs.hortonworks.com/HDP-1.0.4-PREVIEW-6/Using_HDP_Single_Box_VM/HDP_Single_Box_VM.htm	基于最新的稳定的 Apache 的发行版
Think Big Analytics.Inc	http://thinkbigacademy.s3-website-us-east-1.amazonaws.com/vm/README.html	基于最新的稳定的 Apache 的发行版

下一步，可以直接从第 2.3 节“Hive 内部是什么”开始看。

2.2 安装详细步骤

虽然使用一个预先配置好的虚拟机可能是使用 Hive 的一个便捷方式，但是自己安装 Hadoop 和 Hive 可以让用户更清楚地知道这些工具是如何工作的，特别是，如果用户是一个开发者的话，这个很重要。

下面所提供的指南描述的是在用户个人 Linux 或者 Mac OS X 工作站中安装 Hadoop 和 Hive 所必需的最少的步骤。对于生产环境下的安装过程，请查阅所使用的 Hadoop 分支

1.这些是写这本书时所提供的 URL 链接。

推荐的安装流程。

2.2.1 装 Java

Hive 依赖于 Hadoop，而 Hadoop 依赖于 Java。用户需要确认使用的操作系统中安装有 v1.6.* 或者 v1.7.* 版本的 JVM (Java 虚拟机)。尽管用户执行 Hive 只需要使用到 JRE (Java 运行时环境)，但是本书中还是演示了如何使用 Java 编码对 Hive 进行扩展，因此用户还需要安装完整的 JDK (Java 开发工具集) 来编译这些例子。但是，如果用户不是开发者，可以不进行此操作，本书所附带的源码分支版本中已经包含有编译过的例子了。安装完成后，用户需要确认 Java 已经存在于环境中，而且已经设置好了 JAVA_HOME 环境变量。

1. Linux 系统中 Java 安装步骤

在 Linux 操作系统上，下面的指令是在 `/etc/profile.d/` 目录下创建一个 `bash` 文件，其为所有的用户定义了一个 `JAVA_HOME` 环境变量，需要 root 访问权限才能够修改这个目录下的环境变量设置，而且修改将会影响到系统中所有用户。(我们使用 `$` 作为 `bash` shell 提示符。) Oracle JVM 安装程序通常会将软件安装在 `/usr/java/jdk-1.6.X` (对于 v1.6 版本) 目录下，而且会从 `/usr/java/default` 和 `/usr/java/latest` 路径建立软链接到安装路径下。

```
$ /usr/java/latest/bin/java -version
java version "1.6.0_23"
Java(TM) SE Runtime Environment (build 1.6.0_23-b05)
Java HotSpot(TM) 64-Bit Server VM (build 19.0-b09, mixed mode)
$ sudo echo "export JAVA_HOME=/usr/java/latest" > /etc/profile.d/java.sh
$ sudo echo "PATH=$PATH: $JAVA_HOME/bin" >> /etc/profile.d/java.sh
$ . /etc/profile
$ echo $JAVA_HOME
/usr/java/latest
```



提示

如果之前是使用“特权账号”来执行某个命令，而不是在执行命令前使用 `sudo` 命令来执行的话(也就是需要执行的命令分为两部分：`sudo` 加上用户需要执行的命令)，那么只需要按照提示要求输入密码即可。如果是个人计算机的话，用户账号通常就具有“`sudo` 权限”。如果没有这个权限，可以联系管理员执行那些命令。

不过，如果用户不希望修改影响到系统里所有的用户，一个替代方案就是，将 `PATH` 和 `JAVA_HOME` 环境变量的定义加入到用户的 `$HOME/.bashrc` 文件中。

```
export JAVA_HOME=/usr/java/latest
export PATH=$PATH:$JAVA_HOME/bin
```

2. Mac OS X 系统中 Java 安装步骤

Mac OS X 系统没有 `/etc/profile.d` 这样的目录，而且它们通常是单用户系统，因此最好将环境变量的定义语句放置在 `$HOME/.bashrc` 文件中。Java 路径同样也是不同的，而且可以位于多个不同的位置¹。这里有一些例子。用户需要确认自己的 Mac 电脑中 Java 的安装路径，然后对下面的定义进行相应的调整。下面是 Mac OS X 系统中 Java 1.6 的环境配置实例：

```
$ export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/  
1.6/Home  
$ export PATH=$PATH:$JAVA_HOME/bin
```

下面是 Mac OS X 系统中 Java 1.7 的环境配置实例。

```
$ export JAVA_HOME=/Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/  
Home  
$ export PATH=$PATH:$JAVA_HOME/bin
```

OpenJDK 1.7 发行版也是安装在 `/Library/Java/JavaVirtualMachines` 目录下的。

2.2.2 安装 Hadoop

Hive 运行在 Hadoop 之上。Hadoop 是一个非常活跃的开源项目，其具有很多的发行版和分支。同时，很多的商业软件公司现在也在开发他们自己的 Hadoop 分支，有时会对某些组件进行个性化的增强或者替换。这种形式可以促进创新，但是同时也会产生潜在的混乱和兼容性问题。

保持使用最新版本的软件可以让用户使用最新的增强功能并且新版本通常会修复很多的 BUG。随后，有时用户可能会发现新的 BUG 以及兼容性问题。本书中，我们将介绍如何安装 Apache Hadoop v0.20.2 发行版。这个版本并非是最新稳定发行版，但是这个版本是性能和兼容性都可以信赖的一个标准版。

不过，用户应该能够毫无问题地选择一个不同版本、分支或者发行版来学习和使用 Hive，例如 Apache Hadoop v0.20.205 版本或者 1.0.* 版本，Cloudera 的 CDH3 或者 CDH4 版本，MapR 的 M3 或者 M5 版本，以及即将面市的 Hortonworks 分支版本。注意 Cloudera，MapR 以及未来的 Hortonworks 分支版都包含有一个捆绑的 Hive 发行版。

不过，我们不建议安装新的阿尔法版的“下一代的”Hadoop v2.0 版本（也就是所谓的 v0.23 版），至少本书的目标不是这个版本。虽然这个发行版将给 Hadoop 生态圈带来重大的增强，但是对于我们来说太新了。

1.至少这个是 Dean 的 Mac 电脑中当前的配置情况。这个差异可能实际上反映了一个事实，即：Mac OS X 中 Java 的管理工作自 Java 1.7 版本开始从 Apple 过渡到了 Oracle。

在 Linux 系统中，可以通过执行如下命令安装 Hadoop。注意：对于 `wget` 命令，因为这行太长，我们将其分为了两行。

```
$ cd ~ # or use another directory of your choice.
$ wget \
  http://www.us.apache.org/dist/hadoop/common/hadoop-0.20.2/hadoop-0.20.
2.tar.gz
$ tar -xzf hadoop-0.20.2.tar.gz
$ sudo echo "export HADOOP_HOME=$PWD/hadoop-0.20.2" > /etc/profile.d/
hadoop.sh
$ sudo echo "PATH=$PATH:$HADOOP_HOME/bin" >> /etc/profile.d/hadoop.sh
$ . /etc/profile
```

对于 Mac OS X 系统，可以通过执行如下命令来安装 Hadoop。注意：对于 `curl` 命令，因为这行太长，我们将其分为了两行。

```
$ cd ~ # or use another directory of your choice.
$ curl -o \
  http://www.us.apache.org/dist/hadoop/common/hadoop-0.20.2/hadoop-0.20.
2.tar.gz
$ tar -xzf hadoop-0.20.2.tar.gz
$ echo "export HADOOP_HOME=$PWD/hadoop-0.20.2" >> $HOME/.bashrc
$ echo "PATH=$PATH:$HADOOP_HOME/bin" >> $HOME/.bashrc
$ . $HOME/.bashrc
```

在下文中，我们将假设用户，像上面的命令一样，已经将 `$HADOOP_HOME/bin` 加入到环境变量路径中了。这样设置后用户只需要输入 `hadoop` 命令即可，无需加上路径前缀。

2.2.3 本地模式、伪分布式模式和分布式模式

在继续讲解之前，我们先阐明 Hadoop 的不同运行模式。我们前面提到默认的模式是本地模式，这种模式下使用的是本地文件系统。在本地模式下，当执行 Hadoop job 时（包含有大多数的 Hive 查询），Map task 和 Reduce task 在同一个进程中执行。

真实的集群配置的都是分布式模式，其中所有没有完整 URL 指定的路径默认都是分布式文件系统（通常是 HDFS）中的路径，而且由 *JobTracker* 服务来管理 job，不同的 task 在不同的进程中执行。

对于在个人计算机上工作的开发者来说，一个进退两难的现实问题是，本地模式并不能真实地反映真实集群的行为状况，这个是测试程序时需要记住的事情。为了解决这个需求，一台物理机可以配置成在伪分布式模式下执行。这种模式下执行的行为和在分布式模式下的行为是一致的。也就是说，引用的文件系统默认为分布式文件系统，而且由 *JobTracker* 服务来管理 job，但是实际上只有一台物理机。因此，例如，HDFS 文件块冗余数这时限制为一个备份。换句话说，行为类似于只有一个节点的“集群”。在第 2.5 节“配置 Hadoop 环境”中我们会讨论这些配置项。

因为 Hive 中大多数工作是使用 Hadoop 的 job，所有 Hive 的行为可以反映出用户所使用的 Hadoop 运行模式。不过，即使在分布式模式下执行，Hive 还是可以在提交查询前判断是否可以使用本地模式来执行这个查询。这时它会读取数据文件，然后自己管理 MapReduce task，最终提供更快执行方式。不过，对于 Hadoop 来说，不同模式之间的差异相对于部署方式更多地在于执行方式上。

本书中大部分情况下，不会关心用户使用的是哪种模式。我们将假定用户是以本地模式在个人计算机上工作的，而且我们将讨论这个模式所影响的情况。



提示

当处理小数据集时，使用本地模式执行可以使 Hive 执行得更快些。设置如下这个属性 `sethive.exec.mode.local.auto=true` 时，将会触发 Hive 更主动地使用这种模式，即使当前用户是在分布式模式或伪分布式模式下执行 Hadoop 的。如果想默认使用这个配置，可以将这个命令加到 `$HOME/.hiverc` 文件中（参考第 2.7.5 “`.hiverc` 文件” 章节）。

2.2.4 测试 Hadoop

假设用户使用的是本地模式，我们通过两种不同方式来看看本地文件系统。下面这个是通过 Linux `ls` 命令查看 Linux 系统 “root” 目录下的目录信息：

```
$ ls /
bin  cgroup  etc  lib  lost+found  mnt  opt  root  selinux  sys  user  var
boot  dev  home  lib64  media  null  proc  sbin  srv  tmp  usr
```

Hadoop 本身提供了一个 `dfs` 工具，这个工具可以提供对当前默认文件系统的基本文件的操作，例如 `ls` 命令。因为现在我们使用的是本地模式，所以当前默认的文件系统是本地文件系统¹。

```
$ hadoop dfs -ls /
Found 26 items
drwxrwxrwx - root root 24576 2012-06-03 14:28 /tmp
drwxr-xr-x - root root 4096 2012-01-25 22:43 /opt
drwx----- - root root 16384 2010-12-30 14:56 /lost+found
drwxr-xr-x - root root 0 2012-05-11 16:44 /selinux
dr-xr-x--- - root root 4096 2012-05-23 22:32 /root
...
```

如果用户遇到错误信息提示“没有发现命令 `hadoop`”，那么可以通过直接指定绝对路径来执行这个命令（例如，`$HOME/hadoop-0.20.2/bin/hadoop`），或者将 `bin` 目录加载

1.不幸的是，`dfs -ls` 命令仅提供“长列表”格式。没有像 Linux 系统中 `ls` 命令显示的那种简短格式。

到 PATH 环境变量中，这个在前面第 2.2.2 节“安装 Hadoop”中讨论过了。



提示

当你发现需要频繁地使用 `hadoop dfs` 命令时，最好为这个命令定义一个别名（例如，`alias hdfs="hadoop dfs"`）。

Hadoop 提供了 MapReduce 计算框架。Hadoop 分支中都会包含有一个 Word Count 算法实现，这个我们在第 1 章讨论过了。现在，我们来运行它！

首先我们要创建一个输入目录（要位于用户当前工作目录内）用于存放将要使用 Hadoop 进行处理的文件：

```
$ mkdir wc-in
$ echo "bla bla" > wc-in/a.txt
$ echo "bla wa wa " > wc-in/b.txt
```

使用 `hadoop` 命令指定我们刚刚创建好的文件输入目录来执行 Word Count 程序。需要注意的是，最好每次指定的文件输入和输出路径都是文件夹，而不是文件。这是因为通常输入和（或）输出目录中都有很多文件，这是系统并行性的结果。

如果用户是在本地安装的软件上以本地模式运行这些命令的话，那么 `hadoop` 命令将会在同一进程内加装 MapReduce 组件。如果用户是在集群中运行或者使用伪分布式模式在单台机器上执行的话，那么 `hadoop` 命令将会使用 JobTracker 服务启动一个或者多个不同的进程（也因此如下命令的输出也将是有些差异的）。同时，如果用户使用了和例子中版本不同的 Hadoop 的话，那么需要根据情况修改一下这个 `examples.jar` 的名字。

```
$ hadoop jar $HADOOP_HOME/hadoop-0.20.2-examples.jar wordcount wc-in
wc-out
12/06/03 15:40:26 INFO input.FileInputFormat: Total input paths to process : 2
...
12/06/03 15:40:27 INFO mapred.JobClient: Running job: job_local_0001
12/06/03 15:40:30 INFO mapred.JobClient: map 100% reduce 0%
12/06/03 15:40:41 INFO mapred.JobClient: map 100% reduce 100%
12/06/03 15:40:41 INFO mapred.JobClient: Job complete: job_local_0001
```

这个 Word Count 程序的输出结果可以使用本地文件系统的命令进行查看：

```
$ ls wc-out/*
part-r-00000
$ cat wc-out/*
bla      3
wa       2
```

同时也可以等价的 `dfs` 命令进行查看（这是因为，我们假定是以本地模式执行的）：

```
$ hadoop dfs -cat wc-out/*
```

```
bla      3
wa       2
```



提示

对于非常大的文件，当你只想查询其开始或者结尾部分信息时，这里没有提供 `-moew`、`-head` 或者 `-tail` 子命令。替代方式是，将 `-cat` 命令的输出通过管道传递给 shell 中的 `more`、`head` 或者 `tail` 命令，例如，`hadoop dfs -cat wc-out/* | more`。

注意了我们已经安装了一个 Hadoop 并且对其进行测试了，下面我们来安装 Hive。

2.2.5 安装 Hive

安装 Hive 的过程和安装 Hadoop 的过程非常相似。我们需要先下载一个 Hive 软件压缩包，然后进行解压缩。通常这个压缩包内不会包含有某个版本的 Hadoop。一个 Hive 二进制包可以在多个版本的 Hadoop 上工作。这也意味着和 Hadoop 版本升级相比，升级 Hive 到新的版本会更加容易和低风险。

Hive 使用环境变量 `HADOOP_HOME` 来指定 Hadoop 的所有相关 JAR 和配置文件。因此，在继续进行之前请确认下是否设置好了这个环境变量，这个我们之前有讨论过的。如下命令在 Linux 和 Mac OS X 系统中都可以执行。

```
$ cd ~ # or use another directory of your choice.
$ curl -o http://archive.apache.org/dist/hive/hive-0.9.0/hive-0.9.0-bin.tar.gz
$ tar -xzf hive-0.9.0.tar.gz
$ sudo mkdir -p /user/hive/warehouse
$ sudo chmod a+rwX /user/hive/warehouse
```

正如用户可以从这些命令推断到的，我们使用的是编写本书时最新最稳定的 Hive 发行版，也就是 `v0.9.0` 版本。不过，本书中大多数的资料也都是可以在 Hive `v0.7.*` 和 Hive `v0.8.*` 版本上执行的。当我们碰到这些资源的时候，我们会讲述它们之间有何差异。

用户可能需要将 `hive` 命令加入到环境变量路径中去，正如前面我们对 `hadoop` 进行的处理一样。我们将采用同样的方式，首先定义一个 `HIVE_HOME` 变量，但是和 `HADOOP_HOME` 不同的是，这个变量并非是必须要定义的。本书中某些例子中我们是假定已经定义了这个变量的。

对应 Linux 系统，执行这些命令：

```
$ sudo echo "export HIVE_HOME=$PWD/hive-0.9.0" > /etc/profile.d/hive.sh
$ sudo echo "PATH=$PATH:$HIVE_HOME/bin >> /etc/profile.d/hive.sh
$ . /etc/profile
```

对应 Mac OS X 系统，执行这些命令：

```
$ echo "export HIVE_HOME=$PWD/hive-0.9.0" >> $HOME/.bashrc
$ echo "PATH=$PATH:$HIVE_HOME/bin" >> $HOME/.bashrc
$ . $HOME/.bashrc
```

2.3 Hive 内部是什么

Hive 二进制分支版本核心包含 3 个部分。主要部分是 Java 代码本身。在 `$HIVE_HOME/lib` 目录下可以发现众多的 JAR（Java 压缩包）文件，例如 `hive-exec*.jar` 和 `hive-metastore*.jar`。每个 JAR 文件都实现了 Hive 功能中某个特定的部分，具体详情我们现在无需关心。

`$HIVE_HOME/bin` 目录下包含可以执行各种各样 Hive 服务的可执行文件，包括 `hive` 命令行界面（也就是 CLI）。CLI 是我们使用 Hive 的最常用方式。除非有特别说明，否则我们都使用 `hive`（小写，固定宽度的字体）来代表 CLI。CLI 可用于提供交互式的界面供输入语句或者可以供用户执行含有 Hive 语句的“脚本”，这个我们后面会有介绍。

Hive 还有一些其他组件。Thrift 服务提供了可远程访问其他进程的功能，也提供使用 JDBC 和 ODBC 访问 Hive 的功能。这些都是基于 Thrift 服务实现的。在后面的多章内容中我们将讲述这些功能。

所有的 Hive 客户端都需要一个 `metastoreservice`（元数据服务），Hive 使用这个服务来存储表模式信息和其他元数据信息。通常情况下会使用一个关系型数据库中的表来存储这些信息。默认情况下，Hive 会使用内置的 Derby SQL 服务器，其可以提供有限的、单进程的存储服务。例如，当使用 Derby 时，用户不可以执行 2 个并发的 Hive CLI 实例，然而，如果是在个人计算机上或者某些开发任务上使用的话这样也是没问题的。对于集群来说，需要使用 MySQL 或者类似的关系型数据库。第 2.5.3 节“使用 JDBC 连接元数据”中我们将详细讨论。

最后，Hive 还提供了一个简单的网页界面，也就是 Hive 网页界面（HWI），提供了远程访问 Hive 的服务。

`conf` 目录下存放了配置 Hive 的配置文件。Hive 具有非常多的配置属性，根据需要后面我们会进行介绍。这些属性控制的功能包括元数据存储（如数据存放在哪里）、各种各样的优化和“安全控制”，等等。

2.4 启动 Hive

终于我们可以从 Hive 命令行界面（CLI）开始执行一些命令了！我们会简要地说明一

下出现了什么情况，而在后面才会进行详尽的讨论。

在后面的会话中，我们将使用`$HIVE_HOME/bin/hive`命令，这是个 `bash shell` 脚本，用于启动 CLI。下面脚本中出现的`$HIVE_HOME` 可以根据需要替换为用户的 Hive 安装目录路径。如果用户已经将`$HIVE_HOME/bin` 加入到环境变量 `PATH` 中了，那么只需要输入 `hive` 就可以执行命令了。本书的后续部分将假定用户已经加入到环境变量中了。

和以前一样，`$` 是 `bash` 提示符。在 Hive CLI 中，字符串 `hive>` 是 Hive 的提示符，而大于号 `>` 是第 2 个提示符。这里有个样例会话，为清晰表达，我们在每个命令后面都人为地增加了一个空行：

```
$ cd $ HIVE_HOME
$ bin/hive
Hive history file=/tmp/myname/hive_job_log_myname_201201271126_1992326118.
txt
hive> CREATE TABLE x (a INT);
OK
Time taken: 3.543 seconds

hive> SELECT * FROM x;
OK
Time taken: 0.231 seconds

hive> SELECT *
> FROM x;
OK
Time taken: 0.072 seconds

hive> DROP TABLE x;
OK
Time taken: 0.834 seconds

hive> exit;
$
```

CLI 所打印出的第 1 行显示的是 CLI 将关于用户所执行的命令和查询的日志数据所存放在的本地文件系统中的位置。如果一个命令或者查询执行成功了，那么输出中的第 1 行将是 `OK`，然后才会紧跟着输出内容，最后以一行表示命令或者查询执行所消耗的时间的输出信息结尾。



提示

贯穿本书，我们将按照 SQL 惯例使用大写字母显示 Hive 的关键字（例如，`CREATE`、`TABLE`、`SELECT` 和 `FROM`），尽管在 Hive 中关键字是大小写无关的，这也和 SQL 的惯例是一致的。

在后面，对于所有的会话我们通常都会在命令的输出后面加上空行。同

时，在启动一个会话的时候，我们也将省略掉那行关于日志文件所在位置的输出。对于单个命令或者查询，除非在特别的情况下（例如当我们期望强调某个命令或者查询执行成功，但是其又没有其他输出信息时），否则我们也将省略掉“OK”和“Time taken:...”这些行信息。

通过上面一系列连续的提示符，我们创建了一个简单的表，表名是 `x`，包含有一个名为 `a` 的 `INT`（4 字节整型）类型字段，然后对这个表查询了 2 次，第 2 次查询是为了显示查询语句和命令，可以多行输入。最后，我们删除了这个表。

如果用户使用默认的 `Derby` 数据库作为元数据存储的话，那么这时可以注意到在用户当前工作目录下新出现了一个名为 `metastore_db` 的目录，这个目录是在启动 `Hive` 会话时由 `Derby` 创建的。如果用户使用的是之前所介绍的虚拟机（VM），由于其配置可能不同，就会使得其行为也可能是不同的，这个我们后面会做讨论。

在用户所在的任意工作目录下都创建一个名为 `metastore_db` 的子目录并不方便，因为当用户切换到新的工作目录下时，`Derby` 会“忘记”在前一个目录下的元数据存储信息！在下一节中，我们将讨论如何为元数据存储数据库配置一个永久的路径，同时还会做其他一些调整。

2.5 配置 Hadoop 环境

下面我们稍微深入地看看 Hadoop 的不同模式并讨论和 `Hive` 相关的更多配置问题。

如果用户正在一个已经存在的集群中使用 Hadoop 或者是使用一个虚拟机实例的话，那么可以跳过本节。如果你是一个开发者或者你是自己安装 Hadoop 和 `Hive` 的，你将会对本节后面的内容感兴趣。不过，我们不会提供一个完整的讨论。参考附录 A “*Hadoop: The Definitive Guide by Tom White*” 来查看不同模式下的详细配置。

2.5.1 本地模式配置

回想下，在本地模式中，所有提及的文件都存储在本地文件系统而不是分布式文件系统中。其中没有服务在运行。相反地，用户的 `job` 在同一个 JVM 实例中执行所有的任务。

下面图 2-1 阐明了在本地模式下执行 Hadoop `job` 的过程。

如果用户计划经常使用本地模式的话，那么非常有必要为 `Derby metastore_db`（这里 `Hive` 存储了用户的表等元数据信息）配置一个标准的位置。

用户如果不想使用默认的路径，那么还可以配置一个不同的目录来存储表数据。对于本地模式，默认路径是 `file:///user/hive/warehouse`，对于其他模式，默认存储路径是 `hdfs://namenode_server/user/hive/warehouse`。

首先，切换到\$HIVE_HOME/conf目录下。好奇者可能会看到hive-default.xml.template这个大文件。这个文件中包含了Hive提供的配置属性以及默认的属性值。这些属性中的绝大多数，用户可以直接忽略不管。用户所作的配置修改只需要在hive-site.xml文件中进行就可以了。如果这个文件不存在，那么用户需要自己创建一个。

这里有个配置文件样例，其为本地模式执行配置了几个属性（见例2-1）。

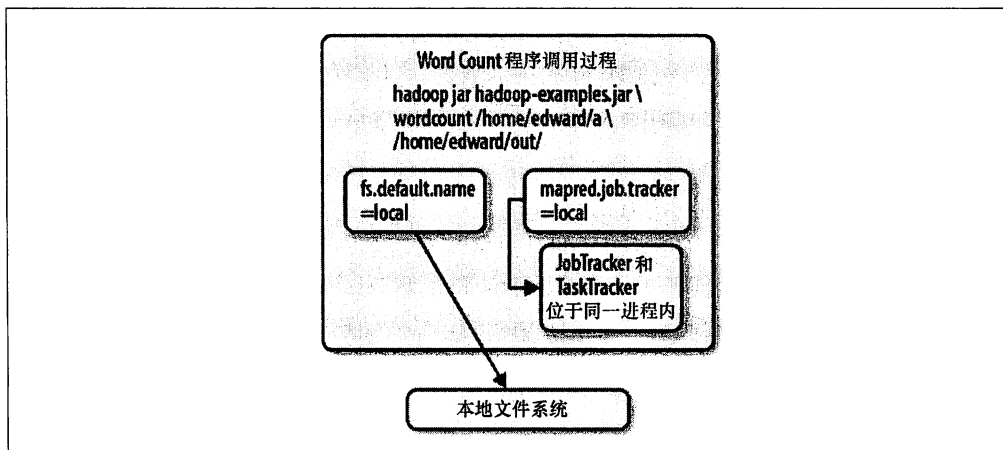


图 2-1 本地模式下的 Hadoop

例 2-1 本地模式下的 hive-site, xml 配置文件。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>/home/me/hive/warehouse</value>
    <description>
      Local or HDFS directory where Hive keeps table contents.
    </description>
  </property>
  <property>
    <name>hive.metastore.local</name>
    <value>true</value>
    <description>
      Use false if a production metastore server is used.
    </description>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby;;databaseName=/home/me/hive/metastore_db;create=true</value>
    <description>
```

```
    The JDBC connection URL.
  </description>
</property>
</configuration>
```

用户可以根据需要从配置文件中移除掉某些不需要改变的属性，也就是 `<property>...</property>` 标签包含的内容。

正如 `<description>` 标签内所表明的，属性 `hive.metastore.warehouse.dir` 告诉 Hive 在本地文件系统中使用哪个路径来存储 Hive 表中的数据。（这个值会追加到 Hadoop 配置文件中所配置的属性 `fs.default.name` 的值，其默认为 `file:///`。）用户可以根据需要为这个属性指定任意的目录路径。

属性 `hive.metastore.local` 的默认值就是 `true`，因此在例 2-1 中我们其实是没必要将这个属性加进去的。放在例子中更多的目的是提供文档信息。这个属性控制着是否连接到一个远程 metastore server 或者是否作为 Hive Client JVM 的构成部分重新打开一个新的 metastore server。这个设置通常是设置成 `true` 的，然后使用 JDBC 直接和一个关系型数据库通信。当设置为 `false` 时，Hive 将会通过一个 metastore server 来进行通信，这个我们将在第 16.8 节“metastore 方法”中进行讨论。

属性 `javax.jdo.option.ConnectionURL` 的值对默认的值进行了简单的修改。这个属性告诉 Hive 如何连接 metastore server。默认情况下，它使用当前的工作目录作为属性值字符串中的 `databaseName` 部分。如例 2-1 中所示，我们使用 `databaseName=/home/me/hive/metastore_db` 作为绝对路径，它是 `metastore_db` 目录所在的路径。这样设置可以解决每次开启一个新的 Hive 会话时，Hive 自动删除工作目录下的 `metastore_db` 目录的问题。现在，我们不管在哪个目录下工作都可以访问到所有的元数据。

2.5.2 分布式模式和伪分布式模式配置

在分布式模式下，集群中会启动多个服务。*JobTracker* 管理着 job，而 HDFS 则由 *NameNode* 管理着。每个工作节点上都有 job task 在执行，由每个节点上的 *TaskTracker* 服务管理着；而且每个节点上还存放有分布式文件系统中的文件数据块，由每个节点上的 *DataNode* 服务管理着。

图 2-2 展示了 Hadoop 集群的一个典型的分布式模式配置。

我们约定对于集群的内外网，使用 `*.domain.pvt` 作为我们的 DNS 命名规范。

*伪分布式模式*也几乎是相同的，事实上它就是一个单节点的集群。

我们假定用户的管理员已经配置好了 Hadoop，包括分布式文件系统（例如，HDFS，或者请参考附录 A Hadoop: The Definitive Guide by Tom White）。因此，我们将关注其与 Hive 所要求的不同的配置过程。

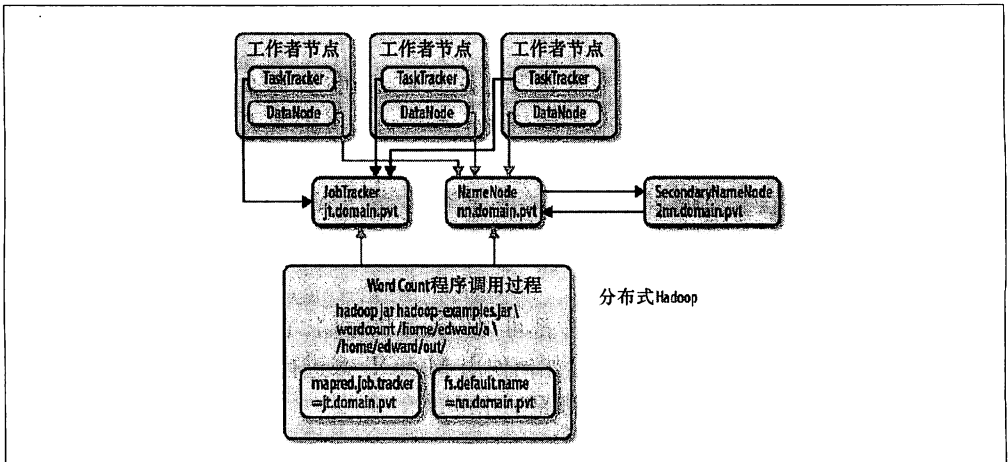


图 2-2 分布式模式下的 Hadoop

用户可能需要配置的一个 Hive 属性是表存储所位于的顶级文件目录，其由属性 `hive.metastore.warehouse.dir` 指定，在第 2.5.1 节“本地模式配置”中也有进行讨论。

Apache Hadoop 和 MapR 分支中这个属性的默认值是 `/user/hive/warehouse`，当 Hadoop 配置的是分布式模式或者伪分布式模式时，这个路径被认为是分布式文件系统中的路径。对于 Amazon 弹性 MapReduce 系统 (EMR)，当使用 Hive0.M.N 版本时，这个属性的默认值是 `/mnt/hive_0M_N/warehouse`（例如，`/mnt/hive_08_1/warehouse`）。

为这个属性指定不同的值可以允许每个用户定义其自己的数据仓库目录，这样就可以避免影响其他系统用户。因此，用户可能需要使用如下语句来为其自己指定数据仓库目录：

```
set hive.metastore.warehouse.dir=/user/myname/hive/warehouse;
```

如果每次启动 Hive CLI 或者在每个 Hive 脚本前都指定这行语句，那么显得过于冗长。当然，也更容易会忘记设置这个属性值。因此，最好将和这个命令类似的所有命令放置在 `$HOME/.hiverc` 文件中，每一次启动 Hive 都会执行这个文件。参考第 2.7.5 节“hiverc 文件”获得更详细信息。

由此处开始我们将假定值是 `/user/hive/warehouse`。

2.5.3 使用 JDBC 连接元数据

Hive 所需要的组件中只有一个外部组件是 Hadoop 没有的，那就是 *metastore*（元数据存储）组件。元数据存储中存储了如表的模式和分区信息等元数据信息。用户在执行如 `create table x...` 或者 `alter table y...` 等命令时会指定这些信息。因为多用户和系统可能需要并发访问元数据存储，所以默认的内置数据库并不适用于生产环境。



提示

如果用户使用的是单个节点上的伪分布式模式，那么用户可能会发现，为元数据存储设置一个完整的关系型数据库并没多大用处，反而，用户可能希望继续使用默认的 *Derby* 进行元数据存储，但是可以为其设置一个中央位置来储存数据，这个在第 2.5.1 节“本地模式配置”中已经讨论过了。

任何一个适用 JDBC 进行连接的数据库都可用作元数据存储。在实践中，大多数的 Hive 客户端会使用 MySQL。我们也将讨论如何使用 MySQL 来进行元数据存储。执行过程对于其他适用于 JDBC 连接的数据库都是适用的。



提示

像表的模式信息、分区信息等这些必须的元数据，其信息量是很小的，通常比存储在 Hive 中的数据的量要少的多。因此，用户其实无需为元数据存储提供一个强劲的专用数据库服务器。不过，因为这是一个单点问题 (SPOF)，所以强烈建议用户使用对于其他关系型数据库实例同样适用的标准技术来对这个数据库进行冗余存储和数据备份。这里我们不会探讨这些技术。

对于 MySQL 配置，我们需要知道指定服务运行在哪个服务器和端口。我们将假定是在 `db1.mydomain.pvt` 服务器的 3306 端口上，这个端口也是标准的 MySQL 端口。最后，我们假定存储数据库名为 `hive_db`。我们在例 2-2 中定义了这些属性。

例 2-2 `hive-site.xml` 中的元数据存储数据库配置。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://db1.mydomain.pvt/hive_db?createDatabaseIfNotExist=true</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>database_user</value>
  </property>
  <property>
```

```
<name>javax.jdo.option.ConnectionPassword</name>
  <value>database_pass</value>
</property>
</configuration>
```

用户可能已经注意到 `ConnectionURL` 属性值前缀是 `jdbc:mysql`。

为了使 Hive 能够连接上 MySQL，我们需要将 JDBC 驱动放置在类路径下。MySQL JDBC 驱动 (Jconnector) 可以从如下网址下载：<http://www.mysql.com/downloads/connector/j/>。这个驱动可以放置在 Hive 的库路径下，也就是 `$HIVE_HOME/lib` 目录下。有些团队会将所有这些支持类的库放置在 Hadoop 的库目录下。

驱动和配置设置正确后，Hive 就会将元数据信息存储到 MySQL 中。

2.6 Hive 命令

`$HIVE_HOME/bin/hive` 这个 shell 命令（后面我们省略称为 `hive`）是通向包括 *命令行界面* 也就是 CLI 等 Hive 服务的通道。

我们假定用户已经将 `$HIVE_HOME/bin` 加入到环境变量 `PATH` 中了，则用户只需要在 shell 提示符中输入 `hive`，就可以使用户的 shell 环境（例如 `bash` 环境）找到这个命令。

命令选项

如果用户执行下面的命令，那么可以查看到 `hive` 命令的一个简明说明的选项列表。下面是 `Hive v0.8.*` 和 `Hive v0.9.*` 系列版本的输出：

```
$ bin/hive --help
Usage ./hive <parameters> --service serviceName <service parameters>
Service List: cli help hiveserver hwi jar lineage metastore rcfilecat
Parameters parsed:
  --auxpath : Auxiliary jars
  --config  : Hive configuration directory
  --service : Starts specific service/component. cli is default
Parameters used:
  HADOOP_HOME : Hadoop install directory
  HIVE_OPT    : Hive options
For help on a particular service:
  ./hive --service serviceName --help
Debug help: ./hive --debug --help
```

需要注意 *Service List* 后面的内容。这里提供了几个服务，包括我们绝大多数时间将要使用的 CLI。用户可以通过 `--service name` 服务名称来启用某个服务，尽管其中有几个服务也是有快捷启动方式的。

表 2-2 中描述了最有用的服务。

表 2-2 Hive 服务

选项	名称	描述
cli	命令行界面	用户定义表, 执行查询等。如果没有指定其他服务, 这个是默认的服务。参考第 2.7 节 “命令行界面”
hiveserver	Hive Server	监听来自于其他进程的 Thrift 连接的一个守护进程
hwi	Hive Web 界面	是一个可以执行查询语句和其他命令的简单的 Web 界面, 这样可以不用登录到集群中的某台机器上使用 CLI 来进行查询
jar		hadoop jar 命令的一个扩展, 这样可以执行需要 Hive 环境的应用
metastore		启动一个扩展的 Hive 元数据服务, 可以供多客户端使用 (参考第 2.5.3 节 “使用 JDBC 连接元数据”)
rcfilecat		一个可以打印出 RCFile 格式 (参考第 15.3.2 节 “RCFile”) 文件内容的工具

--auxpath 选项允许用户指定一个以冒号分割的 “附属的” Java 包 (JAR), 这些文件中包含有用户可能需要的自定义扩展等。

--config 文件目录 这个命令允许用户覆盖 \$HIVE_HOME/conf 中默认的属性配置, 而指向一个新的配置文件目录。

2.7 命令行界面

命令行界面, 也就是 CLI, 是和 Hive 交互的最常用的方式。使用 CLI, 用户可以创建表, 检查模式以及查询表, 等等。

2.7.1 CLI 选项

下面这个命令显示了 CLI 所提供的选项列表。这里显示的是 Hive v0.8.* 和 Hive v0.9.* 版本的输出:

```
$ hive --help --service cli
usage: hive
  -d,--define <key=value>      Variable substitution to apply to hive
                                commands. e.g. -d A=B or --define A=B
  -e <quoted-query-string>     SQL from command line
  -f <filename>                 SQL from files
  -H,--help                     Print help information
  -h <hostname>                 connecting to Hive Server on remote host
                                --hiveconf <property=value> Use value for given property
                                --hivevar <key=value>      Variable substitution to apply to hive
                                commands. e.g. --hivevar A=B
```

```

-i <filename>           Initialization SQL file
-p <port>               connecting to Hive Server on port number
-S,--silent            Silent mode in interactive shell
-v,--verbose           Verbose mode (echo executed SQL to the
                       console)

```

这个命令的一个简化版表示方式是 `hive -h`。然而从技术上来说并不支持这个选项。这个命令会输出帮助信息，同时还输出一条提示信息提示“选项 h 缺少参数”。

对于 Hive v0.7.* 版本，不支持 `-d`、`--hivevar` 和 `-p` 选项。

下面我们来更详细地探讨这些选项。

2.7.2 变量和属性

`--define key=value` 实际上和 `--hivevar key=value` 是等价的。二者都可以让用户在命令行定义用户自定义变量以便在 Hive 脚本中引用，来满足不同情况的执行。这个功能只有 Hive v0.8.0 版本和之后的版本才支持。

当用户使用这个功能时，Hive 会将这些键-值对放到 `hivevar` 命名空间，这样可以和其他 3 种内置命名空间（也就是 `hiveconf`、`system` 和 `env`），进行区分。



提示

变量或者属性是在不同的上下文中使用的术语，但是在大多数情况下它们的功能是相同的。

表 2-3 描述了命名空间选项。

表 2-3 Hive 中变量和属性命名空间

命名空间	使用权限	描述
hivevar	可读/可写	(Hive v0.8.0 以及之后版本) 用户自定义变量
hiveconf	可读/可写	Hive 相关的配置属性
system	可读/可写	Java 定义的配置属性
env	只可读	Shell 环境（例如 bash）定义的环境变量

Hive 变量内部是以 Java 字符串的方式存储的。用户可以在查询中引用变量。Hive 会先使用变量值替换掉查询的变量引用，然后才会将查询语句提交给查询处理器。

在 CLI 中，可以使用 `SET` 命令显示或者修改变量值。例如，下面这个会话先显示一个变量的值，然后再显示 `env` 命名空间中定义的所有变量！为了更清晰地表现，我们省略掉了这个 Hive 会话中的一些输出信息，而且在每行命令之间人为地增加了一个空白行：

```

$ hive
hive>set env:HOME;
env:HOME=/home/thisuser

hive>set;
... 非常多的输出信息，而且包含有下面这些变量的：
hive.stats.retries.wait=3000
env:TERM=xterm
system:user.timezone=America/New_York
...

hive>set -v;
... 更多的输出信息!...

```

如果不加-v 标记，set 命令会打印出命名空间 hivevar， hiveconf， system 和 env 中所有的变量。使用-v 标记，则还会打印 Hadoop 中所定义的所有属性，例如控制 HDFS 和 MapReduce 的属性。

set 命令还可用于给变量赋新的值。我们特别看下 hivevar 命名空间以及如何通过命令行定义一个变量：

```

$ hive --define foo=bar
hive> set foo;
foo=bar;

hive> set hivevar:foo;
hivevar:foo=bar;

hive> set hivevar:foo=bar2;

hive> set foo;
foo=bar2

hive> set hivevar:foo;
hivevar:foo=bar2

```

我们可以看到，前缀 hivevar:是可选的。--hivevar 标记和--define 标记是相同的。

在CLI中查询语句中的变量引用会先被替换掉然后才会提交给查询处理器。思考如下这个CLI会话（在v0.8.*版本中使用）：

```

hive> create table toss1(i int, ${hivevar:foo} string);

hive> describe toss1;
i      int
bar2   string

hive> create table toss2(i2 int, ${foo} string);

```

```
hive> describe toss2;
i2      int
bar2    string

hive> drop table toss1;
hive> drop table toss2;
```

我们来看看--hiveconf 选项，Hive v0.7.*版本支持这个功能，其用于配置 Hive 行为的所有属性。我们用它来指定 Hive v0.8.0 版本中增加的 hive.cli.print.current.db 属性。开启这个属性可以在 CLI 提示符前打印出当前所在的数据库名（第 4.1 节“Hive 中的数据库”中有关于 Hive 数据库的更多信息），默认的数据库名为 default。这个属性的默认值是 false。

```
$ hive --hiveconf hive.cli.print.current.db=true
hive (default)> set hive.cli.print.current.db;
hive.cli.print.current.db=true

hive (default)> set hiveconf:hive.cli.print.current.db;
hiveconf:hive.cli.print.current.db=true

hive (default)> set hiveconf:hive.cli.print.current.db=false;

hive> set hiveconf:hive.cli.print.current.db=true;

hive (default)> ...
```

我们甚至可以增加新的 hiveconf 属性，这个功能只有 Hive v0.8.0 版本前的版本才支持：

```
$ hive --hiveconf y=5
hive> set y;
y=5

hive> CREATE TABLE whatsit(i int);

hive> ... 装载数据到表 whatsit 中 ...

hive> SELECT * FROM whatsit WHERE i = ${hiveconf:y};
...
```

我们还有必要了解一下system命名空间，Java系统属性对这个命名空间内容具有可读可写权利；而env命名空间，对于环境变量只提供可读权限：

```
hive> set system:user.name;
system:user.name=myusername

hive> set system:user.name=yourusername;
```

```
hive> set system:user.name;
system:user.name=yourusername

hive> set env:HOME;
env:HOME=/home/yourusername

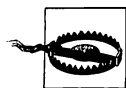
hive> set env:HOME;
env:* variables can not be set.
```

和 hivevar 变量不同，用户必须使用 `system:` 或者 `env:` 前缀来指定系统属性和环境变量。

`env` 命名空间可作为向 Hive 传递变量的一个可选的方式，特别是对于 Hive v0.7.* 版本。考虑如下这个例子：

```
$ YEAR=2012 hive -e "SELECT * FROM mytable WHERE year = ${env:YEAR}";
```

查询处理器会在 WHERE 子句中查看到实际的变量值 2012。



警告

如果你现在使用的是 Hive v0.7.* 版本，那么这本书中一些使用参数和变量的例子和写的可能并不符合。如果有这种情况，那么将变量替换成具体的值即可。



提示

Hive 中所有的内置属性都在 `$HIVE_HOME/conf/hivedefault.xml.template` 中列举出来了，这是个“样例”配置文件。配置文件中还说明了这些属性的默认值。

2.7.3 Hive 中“一次使用”命令

用户可能有时期望执行一个或者多个查询（使用分号分隔），执行结束后 hive CLI 立即退出。Hive 提供了这样的功能，因为 CLI 可以接受 `-e` 命令这种形式。如果表 `mytable` 具有一个字符串字段和一个整型字段，我们可以看到如下输出：

```
$ hive -e "SELECT * FROM mytable LIMIT 3";
OK
name1 10
name2 20
name3 30
Time taken: 4.955 seconds
$
```

临时应急时可以使用这个功能将查询结果保存到一个文件中。增加 `-S` 选项可以开启静

默模式，这样可以在输出结果中去掉“OK”和“Time taken”等行，以及其他一些无关紧要的输出信息，如下面这个例子：

```
$ hive -S -e "select * FROM mytable LIMIT 3" > /tmp/myquery
$ cat /tmp/myquery
name1 10
name2 20
name3 30
```

需要注意的是，Hive 会将输出写到标准输出中。上面例子中的 shell 命令将输出重定向到本地文件系统中，而不是 HDFS 中。

最后，当用户不能完整记清楚某个属性名时，可以使用下面这个非常有用的技巧来模糊获取这个属性名而无需滚动 set 命令的输出结果进行查找。假设用户没记清哪个属性指定了管理表的“warehouse(数据仓库)”的路径，通过如下命令可以查看到：

```
$ hive -S -e "set" | grep warehouse
hive.metastore.warehouse.dir=/user/hive/warehouse
hive.warehouse.subdir.inherit.perms=false
```

这是第一种情况。

2.7.4 从文件中执行 Hive 查询

Hive 中可以使用 -f 文件名方式执行指定文件中的一个或者多个查询语句。按照惯例，一般把这些 Hive 查询文件保存为具有 .q 或者 .hql 后缀名的文件。

```
$ hive -f /path/to/file/withqueries.hql
```

在 Hive shell 中用户可以使用 SOURCE 命令来执行一个脚本文件。下面是一个例子：

```
$ cat /path/to/file/withqueries.hql
SELECT x.* FROM src x;
$ hive
hive> source /path/to/file/withqueries.hql;
...
```

顺便说一下，如果查询中的表名和这个例子并不相关，我们有时会使用 src（代表“源表”）作为表名。这个约定来源于 Hive 源码中的单元测试中的写法。它会在所有测试开始前先创建一个名为 src 的表。

例如，在试用某个内置函数时，通常会写个“查询”语句，然后往这个函数中传入参数，如下面这个例子所示（这个例子在第 15.9 节《XPath 相关的函数》中会讲到）：

```
hive> SELECT xpath('\<a><b id="foo">b1</b><b id="bar">b2</b></a>','\//@id\')
> FROM src LIMIT 1;
[foo,"bar]
```

我们现在不必关心 xpath 的细节问题，但是需要注意，我们向 xpath 中传递了 2 个字符串类型的参数，而且使用了 FROM src LIMIT 1 来指定必须要有 FROM 子句，并限制了输出行数。src 代表一个已经创建好的或者虚拟地创建了的名为 src 的表：

```
CREATE TABLE src(s STRING);
```

同时必须至少有一行的数据在源表里面：

```
$ echo "one row" > /tmp/myfile
$ hive -e "LOAD DATA LOCAL INPATH '/tmp/myfile' INTO TABLE src;
```

2.7.5 hiverc 文件

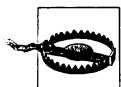
我们最后将要讨论的 CLI 选项是 -i 文件名。这个选项允许用户指定一个文件，当 CLI 启动时，在提示符出现前会先执行这个文件。Hive 会自动在 HOME 目录下寻找名为 .hiverc 的文件，而且会自动执行这个文件中的命令（如果文件中有的话）。

对于用户需要频繁执行的命令，使用这个文件是非常方便的。例如设置系统属性（参考第 2.7.2 节“变量和属性”），或者增加对于 Hadoop 的分布式内存（在第 15 章将会有介绍）进行自定义的 Hive 扩展的 Java 包（JAR 文件）。

下面的例子显示的是一个典型的 \$HOME/.hiverc 文件中的内容：

```
ADD JAR /path/to/custom_hive_extensions.jar;
set hive.cli.print.current.db=true;
set hive.exec.mode.local.auto=true;
```

上面例子第 1 行表示向 Hadoop 分布式内存中增加一个 JAR 文件。第 2 行表示修改 CLI 提示符前显示当前所在的工作数据库，这个我们在前面第 2.7.2 节“变量和属性”中讲述过了。最后 1 行表示“鼓励”Hive 如果可以使用本地模式执行（即使当 Hadoop 是以分布式模式或伪分布式模式执行时）的话就在本地执行，这样可以加快小数据集的数据查询速度。



警告

一个比较容易犯的错误就是忘记在每行的结尾加逗号。如果用户犯了这个错误，那么这个属性将包含后面几行所有的文字，直到发现下一个逗号。

2.7.6 使用 Hive CLI 的更多介绍

CLI 支持其他一些有用的功能。

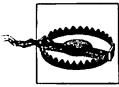
自动补全功能

如果用户在输入的过程中敲击 Tab 制表键，那么 CLI 会自动补全可能的关键字或者函

数名。例如，如果用户输入 SELE 然后按 Tab 键，CLI 将自动补全这个词为 SELECT。

如果用户在提示符后面直接敲击 Tab 键，那么用户会看到如下回复：

```
hive>
Display all 407 possibilities? (y or n)
```



警告

当向 CLI 中输入语句时，如果某些行是以 Tab 键开头的话，就会产生一个常见的令人困惑的错误。用户这时会看到一个“是否显示所有可能的情况”的提示，而且输入流后面的字符会被认为是对这个提示的回复，也因此会导致命令执行失败。

2.7.7 查看操作命令历史

用户可以使用上下箭头来滚动查看之前的命令。事实上，每一行之前的输入都是单独显示的，CLI 不会把多行命令和查询作为一个单独的历史条目。Hive 会将最近的 100, 00 行命令记录到文件 \$HOME/.hivehistory 中。

如果用户想再次执行之前执行过的某条命令，只需要将光标滚动到那条记录然后按 Enter 键就可以了。如果用户需要修改这行记录后再执行，那么需要使用左右方向键将光标移动到需要修改的地方然后重新编辑修改就可以了。修改后用户直接敲击 Enter 键就可以提交这条命令而无需切换到命令尾。



提示

大多数的导航按键使用的 Ctrl+字母的命令和 bash shell 中是相同的（例如，Control+A 代表光标移到到行首，Control+B 代表光标移动到行尾）。然而，类似的“元操作”Option 或者 Escape 键就不起作用了（例如，Option+F 一次向前移动一个单词这样的命令）。相似地，Delete 删除键会删除光标左边的字符，而 Forward Delete 回格键不会删除掉光标当前所在的字符。

2.7.8 执行 shell 命令

用户不需要退出 hive CLI 就可以执行简单的 bash shell 命令。只要在命令前加上 ! 并且以分号 (;) 结尾就可以：

```
hive> ! /bin/echo "what up dog";
"what up dog"
hive> ! pwd;
/home/me/hiveplay
```

Hive CLI 中不能使用需要用户进行输入的交互式命令，而且不支持 shell 的“管道”功能和文件名的自动补全功能。例如，`! ls *.hql`；这个命令表示的是查找文件名为 `*.hql` 的文件，而不是表示显示以 `.hql` 结尾的所有文件。

2.7.9 在 Hive 内使用 Hadoop 的 dfs 命令

用户可以在 Hive CLI 中执行 Hadoop 的 `dfs ...` 命令，只需要将 `hadoop` 命令中的关键字 `hadoop` 去掉，然后以分号结尾就可以了：

```
hive> dfs -ls / ;
Found 3 items
drwxr-xr-x - root supergroup 0 2011-08-17 16:27 /etl
drwxr-xr-x - edward supergroup 0 2012-01-18 15:51 /flag
drwxrwxr-x - hadoop supergroup 0 2010-02-03 17:50 /users
```

这种使用 `hadoop` 命令的方式实际上比与其等价的在 `bash shell` 中执行的 `hadoop dfs...` 命令要更高效。因为后者每次都会启动一个新的 JVM 实例，而 Hive 会在同一个进程中执行这些命令。

用户可以通过如下命令查看 `dfs` 所提供的所有功能选项列表：

```
hive> dfs -help;
```

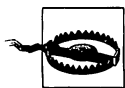
用户还可以登录网址 http://hadoop.apache.org/common/docs/r0.20.205.0/file_system_shell.html 或者是查看用户所使用的 Hadoop 发行版的文档来获取这些命令的说明。

2.7.10 Hive 脚本中如何进行注释

对于 Hive v0.8.0 版本，用户可以使用以 `--` 开头的字符串来表示注释，例如：

```
-- Copyright (c) 2012 Megacorp, LLC.
-- This is the best Hive script ever!!

SELECT * FROM massive_table;
...
```



警告

CLI 是不会解析这些注释行的。因此如果用户在 CLI 中粘贴这些注释语句，那么将会有错误信息。他们只能放在脚本中通过 `hive -f script_name` 的方式执行。

2.7.11 显示字段名称

作为最后一个例子，我们把学到的很多东西都放在了一起。我们可以让 CLI 打印出字段名称（这个功能默认是关闭的）。我们可以通过设置 `hiveconf` 配置项 `hive.cli.print.header` 为 `true` 来开启这个功能：

```
hive> set hive.cli.print.header=true;
```

```
hive> SELECT * FROM system_logs LIMIT 3;
```

```
tstamp severity server message
```

```
1335667117.337715 ERROR server1 Hard drive hdl is 90% full!
```

```
1335667117.338012 WARN server1 Slow response from server2.
```

```
1335667117.339234 WARN server2 Uh, Dude, I'm kinda busy right now...
```

如果用户希望总是看到字段名称, 那么只需要将第 1 行添加到\$HOME/.hiverc 文件中即可。

数据类型和文件格式

Hive 支持关系型数据库中的大多数基本数据类型，同时也支持关系型数据库中很少出现的 3 种集合数据类型，下面我们将简短地介绍一下这样做的原因。

其中一个需要考虑的因素就是这些数据类型是如何在文本文件中进行表示的，同时还要考虑文本存储中为了解决各种性能问题以及其他问题有哪些替代方案。和大多数的数据库相比，Hive 具有一个独特的功能，那就是其对于数据在文件中的编码方式具有非常大的灵活性。大多数的数据库对数据具有完全的控制，这种控制既包括对数据存储到磁盘的过程的控制，也包括对数据生命周期的控制。Hive 将这些方面的控制权转交给用户，以便更加容易地使用各种各样的工具来管理和处理数据。

3.1 基本数据类型

Hive 支持多种不同长度的整型和浮点型数据类型，支持布尔类型，也支持无长度限制的字符串类型。Hive v0.8.0 版本中增加了时间戳数据类型和二进制数组数据类型。

表 3-1 列举了 Hive 所支持的基本数据类型。

表 3-1 基本数据类型

数据类型	长度	例子
TINYINT	1byte 有符号整数	20
SMALINT	2byte 有符号整数	20
INT	4byte 有符号整数	20
BIGINT	8byte 有符号整数	20
BOOLEAN	布尔类型，true 或者 false	TRUE
FLOAT	单精度浮点数	3.14159

数据类型	长度	例子
DOUBLE	双精度浮点数	3.14159
STRING	字符序列。可以指定字符集。可以使用单引号或者双引号	'now is the time', "for all good men"
TIMESTAMP(v0.8.0+)	整数，浮点数或者字符串	1327882394 (Unix 新纪元秒), 1327882394.123456789 (Unix 新纪元秒并跟随有纳秒数) 和 '2012-02-03 12:34:56.123456789' (JDBC 所兼容的 java.sql.Timestamp 时间格式)
BINARY(v0.8.0+)	字节数组	请看后面的讨论

和其他 SQL 方言一样，这些都是保留字。

需要注意的是所有的这些数据类型都是对 Java 中的接口的实现，因此这些类型的具体行为细节和 Java 中对应的类型是完全一致的。例如，STRING 类型实现的是 Java 中的 String，FLOAT 实现的是 Java 中的 float，等等。

在其他 SQL 方言中，通常会提供限制最大长度的“字符数组”（也就是很多字符串）类型，但需要注意的是 Hive 中不支持这种数据类型。关系型数据库提供这个功能是出于性能优化的考虑。因为定长的记录更容易进行建立索引，数据扫描，等等。在 Hive 所处的“宽松”的世界里，不一定拥有数据文件但必须能够支持使用不同的文件格式，Hive 根据不同字段间的分隔符来对其进行判断。同时，Hadoop 和 Hive 强调优化磁盘的读和写的性能，而限制列的值的长度相对来说并不重要。

新增数据类型 TIMESTAMP 的值可以是整数，也就是距离 Unix 新纪元时间（1970 年 1 月 1 日，午夜 12 点）的秒数；也可以是浮点数，即距离 Unix 新纪元时间的秒数，精确到纳秒（小数点后保留 9 位数）；还可以是字符串，即 JDBC 所约定的时间字符串格式，格式为 YYYY-MM-DD hh:mm:ss.fffffffff。

TIMESTAMPS 表示的是 UTC 时间。Hive 本身提供了不同时区间互相转换的内置函数，也就是 to_utc_timestamp 函数和 from_utc_timestamp 函数(详情请查看第 13 章内容)。

BINARY 数据类型和很多关系型数据库中的 VARBINARY 数据类型是类似的，但和 BLOB 数据类型并不相同。因为 BINARY 的列是存储在记录中的，而 BLOB 则不同。BINARY 可以在记录中包含任意字节，这样可以防止 Hive 尝试将其作为数字，字符串等进行解析。

需要注意的是如果用户的目标是省略掉每行记录的尾部的话，那么是无需使用 BINARY 数据类型的。如果一个表的表结构指定的是 3 列，而实际数据文件每行记录包含有 5

个字段的话，那么在 Hive 中最后 2 列数据将会被省略掉。

如果用户在查询中将一个 float 类型的列和一个 double 类型的列作对比或者将一种整型类型的值和另一种整型类型的值做对比，那么结果将会怎么样呢？Hive 会隐式地将类型转换为两个整型类型中值较大的那个类型，也就是会将 FLOAT 类型转换为 DOUBLE 类型，而且如有必要，也会将任意的整型类型转换为 DOUBLE 类型，因此事实上是同类型之间的比较。

如果用户希望将一个字符串类型的列转换为数值呢？这种情况下用户可以显式地将一种数据类型转换为其他一种数据类型，后面会有这样的一个例子，例子中 s 是一个字符串类型列，其值为数值：

```
...cast(s AS INT) ...;
```

(这里需要说明的是，AS INT 是关键字，因此使用小写也是可以的。)

我们将会在第 6.8 节“类型转换”更深入地讨论数据类型转换。

3.2 集合数据类型

Hive 中的列支持使用 struct, map 和 array 集合数据类型。需要注意的是表 3-2 中语法示例实际上调用的是内置函数。

表 3-2 集合数据类型

数据类型	描述	字面语法示例
STRUCT	和 C 语言中的 struct 或者“对象”类似，都可以通过“点”符号访问元素内容。例如，如果某个列的数据类型是 STRUCT{first STRING, last STRING}，那么第 1 个元素可以通过 字段名.first 来引用	struct('John', 'Doe')
MAP	MAP 是一组键-值对元组集合，使用数组表示法（例如[key]）可以访问元素。例如，如果某个列的数据类型是 MAP，其中键->值对是 'first' -> 'John' 和 'last' -> 'Doe'，那么可以通过 字段名['last'] 获取最后 1 个元素	map('first', 'JOHN', 'last', 'Doe')
ARRAY	数组是一组具有相同类型和名称的变量的集合。这些变量称为数组的元素，每个数组元素都有一个编号，编号从零开始。例如，数组值为['John', 'Doe']，那么第 2 个元素可以通过 数组名[1] 进行引用	Array('John', 'Doe')

和基本数据类型一样，这些类型的名称同样是保留字。

大多数的关系型数据库并不支持这些集合数据类型，因为使用它们会趋向于破坏标准

格式。例如，在传统数据模型中，structs 可能需要由多个不同的表拼装而成，表间需要适当地使用外键来进行连接。

破坏标准格式所带来的一个实际问题是会增大数据冗余的风险，进而导致消耗不必要的磁盘空间，还有可能造成数据不一致，因为当数据发生改变时冗余的拷贝数据可能无法进行相应的同步。

然而，在大数据系统中，不遵循标准格式的一个好处就是可以提供更高吞吐量的数据。当处理的数据的数量级是 T 或者 P 时，以最少的“头部寻址”来从磁盘上扫描数据是非常必要的。按数据集进行封装的话可以通过减少寻址次数来提供查询的速度。而如果根据外键关系关联的话则需要进行磁盘间的寻址操作，这样会有非常高的性能消耗。



提示

Hive 中并没有键的概念。但是，用户可以对表建立索引，这些我们在第 7 章将会进行介绍。

这里有一个用于演示如何使用这些数据类型的表结构声明语句，这是一张虚构的人力资源应用程序中的员工表：

```
CREATE TABLE employees (  
    name          STRING,  
    salary        FLOAT,  
    subordinates  ARRAY<STRING>,  
    deductions    MAP<STRING, FLOAT>,  
    address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>;
```

其中，name 是一个简单的字符串；对于大多数雇员来说，salary（薪水）使用 float 浮点数类型来表示就已经足够了；subordinates（下属员工）列表是一个字符串值数组。在该数组中，我们可以认为 name 是“主键”，因此 subordinates 中的每一个元素都将会引用这张表中的另一条记录。对于没有下属的雇员，这个字段对应的值就是一个空的数组。在传统的模型中，将会以另外一种方式来表示这种关系，也就是雇员和雇员的经理这种对应关系。这里我们并非强调我们的模型对于 Hive 来说是最好的，而只是为了举例展示如何使用数组。

字段 deductions 是一个由键-值对构成的 map，其记录了每一次的扣除额，这些钱将会在发薪水的时候从员工工资中扣除掉。map 中的键是扣除金额项目的名称（例如，“国家税收”），而且键可以是一个百分比值，也可以完全就是一个数值。在传统数据模型中，这个扣除额项目的名称（这里也就是 map 的键）可能存在于不同的表中。这些表在存放特定扣除额值的同时，还有一个外键指向对应的雇员记录。

最后，每名雇员的家庭住址使用 struct 数据类型存储，其中的每个域都被作了命名，并且具有一个特定的类型。

请注意后面是如何使用 Java 语法惯例来表示集合数据类型的。例如，MAP<STRING, FLOAT>表示 map 中的每个键都是 STRING 数据类型的，而每个值都是 FLOAT 数据类型的。对于 ARRAY<STRING>，其中的每个条目都是 STRING 类型的。STRUCT 可以混合多种不同的数据类型，但是 STRUCT 中一旦声明好结构，那么其位置就不可再改变。

3.3 文本文件数据编码

下面我们一起来研究文件格式，首先举个最简单的例子，也就是文本格式文件。毫无疑问，用户应该很熟悉以逗号或者制表符分割的文本文件，也就是所谓的逗号分隔值（CSV）或者制表符分割值（TSV）。只要用户需要，Hive 是支持这些文件格式的，在后面将会介绍其具体使用方式。然而，这两种文件格式有一个共同的缺点，那就是用户需要对文本文件中那些不需要作为分隔符处理的逗号或者制表符格外小心。也因此，Hive 默认使用了几个控制字符，这些字符很少出现在字段值中。Hive 使用术语 field 来表示替换默认分隔符的字符，稍后我们将可以看到。表 3-3 列举了这些分隔符。

表 3-3 Hive 中默认的记录和字段分隔符

分隔符	描述
\n	对于文本文件来说，每行都是一条记录，因此换行符可以分割记录
^A (Ctrl+A)	用于分隔字段（列）。在 CREATE TABLE 语句中可以使用八进制编码\001 表示
^B	用于分隔 ARRAY 或者 STRUCT 中的元素，或用于 MAP 中键-值对之间的分隔。在 CREATE TABLE 语句中可以使用八进制编码\002 表示
^C	用于 MAP 中键和值之间的分隔。在 CREATE TABLE 语句中可以使用八进制编码\003 表示

前面章节中介绍的 employees 表的记录看上去和下面展示的这个例子是一样的，其中使用到了 ^A 等字符来作为字段分隔符。像 Emacs 这样的文本编辑器将会以这种形式来展示这些分隔符。因为页面篇幅有限，所以每行数据并非显示在同一行。为了能较清晰地展示每行记录，我们在行与行间额外增加了一个空行：

```
John Doe^A10000.0^AMary Smith^BTodd Jones^AFederal Taxes^C.2^BStateTaxes^C.05^BInsurance^C.1^A1 Michigan Ave.^BChicago^BIL^B60600
```

```
Mary Smith^A80000.0^ABill King^AFederal Taxes^C.2^BState Taxes^C.05^BInsurance^C.1^A100 Ontario St.^BChicago^BIL^B60601
```

```
Todd Jones^A70000.0^AFederal Taxes^C.15^BState Taxes^C.03^BInsurance^C.1^A200 Chicago Ave.^BOak Park^BIL^B60700
```

Bill King^A60000.0^AFederal Taxes^C.15^BState Taxes^C.03^BInsurance^C.1^A300
Obscure Dr.^BObscuria^BIL^B60100

其可读性并不好，但是当然了，我们可以使用 Hive 来读取这些数据。

我们先来看看第 1 行记录来了解一下这个结构。首先，如果使用 JavaScript 数据交换格式 (JSON) 来表示这条记录的话，那么如果增加了嵌套结构中的字段名称的话，样式将会和如下所示的一样：

```
{
  "name": "John Doe",
  "salary": 100000.0,
  "subordinates": ["Mary Smith", "Todd Jones"],
  "deductions": {
    "Federal Taxes": .2,
    "State Taxes": .05,
    "Insurance": .1
  },
  "address": {
    "street": "1 Michigan Ave.",
    "city": "Chicago",
    "state": "IL",
    "zip": 60600
  }
}
```

这里，用户可能会发现在 JSON 中 map 类型和 struct 类型其实是一样的。

那么，我们来看看文本文件的第 1 行分解出来的结果。

- ① John Doe 对应 name 字段，表示用户名。
- ② 100000.0 对应 salary 字段，表示薪水。
- ③ Mary Smith^BTodd Jones 对应 subordinates 字段，表示下属是“Mary Smith”和“Todd Jones”。
- ④ Federal Taxes^C.2^BState Taxes^C.05^BInsurance^C.1 对应 deductions 字段，表示扣除的金额，其中 20%用于上缴“国税”，5%用于上缴“州税”，还有 10%用于上缴“保险费。”
- ⑤ 1 Michigan Ave.^BChicago^BIL^B60600 对应 address 字段，表示住址是“芝加哥第一密歇根大道 60600 号。”

用户可以不使用这些默认的分隔符，而指定使用其他分隔符。当有其他应用程序使用不同的规则写数据时，这是非常必要的。下面这个表结构声明和之前的那个表是一样的，不过这里明确地指定了分隔符：

```
CREATE TABLE employees (
```

```

name          STRING,
salary        FLOAT,
subordinates  ARRAY<STRING>,
deductions    MAP<STRING, FLOAT>,
address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

```

ROW FORMAT DELIMITED 这组关键字必须要写在其他子句（除了 STORED AS ... 子句）之前。

字符\001 是^A 的八进制数。ROW FORMAT DELIMITED FIELDS TERMINATED BY '\001'这个子句表明 Hive 将使用^A 字符作为列分割符。

同样地，字符\002 是^B 的八进制数。ROW FORMAT DELIMITED COLLECTION ITEMS TERMINATED BY '\002'这个子句表明 Hive 将使用^B 作为集合元素间的分隔符。

最后，字符\003 是^C 的八进制数。ROW FORMAT DELIMITED MAP KEYS TERMINATED BY '\003'这个子句表明 Hive 将使用^C 作为 map 的键和值之间的分隔符。

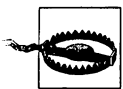
子句 LINES TERMINATED BY '...'和 STORED AS ...不需要 ROW FORMAT DELIMITED 关键字。

事实上，Hive 到目前为止对于 LINES TERMINATED BY ...仅支持字符 '\n'，也就是说行与行之间的分隔符只能为 '\n'。因此这个子句现在使用起来还是有限制的。

用户可以重新指定列分割符及集合元素间分隔符，而 map 中键-值间分隔符仍然使用默认的文本文件格式，因此子句 STORED AS TEXTFILE 很少被使用到。本书中大多数情况下，我们使用的都是缺省情况下默认的 TEXTFILE 文件格式。

当然 Hive 还支持其他一些文件格式，但是我们将其推迟到第 15 章再进行论述。与其相关的一个话题是文件的压缩，这个我们将在第 11 章进行讨论。

因此，虽然用户可以明确指定这些子句，但是在大多数情况下，大多子句还是使用默认的分隔符的，只需要明确指定那些需要替换的分隔符即可。



警告

这些规则只会影响到 Hive 在读取文件后将如何进行划分。只有在一些比较极端的情况下，才需要用户手工去将数据按正确的格式写入文件。

例如，如下表结构声明定义的是一个表数据按照逗号进行分割的表。

```
CREATE TABLE some_data (  
    first FLOAT,  
    second FLOAT,  
    third FLOAT  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ',';
```

用户还可以使用 ‘\t’（也就是制表键）作为字段分隔符。



提示

本例并没有合理地处理 CSV 格式（值是逗号分隔的）和 TSV 格式（值是制表键分隔的）文件通常使用的场景。CSV 和 TSV 格式的文件中文件头中包含有列名，而列值字符串也有可能是包含在引号中的，而且其列值中也有可能包含有逗号或者制表键。第 15 章将介绍如果更优雅地处理这类文件。

这种强大的可定制功能使得可以很容易地使用 Hive 来处理那些由其他工具和各种各样的 ETL（也就是数据抽取、数据转换和数据装载过程）程序产生的文件。

3.4 读时模式

当用户向传统数据库中写入数据的时候，不管是采用装载外部数据的方式，还是采用将一个查询的输出结果写入的方式，或者是使用 UPDATE 语句，等等，数据库对于存储都具有完全的控制力。数据库就是“守门人”。传统数据库是写时模式（schema on write），即数据在写入数据库时对模式进行检查。

Hive 对底层存储并没有这样的控制。对于 Hive 要查询的数据，有很多种方式对其进行创建、修改，甚至损坏。因此，Hive 不会在数据加载时进行验证，而是在查询时进行，也就是读时模式（schema on read）。

那么如果模式和文件内容并不匹配将会怎么样呢？Hive 对此做的非常好，因为其可以读取这些数据。如果每行记录中的字段个数少于对应的模式中定义的字段个数的话，那么用户将会看到查询结果中有很多的 null 值。如果某些字段是数值型的，但是 Hive 在读取时发现存在非数值型的字符串值的话，那么对于那些字段将会返回 null 值。除此之外的其他情况下，Hive 都极力尝试尽可能地将各种错误恢复过来。

HiveQL：数据定义

HiveQL 是 Hive 查询语言。和普遍使用的所有 SQL 方言一样，它不完全遵守任一种 ANSI SQL 标准的修订版。HiveQL 可能和 MySQL 的方言最接近，但是两者还是存在显著性差异的。Hive 不支持行级插入操作、更新操作和删除操作。Hive 也不支持事务。Hive 增加了在 Hadoop 背景下的可以提供更高性能的扩展，以及一些个性化的扩展，甚至还增加了一些外部程序。

当然了，大部分的 HiveQL 还是很常见的。本章以及随后的几章将会使用一些典型的例子来讲解 HiveQL 的那些特性。在某些情况下，我们会从整体上简要地谈到一些细节，然后会在后面的章节里再去比较完整地进行讨论。

本章开始的部分是 HiveQL 所谓的数据定义语言部分，其用于创建、修改和删除数据库、表、视图、函数和索引。本章将会涉及到数据库和表，第 7 章将会讨论视图有关内容，第 8 章将讨论索引，第 13 章会讲述到函数。

随着介绍的继续，我们也将讨论 SHOW 和 DESCRIBE 这些用于列举和描述信息的命令。

随后的几章将会研究 HiveQL 的数据操作语言部分，其用于将数据导入到 Hive 表中，以及将数据抽取到文件系统中。这些章节中还会介绍如何通过查询、分组、过滤、连接等操作研究和操作数据。

4.1 Hive 中的数据库

Hive 中数据库的概念本质上仅仅是表的一个目录或者命名空间。然而，对于具有很多组和用户的大集群来说，这是非常有用的，因为这样可以避免表命名冲突。通常会使用数据库来将生产表组织成逻辑组。

如果用户没有显式指定数据库，那么将会使用默认的数据库 default。

下面这个例子就展示了如何创建一个数据库：

```
hive> CREATE DATABASE financials;
```

如果数据库 financials 已经存在的话，那么将会抛出一个错误信息。使用如下语句可以避免在这种情况下抛出错误信息：

```
hive> CREATE DATABASE IF NOT EXISTS financials;
```

虽然通常情况下用户还是期望在同名数据库已经存在的情况下能够抛出警告信息的，但是 IF NOT EXISTS 这个子句对于那些在继续执行之前需要根据需要实时创建数据库的情况来说是非常有用的。

在所有的数据库相关的命令中，都可以使用 SCHEMA 这个关键字来替代关键字 TABLE。

随时可以通过如下命令方式查看 Hive 中所包含的数据库：

```
hive> SHOW DATABASES;
default
financials

hive> CREATE DATABASE human_resources;

hive> SHOW DATABASES;
default
financials
human_resources
```

如果数据库非常多的话，那么可以使用正则表达式匹配来筛选出需要的数据库名，正则表达式这个概念，将会在第 6.2.3 节“Like 和 RLike”介绍。下面这个例子展示的是列举出所有以字母 h 开头，以其他字符结尾（即.*部分含义）的数据库名：

```
hive> SHOW DATABASES LIKE 'h.*';
human_resources
hive> ...
```

Hive 会为每个数据库创建一个目录。数据库中的表将会以这个数据库目录的子目录形式存储。有一个例外就是 default 数据库中的表，因为这个数据库本身没有自己的目录。

数据库所在的目录位于属性 hive.metastore.warehouse.dir 所指定的顶层目录之后，这个配置项我们已经在前面的第 2.5.1 节“本地模式配置”和第 2.5.2 节“分布式模式和伪分布式模式配置”中进行了介绍。假设用户使用的是这个配置项默认的配置，也就是 /user/hive/warehouse，那么当我们创建数据库 financials 时，Hive 将会对应地创建一个目录/user/hive/warehouse/financials.db。这里请注意，数据库的文件目录名是以.db 结

尾的。

用户可以通过如下的命令来修改这个默认的位置：

```
hive> CREATE DATABASE financials
      > LOCATION '/my/preferred/directory';
```

用户也可以为这个数据库增加一个描述信息，这样通过 DESCRIBE DATABASE <database> 命令就可以查看到该信息。

```
hive> CREATE DATABASE financials
      > COMMENT 'Holds all financial tables';

hive> DESCRIBE DATABASE financials;
financials Holds all financial tables
hdfs://master-server/user/hive/warehouse/financials.db
```

从上面的例子中，我们可以注意到，DESCRIBE DATABASE 语句也会显示出这个数据库所在的文件目录位置路径。在这个例子中，URI 格式是 hdfs。如果安装的是 MapR，那么这里就应该是 maprfs。对于亚马逊弹性 MapReduce (EMR) 集群，这里应该是 hdfs，但是用户可以设置 hive.metastore.warehouse.dir 为亚马逊 S3 特定的格式（例如，属性值设置为 s3n://bucketname...）。用户可以使用 s3 作为模式，但是如果使用新版的规则 s3n 会更好。

前面 DESCRIBE DATABASE 语句的输出中，我们使用了 master-server 来代表 URI 权限，也就是说应该是由文件系统的“主节点”（例如，HDFS 中运行 NameNode 服务的那台服务器）的服务器名加上一个可选的端口号构成的（例如，服务器名：端口号这样的格式）。如果用户执行的是伪分布式模式，那么主节点服务器名称就应该是 localhost。对于本地模式，这个路径应该是一个本地路径，例如 file:///user/hive/warehouse/financials.db。

如果这部分信息省略了，那么 Hive 将会使用 Hadoop 配置文件中的配置项 fs.default.name 作为 master-server 所对应的服务器名和端口号，这个配置文件可以在 \$HADOOP_HOME/conf 这个目录下找到。

需要明确的是，hdfs:///user/hive/warehouse/financials.db 和 hdfs://master-server/user/hive/warehouse/financials.db 是等价的，其中 master-server 是主节点的 DNS 名和可选的端口号。

为了保持完整性，当用户指定一个相对路径（例如，some/relative/path）时，对于 HDFS 和 Hive，都会将这个相对路径放到分布式文件系统的指定根目录下（例如，hdfs:///user/<user-name>）。然而，如果用户是在本地模式下执行的话，那么当前的本地工作目录将是 some/relative/path 的父目录。

为了脚本的可移植性，通常会省略掉那个服务器和端口号信息，而只有在涉及到另一个分布式文件系统实例（包括 S3 存储）的时候才会指明该信息。

此外，用户还可以为数据库增加一些和其相关的键-值对属性信息，尽管目前仅有的功能就是提供了一种可以通过 `DESCRIBE DATABASE EXTENDED <database>` 语句显示出这些信息的方式：

```
hive> CREATE DATABASE financials
  > WITH DBPROPERTIES ('creator' = 'Mark Moneybags', 'date' = '2012-01-02');

hive> DESCRIBE DATABASE financials;
financials  hdfs://master-server/user/hive/warehouse/financials.db

hive> DESCRIBE DATABASE EXTENDED financials;
financials  hdfs://master-server/user/hive/warehouse/financials.db
{date=2012-01-02, creator=Mark Moneybags};
```

`USE` 命令用于将某个数据库设置为用户当前的工作数据库，和在文件系统中切换工作目录是一个概念：

```
hive> USE financials;
```

现在，使用像 `SHOW TABLES` 这样的命令就会显示当前这个数据库下所有的表。

不幸的是，并没有一个命令可以让用户查看当前所在的是哪个数据库！幸运的是，在 Hive 中是可以重复使用 `USE...命令` 的，这是因为在 Hive 中并没有嵌套数据库的概念。

可以回想下，在第 2.7.2 节“变量和属性”中提到过，可以通过设置一个属性值来在提示符里面显示当前所在的数据库（Hive v0.8.0 版本以及之后的版本才支持此功能）：

```
hive> set hive.cli.print.current.db=true;

hive (financials)> USE default;

hive (default)> set hive.cli.print.current.db=false;

hive> ...
```

最后，用户可以删除数据库：

```
hive> DROP DATABASE IF EXISTS financials;
```

`IF EXISTS` 子句是可选的，如果加了这个子句，就可以避免因数据库 `financials` 不存在而抛出警告信息。

默认情况下，Hive 是不允许用户删除一个包含有表的数据库的。用户要么先删除数据库中的表，然后再删除数据库；要么在删除命令的最后面上加上关键字 `CASCADE`，这样可以使 Hive 自行先删除数据库中的表：

```
hive> DROP DATABASE IF EXISTS financials CASCADE;
```

如果使用的是 RESTRICT 这个关键字而不是 CASCADE 这个关键字的话，那么就默认情况一样，也就是，如果想删除数据库，那么必须先要删除掉该数据库中的所有表。如果某个数据库被删除了，那么其对应的目录也同时会被删除。

4.2 修改数据库

用户可以使用 ALTER DATABASE 命令为某个数据库的 DBPROPERTIES 设置键-值对属性值，来描述这个数据库的属性信息。数据库的其他元数据信息都是不可更改的，包括数据库名和数据库所在的目录位置：

```
hive> ALTER DATABASE financials SET DBPROPERTIES ('edited-by' = 'Joe Db');
```

没有办法可以删除或者“重置”数据库属性。

4.3 创建表

CREATE TABLE 语句遵从 SQL 语法惯例，但是 Hive 的这个语句中具有显著的功能扩展，使其可以具有更广泛的灵活性。例如，可以定义表的数据文件存储在什么位置、使用什么样的存储格式，等等。前面我们在第 3.3 节“文本文件数据编码”中已经讨论了很多种存储格式，同时在稍后的第 15 章我们将会再次探讨一下更加高级的格式。本节中，我们会讨论其他一些在 CREATE TABLE 语句中可以使用到的选项，下面这个表结构适用于前面我们在第 3.2 节“集合数据类型”中所声明的 employees 表：

```
CREATE TABLE IF NOT EXISTS mydb.employees (  
    name                STRING COMMENT 'Employee name',  
    salary              FLOAT COMMENT 'Employee salary',  
    subordinates        ARRAY<STRING> COMMENT 'Names of subordinates',  
    deductions          MAP<STRING, FLOAT>  
                        COMMENT 'Keys are deductions names, values are percentages',  
    address             STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>  
                        COMMENT 'Home address')  
COMMENT 'Description of the table'  
TBLPROPERTIES ('creator'='me', 'created_at'='2012-01-02 10:00:00', ...)  
LOCATION '/user/hive/warehouse/mydb.db/employees';
```

首先，我们可以注意到，如果用户当前所处的数据库并非目标数据库，那么用户可以在表名前增加一个数据库名来进行指定的，也就是例子中的 mydb。

如果用户增加上可选项 IF NOT EXISTS，那么若表已经存在了，Hive 就会忽略掉后面的执行语句，而且不会有任何提示。在那些第一次执行时需要创建表的脚本中，这么写是非常有用的。

然而，这个语句还有一个用户需要注意的问题。如果用户所指定的表的模式和已经存在的这个表的模式不同的话，Hive 不会为此做出提示。如果用户的意图是使这个表具有重新指定的那个新的模式的话，那么就需要先删除这个表，也就是丢弃之前的数据，然后再重建这张表。用户可以考虑使用一个或多个 ALTER TABLE 语句来修改已经存在的表的结构。第 4.6 节“修改表”中有更详细的信息。



提示

如果用户使用了 IF NOT EXISTS，而且这个已经存在的表和 CREATE TABLE 语句后指定的模式是不同的。Hive 会忽略掉这个差异。

用户可以在字段类型后为每个字段增加一个注释。和数据库一样，用户也可以为这个表本身添加一个注释，还可以自定义一个或多个表属性。大多数情况下，TBLPROPERTIES 的主要作用是按键-值对的格式为表增加额外的文档说明。但是，当我们检查 Hive 和像 DymamoDB（请参考第 17.5 节“DymamoDB”中的内容）这样的数据库间的集成时，我们可以发现 TBLPROPERTIES 还可用作表示关于数据库连接的必要的元数据信息。

Hive 会自动增加两个表属性：一个是 last_modified_by，其保存着最后修改这个表的用户的用户名；另一个是 last_modified_time，其保存着最后一次修改的新纪元时间秒。



提示

Hive v0.10.0 版本中有一个功能增强计划，也就是增加一个 SHOW TBLPROPERTIES table_name 命令，用于列举出某个表的 TBLPROPERTIES 属性信息。

最后，可以看到我们可以根据情况为表中的数据指定一个存储路径（和元数据截然不同的，元数据总是会保存这个路径）。在这个例子中，我们使用的 Hive 将会使用的默认的路径/user/hive/warehouse/mydb.db/employees。其中，/user/hive/warehouse 是默认的“数据仓库”路径地址（这个之前有讨论过），mydb.db 是数据库目录，employees 是表目录。

默认情况下，Hive 总是将创建的表的目录放置在这个表所属的数据库目录之后。不过，default 数据库是个例外，其在/user/hive/warehouse 下并没有对应一个数据库目录。因此 default 数据库中的表目录会直接位于/user/hive/warehouse 目录之后（除了用户明确指定为其他路径）。



提示

为了避免潜在产生混淆的可能性，如果用户不想使用默认的表路径，那么最好是使用外部表。查看第 56 页的“外部表”获得更详细信息。

用户还可以拷贝一张已经存在的表的表模式（而无需拷贝数据）：

```
CREATE TABLE IF NOT EXISTS mydb.employees2
LIKE mydb.employees;
```

这个版本还可以接受可选的 LOCATION 语句，但是注意其他的属性，包括模式，都是不可能重新定义的，这些信息直接从原始表获得。

SHOW TABLES 命令可以列举出所有的表。如果不增加其他参数，那么只会显示当前工作数据库下的表。假设我们已经创建了一些其他表，table1 和 table2，而且我们的工作数据库是 mydb：

```
hive> USE mydb;

hive> SHOW TABLES;
employees
table1
table2
```

即使我们不在那个数据库下，我们还是可以列举指定数据库下的表的：

```
hive> USE default;

hive> SHOW TABLES IN mydb;
employees
table1
table2
```

如果我们有很多的表，那么我们可以使用正则表达式来过滤出所需要的表名。正则表达式这个概念我们将在第 6.2.3 节“LIKE 和 RLIKE”中进行讨论。

```
hive> USE mydb;

hive> SHOW TABLES 'empl.*';
employees
```

Hive 并非支持所有的正则表达式功能。如果用户了解正则表达式的话，最好事先测试下备选的正则表达式是否真正奏效！

单引号中的正则表达式表示的是选择所有以 empl 开头并以其他任意字符（也就是.*部分）结尾的表名。



提示

IN database_name 语句和对表名使用正则表达式两个功能尚不支持同时使用。

我们也可以使用 `DESCRIBE EXTENDED mydb.employees` 命令来查看这个表的详细表结构信息。(如果我们当前所处的工作数据库就是 `mydb` 的话,那么我们可以不加 `mydb` 这个前缀。)下面显示的内容我们进行过格式调整,使之查看起来更加容易,而且我们还省略掉了一些无关紧要的细节,因为我们只关注我们感兴趣的信息:

```
hive> DESCRIBE EXTENDED mydb.employees;
name      string  Employee name
salary    float   Employee salary
subordinates  array<string>  Names of subordinates
deductions  map<string,float> Keys are deductions names, values are
              percentages
address    struct<street:string,city:string,state:string,zip:int> Home
address

Detailed Table Information      Table(tableName:employees, dbName:mydb,
                                owner:me,
...
location:hdfs://master-server/user/hive/warehouse/mydb.db/employees,
parameters:{creator=me, created_at='2012-01-02 10:00:00',
             last_modified_user=me, last_modified_time=1337544510,
             comment:Description of the table, ...}, ...)
```

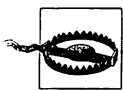
使用 `FORMATTED` 关键字替代 `EXTENDED` 关键字的话,可以提供更加可读的和冗长的输出信息。(译者注:实际情况是使用 `FORMATTED` 要更多些,因为其输出内容详细而且可读性强)

上面输出信息的第一段是和没有使用关键字 `EXTENDED` 或者 `FROMATTED` 时输出的结果一样的(例如,只输出包含有列描述信息的表结构信息)。

如果用户只想查看某一个列的信息,那么只要在表名后增加这个字段的名称即可。这种情况下,使用 `EXTENDED` 关键字也不会增加更多的输出信息:

```
hive> DESCRIBE mydb.employees.salary;
salary float Employee salary
```

回到之前的那个包含详细扩展信息的输出。我们需要特别注意以 `location:` 开头的那行描述信息。这个是 Hive 在 HDFS 中的存储表中数据的完整的 URL 目录路径,这个我们之前有讨论过的。



警告

我们说过 `last_modified_by` 和 `last_modified_time` 两个表属性是会自动创建的。如果没有定义任何的用户自定义表属性的话,那么它们也不会显示在表的详细信息中!

4.3.1 管理表

我们目前所创建的表都是所谓的管理表，有时也被称为内部表。因为这种表，Hive 会（或多或少地）控制着数据的生命周期。正如我们所看见的，Hive 默认情况下会将这些表的数据存储在由配置项 `hive.metastore.warehouse.dir`（例如，`/user/hive/warehouse`）所定义的目录的子目录下。

当我们删除一个管理表时（参考第 4.5 节“删除表”中内容），Hive 也会删除这个表中数据。

但是，管理表不方便和其他工作共享数据。例如，假设我们有一份由 Pig 或者其他工具创建并且主要由这一工具使用的数据，同时我们还想使用 Hive 在这份数据上执行一些查询，可是并没有给予 Hive 对数据的所有权，我们可以创建一个外部表指向这份数据，而并不需要对其具有所有权。

4.3.2 外部表

假设我们正在分析来自股票市场的数据库。我们会定期地从像 Infochimps(<http://infochimps.com/datasets>)这样的数据源接入关于 NASDAQ 和 NYSE 的数据，然后使用很多工具来分析这份数据。（我们可以看到数据集名称分别为 `infochimps_dataset_4777_download_16185` 和 `infochimps_dataset_4778_download_16677`，实际上该数据来源于 Yahoo!财经。）我们后面将要使用的模式和这 2 份源数据都是匹配的。我们假设这些数据文件位于分布式文件系统的 `/data/stocks` 目录下。

下面的语句将创建一个外部表，其可以读取所有位于 `/data/stocks` 目录下的以逗号分隔的数据：

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (  
    exchange      STRING,  
    symbol        STRING,  
    ymd          STRING,  
    price_open   FLOAT,  
    price_high   FLOAT,  
    price_low    FLOAT,  
    price_close  FLOAT,  
    volume       INT,  
    price_adj_close FLOAT)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
LOCATION '/data/stocks';
```

关键字 `EXTENAL` 告诉 Hive 这个表是外部的，而后面的 `LOCATION...` 子句则用于告诉 Hive 数据位于哪个路径下。

因为表是外部的，所以 Hive 并非认为其完全拥有这份数据。因此，删除该表并不会删除掉这份数据，不过描述表的元数据信息会被删除掉。

管理表和外部表有一些小小的区别，那就是，有些 HiveQL 语法结构并不适用于外部表。后面当我们遇到这些问题的时候，我们再来讲述。

然而，我们需要清楚的重要的一点是管理表和外部表之间的差异要比刚开始所看到的小得多。即使对于管理表，用户也是可以知道数据是位于哪个路径下的，因此用户也是可以使用其他工具（例如 hadoop 的 dfs 命令等）来修改甚至删除管理表所在的路径目录下的数据的。可能从严格意义上来说，Hive 是管理着这些目录和文件，但是其并非具有对它们的完全控制权限！回想下，在第 3.4 节“读时模式”中，我们说过，Hive 实际上对于所存储的文件的完整性以及数据内容是否和表模式相一致并没有支配能力，甚至管理表都没有给用户提供这些管理能力。

尽管如此，好的软件设计的一般原则是表达意图。如果数据会被多个工具共享，那么可以创建一个外部表，来明确对数据的所有权。

用户可以在 DESCRIBE EXTENDED tablename 语句的输出中查看到表是否是管理表或外部表。在末尾的详细表信息输出中，对于管理表，用户可以看到如下信息：

```
... tableType:MANAGED_TABLE)
```

对于外部表，用户可以查看到如下信息：

```
... tableType:EXTERNAL_TABLE)
```

对于管理表，用户还可以对一张存在的表进行表结构复制（而不会复制数据）：

```
CREATE EXTERNAL TABLE IF NOT EXISTS mydb.employees3
LIKE mydb.employees
LOCATION '/path/to/data';
```



提示

这里，如果语句中省略掉 EXTERNAL 关键字而且源表是外部表的话，那么生成的新表也将是外部表。如果语句中省略掉 EXTERNAL 关键字而且源表是管理表的话，那么生成的新表也将是管理表。但是，如果语句中包含有 EXTERNAL 关键字而且源表是管理表的话，那么生成的新表将是外部表。即使在这种场景下，LOCATION 子句同样是可选的。

4.4 分区表、管理表

数据分区的一般概念存在已久。其可以有多种形式，但是通常使用分区来水平分散压力，将数据从物理上转移到和使用最频繁的用户更近的地方，以及实现其他目的。

Hive 中有分区表的概念。我们可以看到分区表具有重要的性能优势，而且分区表还可

以将数据以一种符合逻辑的方式进行组织，比如分层存储。

我们首先会讨论下分区管理表。重新来看之前的那张 `employees` 表并假设我们在一个非常大的跨国公司工作。我们的 HR 人员经常会执行一些带 `WHERE` 语句的查询，这样可以将结果限制在某个特定的国家或者某个特定的第一级细分（例如‘美国的州’或者加拿大的省）。（第一级细分是一个存在的术语，例如：http://www.commondatahub.com/state_source.jsp 这里就有使用到。）为简单起见我们将只使用到 `state`（州）。在 `address`（住址）字段中已经重复包含了州信息。这和 `state` 分区是不同的。我们可以在字段 `address` 中删除 `state` 元素。查询中并不会造成模糊不清的问题，因为我们需要通过使用 `address.state` 才能调用到 `address` 中这个元素的值。那么，让我们先按照 `country`（国家）再按照 `state`（州）来对数据进行分区吧：

```
CREATE TABLE employees (
  name          STRING,
  salary        FLOAT,
  subordinates  ARRAY<STRING>,
  deductions    MAP<STRING, FLOAT>,
  address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
PARTITIONED BY (country STRING, state STRING);
```

分区表改变了 Hive 对数据存储的组织方式。如果我們是在 `mydb` 数据库中创建的这个表，那么对于这个表只会有一个 `employees` 目录与之对应：

```
hdfs://master_server/user/hive/warehouse/mydb.db/employees
```

但是，Hive 现在将会创建好可以反映分区结构的子目录。例如：

```
...
.../employees/country=CA/state=AB
.../employees/country=CA/state=BC
...
.../employees/country=US/state=AL
.../employees/country=US/state=AK
...
```

是的，那些是实际的目录名称。州目录下将会包含有零个文件或者多个文件，这些文件中存放着那些州的雇员信息。

分区字段（这个例子中就是 `country` 和 `state`）一旦创建好，表现得就和普通的字段一样。这里有一个已知的异常情况，是由一个 bug 引起的（参考第 6.1.4 节中的“聚合函数”内容）。事实上，除非需要优化查询性能，否则使用这些表的用户不需要关心这些“字段”是否是分区字段。

例如，下面这个查询语句将会查找出在美国伊利诺斯州的所有雇员：

```
SELECT * FROM employees
```



```
WHERE country = 'US' AND state = 'IL';
```

需要注意的是，因为 `country` 和 `state` 的值已经包含在文件目录名称中了，所以也就没有必要将这些值存放到它们目录下的文件中了。事实上，数据只能从这些文件中获得，因此用户需要在表的模式中说明这点，而且这个数据浪费空间。

对数据进行分区，也许最重要的原因就是为了更快地查询。在前面那个将结果范围限制在伊利诺斯州的雇员的查询中，仅仅需要扫描一个目录下的内容即可。即使我们有成千上万个国家和州目录，除了一个目录其他的都可以忽略不计。对于非常大的数据集，分区可以显著地提高查询性能，除非对分区进行常见的范围筛选（例如，按照地理位置范围或按照时间范围等）。

当我们在 `WHERE` 子句中增加谓词来按照分区值进行过滤时，这些谓词被称为分区过滤器。

即使你做一个跨越整个美国的查询，Hive 也只会读取 65 个文件目录，其中包含有 50 个州，9 大地区，以及哥伦比亚特区和 6 个军事属地。可以通过如下链接查看完整的列表：<http://www.50states.com/abbreviations.htm>。

当然，如果用户需要做一个查询，查询对象是全球各地的所有员工，那么这也是可以做到的。Hive 会不得不读取每个文件目录，但这种宽范围的磁盘扫描还是比较少见的。但是，如果表中的数据以及分区个数都非常大的话，执行这样一个包含所有分区的查询可能会触发一个巨大的 MapReduce 任务。一个高度建议的安全措施就是将 Hive 设置为“strict(严格)”模式，这样如果对分区表进行查询而 `WHERE` 子句没有加分区过滤的话，将会禁止提交这个任务。用户也可以按照下面的语句将属性值设置为“nostrict(非严格)”：

```
hive> set hive.mapred.mode=strict;

hive> SELECT e.name, e.salary FROM employees e LIMIT 100;
FAILED: Error in semantic analysis: No partition predicate found for
Alias "e" Table "employees"

hive> set hive.mapred.mode=nonstrict;

hive> SELECT e.name, e.salary FROM employees e LIMIT 100;
John Doe 100000.0
...
```

可以通过 `SHOW PARTITIONS` 命令查看表中存在的所有分区：

```
hive> SHOW PARTITIONS employees;
...
Country=CA/state=AB
country=CA/state=BC
```

```
...
country=US/state=AL
country=US/state=AK
...
```

如果表中现在存在很多的分区，而用户只想查看是否存储某个特定分区键的分区的话，用户还可以在这个命令上增加一个指定了一个或者多个特定分区字段值的 PARTITION 子句，进行过滤查询：

```
hive> SHOW PARTITIONS employees PARTITION(country='US');
country=US/state=AL
country=US/state=AK
...

hive> SHOW PARTITIONS employees PARTITION(country='US', state='AK');
country=US/state=AK
```

DESCRIBE EXTENDED employees 命令也会显示出分区键：

```
hive> DESCRIBE EXTENDED employees;
name          string,
salary        float,
...
address       struct<...>,
country       string,
state         string

Detailed Table Information...
partitionKeys:[FieldSchema(name:country, type:string, comment:null),
FieldSchema(name:state, type:string, comment:null)],
...
```

输出信息中的模式信息部分会将 country 和 state 以及其他字段列在一起，因为就查询而言，它们就是字段。Detailed Table Information（详细表信息）将 country 和 state 作为分区键处理。这两个键当前的注释都是 null，我们也可以像给普通的字段增加注释一样给分区字段增加注释。

在管理表中用户可以通过载入数据的方式创建分区。如下例中的语句在从一个本地目录（\$HOME/california-employees）载入数据到表中的时候，将会创建一个 US 和 CA（表示加利福尼亚州）分区。用户需要为每个分区字段指定一个值。请注意我们在 HiveQL 中是如何引用 HOME 环境变量的：

```
LOAD DATA LOCAL INPATH '${env:HOME}/california-employees'
INTO TABLE employees
PARTITION (country = 'US', state = 'CA');
```

Hive 将会创建这个分区对应的目录 .../employees/country=US/state=CA，而且

\$HOME/california-employees 这个目录下的文件将会被拷贝到上述分区目录下。参见第 5.1 节“向管理表中装载数据”中的内容。

4.4.1 外部分区表

外部表同样可以使用分区。事实上，用户可能会发现，这是管理大型生产数据集最常见的情况。这种结合给用户提供了一个可以和其他工具共享数据的方式，同时也可以优化查询性能。

因为用户可以自己定义目录结构，因此用户对于目录结构的使用具有更多的灵活性。稍后我们将看到一个特别有用的例子。

我们举一个新例子，非常适合这种场景，即日志文件分析。对于日志信息，大多数的组织使用一个标准的格式，其中记录有时间戳、严重程度（例如 ERROR、WARNING、INFO），也许还包含有服务器名称和进程 ID，然后跟着一个可以为任何内容的文本信息。假设我们是在我们的环境中进行数据抽取、数据转换和数据装载过程（ETL），以及日志文件聚合过程的，将每条日志信息转换为按照制表键分割的记录，并将时间戳解析成年、月和日 3 个字段，剩余的 hms 部分（也就是时间戳剩余的小时、分钟和秒部分）作为一个字段，因为这样显得清楚多了。一种方式是用户可以使用 Hive 或者 Pig 内置的字符串解析函数来完成这个日志信息解析过程。另一种方式是，我们可以使用较小的数值类型来保存时间戳相关的字段以节省空间。这里，我们没有采用后面的解决办法。

我们可以按照如下方式来定义对应的 Hive 表：

```
CREATE EXTERNAL TABLE IF NOT EXISTS log_messages (  
    hms          INT,  
    severity     STRING,  
    server       STRING,  
    process_id  INT,  
    message      STRING)  
PARTITIONED BY (year INT, month INT, day INT)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

我们现在假定将日志数据按照天进行划分，划分数据尺寸合适，而且按天这个粒度进行查询速度也足够快。

回想下，之前我们创建过一个非分区外部表，是一个股票交易表，那时要求使用一个 LOCATION 子句。对于外部分区表则没有这样的要求。有一个 ALTER TABLE 语句可以单独进行增加分区。这个语句需要为每一个分区键指定一个值，本例中，也就是需要为 year、month 和 day 这 3 个分区键都指定值（关于这个功能请参考 4.6 节“修改表”中的内容）。下面是一个例子，演示如何增加一个 2012 年 1 月 2 日的分区：

```
ALTER TABLE log_messages ADD PARTITION(year = 2012, month = 1, day = 2)
```

```
LOCATION 'hdfs://master_server/data/log_messages/2012/01/02';
```

我们使用的目录组织习惯完全由我们自己定义。这里，我们按照分层目录结构组织，因为这是一个合乎逻辑的数据组织方式，但是并非要求一定如此。我们可以遵从 Hive 的目录命名习惯（例如，`../exchange=NASDAQ/symbol=AAPL`），但是也并非要求一定如此。

这种灵活性的一个有趣的优点是我们可以使用像 Amazon S3 这样的廉价的存储设备存储旧的数据，同时保存较新的更加“有趣的”数据到 HDFS 中。例如，每天我们可以使用如下的处理过程将一个月前的旧数据转移到 S3 中。

① 将分区下的数据拷贝到 S3 中。例如，用户可以使用 `hadoop distcp` 命令：

```
hadoop distcp /data/log_message/2011/12/02 s3n: //ourbucket/logs/2011/12/02
```

② 修改表，将分区路径指向到 S3 路径：

```
ALTER TABLE log_messages PARTITION(year = 2011, month = 12, day = 2)
SET LOCATION 's3n://ourbucket/logs/2011/01/02';
```

③ 使用 `hadoop fs -rmr` 命令删除掉 HDFS 中的这个分区数据：

```
hadoop fs -rmr /data/log_messages/2011/01/02
```

并非一定要是 Amazon 弹性 MapReduce 用户才能够这样使用 S3。Apache Hadoop 分支版本包含了对 S3 的支持。用户仍旧是可以查询这些数据的，甚至允许查询越过一个月时间范围的“界限”，也就是有些数据是从 HDFS 中读取的，有些数据是从 S3 中读取的！

顺便说一下，Hive 不关心一个分区对应的分区目录是否存在或者分区目录下是否有文件。如果分区目录不存在或分区目录下没有文件，则对于这个过滤分区的查询将没有返回结果。当用户想在另外一个进程开始往分区中写数据之前创建好分区时，这样做是很方便的。数据一旦存在，对于这份数据的查询就会有返回结果。

这个功能所具有的另一个好处是：可以将新数据写入到一个专用的目录中，并与位于其他目录中的数据存在明显的区别。同时，不管用户是将旧数据转移到一个“存档”位置还是直接删除掉，新数据被篡改的风险都被降低了，因为新数据的数据子集位于不同的目录下。

和非分区外部表一样，Hive 并不控制这些数据。即使表被删除，数据也不会被删除。

和分区管理表一样，通过 `SHOW PARTITIONS` 命令可以查看一个外部表的分区：

```
hive> SHOW PARTITIONS log_messages;
...
year=2011/month=12/day=31
year=2012/month=1/day=1
year=2012/month=1/day=2
...
```

同样地, DESCRIBE EXTENDED log_messages 语句会将分区键作为表的模式的一部分, 和 partitionKeys 列表的内容同时进行显示:

```
hive> DESCRIBE EXTENDED log_messages;
...
message      string,
year         int,
month        int,
day          int

Detailed Table Information...
partitionKeys:[FieldSchema(name:year, type:int, comment:null),
FieldSchema(name:month, type:int, comment:null),
FieldSchema(name:day, type:int, comment:null)],
...
```

这个输出缺少了一个非常重要的信息, 那就是分区数据实际存在的路径。这里有一个路径字段, 但是该字段仅仅表示如果表是管理表其会使用到的 Hive 默认的目录。不过, 我们可以通过如下方式查看到分区数据所在的路径:

```
hive> DESCRIBE EXTENDED log_messages PARTITION (year=2012, month=1,
day=2);
...
location:s3n://ourbucket/logs/2011/01/02,
...
```

我们通常会使用分区外部表, 因为它具有非常多的优点, 例如逻辑数据管理、高性能的查询等。

ALTER TABLE ... ADD PARTITION 语句并非只有对外部表才能够使用。对于管理表, 当有分区数据不是由我们之前讨论过的 LOAD 和 INSERT 语句产生时, 用户同样可以使用这个命令指定分区路径。用户需要记住并非所有的表数据都是放在通常的 Hive “warehouse” 目录下的, 同时当删除管理表时, 这些数据不会连带被删除掉! 因此, 从“理智的”角度来看, 是否敢于对管理表使用这个功能是一个问题。

4.4.2 自定义表的存储格式

在第 3.3 节“文本文件数据编码”中, 我们谈论过 Hive 的默认存储格式是文本文件格式, 这个也可以通过可选的子句 STORED AS TEXTFILE 显式指定, 同时用户还可以在创建表时指定各种各样的分隔符。这里我们重新展示下之前讨论过的那个 employees 表:

```
CREATE TABLE employees (
  name          STRING,
  salary        FLOAT,
  subordinates  ARRAY<STRING>,
  deductions    MAP<STRING, FLOAT>,
  address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
```

```
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\001'  
COLLECTION ITEMS TERMINATED BY '\002'  
MAP KEYS TERMINATED BY '\003'  
LINES TERMINATED BY '\n'  
STORED AS TEXTFILE;
```

TEXTFILE 意味着所有字段都使用字母、数字、字符编码，包括那些国际字符集，尽管我们可以发现 Hive 默认是使用不可见字符来作为“终结者”（分隔符）的。使用 TEXTFILE 就意味着，每一行被认为是一个单独的记录。

用户可以将 TEXTFILE 替换为其他 Hive 所支持的内置文件格式，包括 SEQUENCEFILE 和 RCFILE，这两种文件格式都是使用二进制编码和压缩（可选）来优化磁盘空间使用以及 I / O 带宽性能的。这些文件格式在第 11 章和第 15 章将会有更详细的介绍。

对于记录是如何被编码成文件的，以及列是如何被编码为记录的，Hive 指出了它们之间的不同。用户可以分别自定义这些行为。

记录编码是通过一个 `inputformat` 对象来控制的（例如 TEXTFILE 后面的 Java 代码实现）。Hive 使用了一个名为 `org.apache.hadoop.mapred.TextInputFormat` 的 Java 类（编译后的模块）。如果用户不熟悉 Java 的话，这种点分割的命名语法表明了包的一个分层的树形命名空间，这个结构和 Java 代码的目录结构是对应的。最后一个名字，`TextInputFormat`，是位于最顶层包 `mapred` 下的一个类。

记录的解析是由序列化器/反序列化器（或者缩写为 SerDe）来控制的。对于 TEXTFILE 和我们在第 3 章讨论的编码以及上面我们所重申的例子，Hive 所使用的 SerDe 是另外一个被称为 `org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe` 的 Java 类。

为了保持完整性，Hive 还使用一个叫做 `outputformat` 的对象来将查询的输出写入到文件中或者输出到控制台。对于 TEXTFILE，用于输出的 Java 类名为 `org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat`。



提示

Hive 使用一个 `inputformat` 对象将输入流分割成记录，然后使用一个 `outputformat` 对象来将记录格式化为输出流（例如查询的输出结果），再使用一个 `SerDe` 在读数据时将记录解析成列，在写数据时将列编码成记录。在第 15 章我们将对此展开更深入的讨论。

用户还可以指定第三方的输入和输出格式以及 SerDe，这个功能允许用户自定义 Hive 本身不支持的其他广泛的文件格式。

这里有一个使用了自定义 SerDe、输入格式和输出格式的完整的例子，其可以通过 Avro 协议访问这些文件，第 5.11 节“AvroHive SerDe”中将会介绍到 Avro 协议：

```
CREATE TABLE kst
PARTITIONED BY (ds string)
ROW FORMAT SERDE 'com.linkedin.haiivvreo.AvroSerDe'
WITH SERDEPROPERTIES ('schema.url'='http://schema_provider/kst.avsc')
STORED AS
INPUTFORMAT 'com.linkedin.haiivvreo.AvroContainerInputFormat'
OUTPUTFORMAT 'com.linkedin.haiivvreo.AvroContainerOutputFormat';
```

ROW FORMAT SERDE ... 指定了使用的 SerDe。Hive 提供了 WITH SERDEPROPERTIES 功能，允许用户传递配置信息给 SerDe。Hive 本身并不知晓这些属性的含义，需要 SerDe 去决定这些属性所代表的含义。需要注意的是，每个属性名称和值都应该是带引号的字符串。

STORED AS INPUTFORMAT ... OUTPUTFORMAT ... 子句分别指定了用于输入格式和输出格式的 Java 类。如果要指定，用户必须对输入格式和输出格式都进行指定。

需要注意的是，DESCRIBE EXTENDED table 命令会在 DETAILED TABLE INFORMATION 部分列举出输入和输出格式以及 SerDe 和 SerDe 所自带的属性信息。对于我们的例子，我们可以查看到如下信息：

```
hive> DESCRIBE EXTENDED kst
...
inputFormat:com.linkedin.haiivvreo.AvroContainerInputFormat,
outputFormat:com.linkedin.haiivvreo.AvroContainerOutputFormat,
...
serdeInfo:SerDeInfo(name:null,
serializationLib:com.linkedin.haiivvreo.AvroSerDe,
parameters:{schema.url=http://schema_provider/kst.avsc})
...
...
```

最后，还有一些额外的 CREATE TABLE 子句来更详细地描述数据期望是按照什么样子的存储的。我们重新扩展下在第 4.3.2 节“外部表”中使用到的 stocks 表：

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
  exchange      STRING,
  symbol        STRING,
  ymd           STRING,
  price_open    FLOAT,
  price_high    FLOAT,
  price_low     FLOAT,
  price_close   FLOAT,
  volume        INT,
  price_adj_close FLOAT)
CLUSTERED BY (exchange, symbol)
SORTED BY (ymd ASC)
```

```
INTO 96 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/stocks';
```

CLUSTERED BY ... INTO ... BUCKETS 子句还可以后接一个可选的 SORTED BY ... 子句，用来优化某些特定类型的查询，在第 9.6 节“分桶表数据存储”中将会更详细地进行讨论。

4.5 删除表

Hive 支持和 SQL 中 DROP TABLE 命令类似的操作：

```
DROP TABLE IF EXISTS employees;
```

可以选择是否使用 IF EXISTST 关键字。如果没有使用这个关键字而且表并不存在的话，那么将会抛出一个错误信息。

对于管理表，表的元数据信息和表内的数据都会被删除。



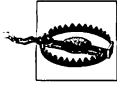
提示

事实上，如果用户开启了 Hadoop 回收站功能（这个功能默认是关闭的），那么数据将会被转移到用户在分布式文件系统中的用户根目录下的 *Trash* 目录下，也就是 HDFS 中的 */user/\$USER/Trash* 目录。如果想开启这个功能，只需要将配置属性 *fs.trash.interval* 的值设置为一个合理的正整数即可。这个值是“回收站检查点”间的时间间隔，单位是分钟。因此如果设置值为 1440，那么就表示是 24 小时。不过并不能保证所有的分布式系统以及所有版本都是支持的这个功能的。如果用户不小心删除了一张存储着重要数据的管理表的话，那么可以先重建表，然后重建所需要的分区，再从 *Trash* 目录中将误删的文件移动到正确的文件目录下（使用文件系统命令）来重新存储数据。

对于外部表，表的元数据信息会被删除，但是表中的数据不会被删除。

4.6 修改表

大多数的表属性可以通过 ALTER TABLE 语句来进行修改。这种操作会修改元数据，但不会修改数据本身。这些语句可用于修改表模式中出现的错误、改变分区路径（在第 4.4.1 节“外部分区表”中有讨论过），以及其他一些操作。



警告

ALTER TABLE 仅仅会修改表元数据，表数据本身不会有任何修改。需要用户自己确认所有的修改都和真实的数据是一致的。

4.6.1 表重命名

使用以下这个语句可以将表 log_messages 重命名为 logmsgsgs:

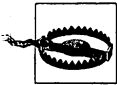
```
ALTER TABLE log_messages RENAME TO logmsgsgs;
```

4.6.2 增加、修改和删除表分区

正如我们前面所见到的，ALTER TABLE table ADD PARTITION ... 语句用于为表（通常是外部表）增加一个新的分区。这里我们增加可提供的可选项，然后多次重复前面的分区路径语句：

```
ALTER TABLE log_messages ADD IF NOT EXISTS
PARTITION (year = 2011, month = 1, day = 1) LOCATION '/logs/2011/01/01'
PARTITION (year = 2011, month = 1, day = 2) LOCATION '/logs/2011/01/02'
PARTITION (year = 2011, month = 1, day = 3) LOCATION '/logs/2011/01/03'
...;
```

当使用 Hive v0.8.0 或其后的版本时，在同一个查询中可以同时增加多个分区。一如既往，IF NOT EXISTS 也是可选的，而且含义不变。



警告

Hive v0.7.* 版本允许用户指定多个分区，但是实际上只会使用第一个指定的分区，而将其他的分区默认省略掉了！其替代方案是，对每一个分区都使用 ALTER STATEMENT 语句。

同时，用户还可以通过高效地移动位置来修改某个分区的路径：

```
ALTER TABLE log_messages PARTITION(year = 2011, month = 12, day = 2)
SET LOCATION 's3n://ourbucket/logs/2011/01/02';
```

这个命令不会将数据从旧的路径转移走，也不会删除旧的数据。

最后，用户可以通过如下语句删除某个分区：

```
ALTER TABLE log_messages DROP IF EXISTS PARTITION(year = 2011, month = 12,
day = 2);
```

按照常规，上面语句中的 IF EXISTS 子句是可选的。对于管理表，即使是使用 ALTER TABLE ... ADD PARTITION 语句增加的分区，分区内的数据也是会同时和元数据信息一起被删除的。对于外部表，分区内数据不会被删除。

还有其他一些和分区相关的 ALTER 语句将会在 第 4.6.7 节“修改存储属性”和第 4.6.8 节“众多的修改表语句”中进行讨论。

4.6.3 修改列信息

用户可以对某个字段进行重命名，并修改其位置、类型或者注释：

```
ALTER TABLE log_messages
CHANGE COLUMN hms hours_minutes_seconds INT
COMMENT 'The hours, minutes, and seconds part of the timestamp'
AFTER severity;
```

即使字段名或者字段类型没有改变，用户也需要完全指定旧的字段名，并给出新的字段名及新的字段类型。关键字 COLUMN 和 COMMENT 子句都是可选的。前面所演示的例子中，我们将字段转移到 severity 字段之后。如果用户想将这个字段移动到第一个位置，那么只需要使用 FIRST 关键字替代 AFTER other_column 子句即可。

和通常一样，这个命令只会修改元数据信息。如果用户移动的是字段，那么数据也应当和新的模式匹配或者通过其他某些方法修改数据以使其能够和模式匹配。

4.6.4 增加列

用户可以在分区字段之前增加新的字段到已有的字段之后。

```
ALTER TABLE log_messages ADD COLUMNS (
  app_name STRING COMMENT 'Application name',
  session_id LONG COMMENT 'The current session id');
```

COMMENT 子句和通常一样，是可选的。如果新增的字段中有某个或多个字段位置是错误的，那么需要使用 ALTER COLUMN 表名 CHANGE COLUMN 语句逐一将字段调整到正确的位置。

4.6.5 删除或者替换列

下面这个例子移除了之前所有的字段并重新指定了新的字段：

```
ALTER TABLE log_messages REPLACE COLUMNS (
  hours_mins_secs INT COMMENT 'hour, minute, seconds from timestamp',
  severity          STRING COMMENT 'The message severity'
  message          STRING COMMENT 'The rest of the message');
```

这个语句实际上重命名了之前的 hms 字段并且从之前的表定义的模式中移除了字段 server 和 process_id。因为是 ALTER 语句，所以只有表的元数据信息改变了。

REPLACE 语句只能用于使用了如下 2 种内置 SerDe 模块的表：DynamicSerDe 或者 MetadataTypedColumnsetSerDe。回想一下，SerDe 决定了记录是如何分解成字段的（反序列化过程）以及字段是如何写入到存储中的（序列化过程）。第 15 章有关于 SerDe

更详细的信息。

4.6.6 修改表属性

用户可以增加附加的表属性或者修改已经存在的属性，但是无法删除属性：

```
ALTER TABLE log_messages SET TBLPROPERTIES (  
    'notes' = 'The process id is no longer captured; this column is always  
    NULL');
```

4.6.7 修改存储属性

有几个 ALTER TABLE 语句用于修改存储格式和 SerDe 属性。

下面这个语句将一个分区的存储格式修改成了 SEQUENCE FILE，我们在第 4.3 节“创建表”中讨论过存储格式（第 11.5 节“sequence file 存储格式”和第 15 章有更详细的信息）：

```
ALTER TABLE log_messages  
PARTITION(year = 2012, month = 1, day = 1)  
SET FILEFORMAT SEQUENCEFILE;
```

如果表是分区表，那么需要使用 PARTITION 子句。

用户可以指定一个新的 SerDe，并为其指定 SerDe 属性，或者修改已经存在的 SerDe 的属性。下面这个例子演示的表使用了一个名为 com.example.JSONSerDe 的 Java 类来处理记录使用 JSON 编码的文件：

```
ALTER TABLE table_using_JSON_storage  
SET SERDE 'com.example.JSONSerDe'  
WITH SERDEPROPERTIES (  
    'prop1' = 'value1',  
    'prop2' = 'value2');
```

SERDEPROPERTIES 中的属性会被传递给 SerDe 模块（本例中，也就是 com.example.JSONSerDe 这个 Java 类）。需要注意的是，属性名（例如 prop1）和属性值（例如 value1）都应当是带引号的字符串。

SERDEPROPERTIES 这个功能是一种方便的机制，它使得 SerDe 的各种实现都允许用户进行自定义。第 15.10 节“JSON SerDe”中我们将可以看到一个真实的 JSON SerDe 例子，以及它是如何使用 SERDEPROPERTIES。

下面这个例子演示了如何向一个已经存在着的 SerDe 增加新的 SERDEPROPERTIES 属性：

```
ALTER TABLE table_using_JSON_storage  
SET SERDEPROPERTIES (  
    'prop3' = 'value3',  
    'prop4' = 'value4');
```

我们可以修改在第4.3节“创建表”中讲到的存储属性：

```
ALTER TABLE stocks
  CLUSTERED BY (exchange, symbol)
  SORTED BY (symbol)
  INTO 48 BUCKETS;
```

`SORTED BY` 子句是可选的，但是 `CLUSTER BY` 和 `INTO ... BUCKETS` 子句是必选的。（在第 9.6 节“分桶表数据存储”中将会更详细介绍数据分桶。）

4.6.8 众多的修改表语句

在第 12.3.2 节“执行钩子”中，我们将讨论一种为各种操作增加执行“钩子”的技巧。`ALTER TABLE ... TOUCH` 语句用于触发这些钩子：

```
ALTER TABLE log_messages TOUCH
  PARTITION(year = 2012, month = 1, day = 1);
```

`PARTITION` 子句用于分区表。这种语句的一个典型应用场景是，当表中存储的文件在 Hive 之外被修改了，就会触发钩子的执行。例如，某个脚本往分区 2012/01/01 中写入了新的日志信息文件，可以在 Hive CLI 中进行下面的调用：

```
hive -e 'ALTER TABLE log_messages TOUCH PARTITION(year = 2012, month = 1,
  day = 1);'
```

如果表或者分区并不存在，那么这个语句也不会创建表或者分区。在这种情况下要使用合适的创建策略。

`ALTER TABLE ... ARCHIVE PARTITION` 语句会将这个分区内的文件打成一个 Hadoop 压缩包（HAR）文件。但是这样仅仅可以降低文件系统中的文件数以及减轻 *NameNode* 的压力，而不会减少任何的存储空间（例如，通过压缩）：

```
ALTER TABLE log_messages ARCHIVE
  PARTITION(year = 2012, month = 1, day = 1);
```

使用 `UNARCHIVE` 替换 `ARCHIVE` 就可以反向操作。这个功能只能用于分区表中独立的分区。

最后，Hive 提供了各种保护。下面的语句可以分别防止分区被删除和被查询：

```
ALTER TABLE log_messages
  PARTITION(year = 2012, month = 1, day = 1) ENABLE NO_DROP;

ALTER TABLE log_messages
  PARTITION(year = 2012, month = 1, day = 1) ENABLE OFFLINE;
```

使用 `ENABLE` 替换 `DISABLE` 可以达到反向操作的目的。这些操作也都不可用于非分区表。

HiveQL：数据操作

本章将继续讨论 HiveQL，也就是 Hive 查询语言，并关注于向表中装载数据和从表中抽取数据到文件系统的数据操作语言部分。

本章中，当我们讨论通过查询语言生成目标表时，大量使用到了 SELECT ... WHERE 语句。那么，我们为什么不先讲述 SELECT ... WHERE 语句，而直到下一章也就是第 6 章才会阐述呢？

既然我们刚讨论了如何创建表，我们就会期望先解决随之而来的下一个问题，即如何装载数据到这些表中，然后我们才能有些东西供查询吧！我们假定用户已经理解了 SQL 的基础知识，因此这些语句对用户来说应该并不陌生。如果用户对此并不熟悉，那么请到第 6 章获取相关更详细的介绍。

5.1 向管理表中装载数据

既然 Hive 没有行级别的数据插入、数据更新和删除操作，那么往表中装载数据的唯一途径就是使用一种“大量”的数据装载操作。或者通过其他方式仅仅将文件写入到正确的目录下。

在第 4.4 节“分区表、管理表”中我们已经看到了一个如何装载数据到管理表中的例子，这里我们稍微对其增加些内容重新进行展示。我们新增了一个关键字 OVERWRITE:

```
LOAD DATA LOCAL INPATH '${env:HOME}/california-employees'  
OVERWRITE INTO TABLE employees  
PARTITION (country = 'US', state = 'CA');
```

如果分区目录不存在的话，这个命令会先创建分区目录，然后再将数据拷贝到该目录下。

如果目标表是非分区表，那么语句中应该省略 PARTITION 子句。

通常情况下指定的路径应该是一个目录，而不是单个独立的文件。Hive 会将所有文件都拷贝到这个目录中。这使得用户将更方便地组织数据到多文件中，同时，在不修改 Hive 脚本的前提下修改文件命名规则。不管怎么样，文件都会被拷贝到目标表路径下而且文件名会保持不变。

如果使用了 LOCAL 这个关键字，那么这个路径应该为本地文件系统路径。数据将会被拷贝到目标位置。如果省略掉 LOCAL 关键字，那么这个路径应该是分布式文件系统的路径。这种情况下，数据是从这个路径转移到目标位置的。



提示

LOAD DATA LOCAL... 拷贝本地数据到位于分布式文件系统上的目标位置，而 LOAD DATA ... (也就是没有使用 LOCAL) 转移数据到目标位置。

之所以会存在这种差异，是因为用户在分布式文件系统中可能并不需要重复的多份数据文件拷贝。

同时，因为文件是以这种方式移动的，Hive 要求源文件和目标文件以及目录应该在同一个文件系统中。例如，用户不可以使用 LOAD DATA 语句将数据从一个集群的 HDFS 中转载（转移）到另一个集群的 HDFS 中。

指定全路径会具有更好的鲁棒性，但也同样支持相对路径。当使用本地模式执行时，相对路径相对的是当 Hive CLI 启动时用户的工作目录。对于分布式或者伪分布式模式，这个路径解读为相对于分布式文件系统中用户的根目录，该目录在 HDFS 和 MapRFS 中默认为 /user/\$USER。

如果用户指定了 OVERWRITE 关键字，那么目标文件夹中之前存在的数据将会被先删除掉。如果没有这个关键字，仅仅会把新增的文件增加到目标文件夹中而不会删除之前的数据。然而，如果目标文件夹中已经存在和装载的文件同名的文件，那么旧的同名文件将会被覆盖重写。（译者注：事实上如果没有使用 OVERWRITE 关键字，而目标文件夹下已经存在同名的文件时，会保留之前的文件并且会重命名新文件为“之前的文件名_序列号”）



提示

Hive v0.9.0 版本之前的版本中存在如下 bug: 如果没有使用 OVERWRITE 关键字，目标文件夹中已经存在和转载文件同名的文件的话，之前的文件会被覆盖重写。因此会产生数据丢失。这个 bug 在 v0.9.0 版本中已经修复了。

如果目标表是分区表那么需要使用 PARTITION 子句, 而且用户还必须为每个分区的键指定一个值。

按照之前所说的那个例子, 数据现在将会存放到如下这个文件夹中:

```
hdfs://master_server/user/hive/warehouse/mydb.db/employees/country=US/
state=CA
```

对于 INPATH 子句中使用的文件路径还有一个限制, 那就是这个路径下不可以包含任何文件夹。

Hive 并不会验证用户装载的数据和表的模式是否匹配。然而, Hive 会验证文件格式是否和表结构定义的一致。例如, 如果表在创建时定义的存储格式是 SEQUENCEFILE, 那么转载进去的文件也应该是 sequencefile 格式的才行。

5.2 通过查询语句向表中插入数据

INSERT 语句允许用户通过查询语句向目标表中插入数据。依旧使用前章中表 employees 作为例子, 这里有一个俄亥俄州的例子, 这里事先假设另一张名为 staged_employees 的表里已经有相关数据了。在表 staged_employees 中我们使用不同的名字来表示国家和州, 分别称作 cnty 和 st, 这样做的原因稍后会进行说明。

```
INSERT OVERWRITE TABLE employees
PARTITION (country = 'US', state = 'OR')
SELECT * FROM staged_employees se
WHERE se.cnty = 'US' AND se.st = 'OR';
```

这里使用了 OVERWRITE 关键字, 因此之前分区中的内容 (如果是非分区表, 就是之前表中的内容) 将会被覆盖掉。

这里如果没有使用 OVERWRITE 关键字或者使用 INTO 关键字替换掉它的话, 那么 Hive 将会以追加的方式写入数据而不会覆盖掉之前已经存在的内容。这个功能只有 Hive v0.8.0 版本以及之后的版本中才有。

这个例子展示了这个功能非常有用的一个常见的场景, 即: 数据已经存在于某个目录下, 对于 Hive 来说其为一个外部表, 而现在想将其导入到最终的分区表中。如果用户想将源表数据导入到一个具有不同记录格式 (例如, 具有不同的字段分割符) 的目标表中的话, 那么使用这种方式也是很好的。

然而, 如果表 staged_employees 非常大, 而且用户需要对 65 个州都执行这些语句, 那么也就意味这需要扫描 staged_employees 表 65 次! Hive 提供了另一种 INSERT 语法, 可以只扫描一次输入数据, 然后按多种方式进行划分。如下例子显示了如何为 3 个州创建表 employees 分区:

```

FROM staged_employees se
INSERT OVERWRITE TABLE employees
  PARTITION (country = 'US', state = 'OR')
  SELECT * WHERE se.cnty = 'US' AND se.st = 'OR'
INSERT OVERWRITE TABLE employees
  PARTITION (country = 'US', state = 'CA')
  SELECT * WHERE se.cnty = 'US' AND se.st = 'CA'
INSERT OVERWRITE TABLE employees
  PARTITION (country = 'US', state = 'IL')
  SELECT * WHERE se.cnty = 'US' AND se.st = 'IL';

```

这里我们使用了缩排使得每组句子看上去更清楚。从 `staged_employees` 表中读取的每条记录都会经过一条 `SELECT ... WHERE ...` 句子进行判断。这些句子都是独立进行判断的，这不是 `IF ... THEN ... ELSE ...` 结构！

事实上，通过使用这个结构，源表中的某些数据可以被写入目标表的多个分区中或者不被写入任一个分区中。

如果某条记录是满足某个 `SELECT ... WHERE ...` 语句的话，那么这条记录就会被写入到指定的表和分区中。简单明了地说，每个 `INSERT` 子句，只要有需要，都可以插入到不同的表中，而那些目标表可以是分区表也可以是非分区表。

因此，输入的某些数据可能输出到多个输出位置而其他一些数据可能就被删除掉了！

当然，这里可以混合使用 `INSERT OVERWRITE` 句式和 `INSERT INTO` 句式。

动态分区插入

前面所说的语法中还是有一个问题，即：如果需要创建非常多的分区，那么用户就需要写非常多的 SQL！不过幸运的是，Hive 提供了一个动态分区功能，其可以基于查询参数推断出需要创建的分区名称。相比之下，到目前为止我们所看到的都是静态分区。

请看如下对前面例子修改后的句子：

```

INSERT OVERWRITE TABLE employees
PARTITION (country, state)
SELECT ..., se.cnty, se.st
FROM staged_employees se;

```

Hive 根据 `SELECT` 语句中最后 2 列来确定分区字段 `country` 和 `state` 的值。这就是为什么在表 `staged_employees` 中我们使用了不同的命名，就是为了强调源表字段值和输出分区值之间的关系是根据位置而不是根据命名来匹配的。

假设表 `staged_employees` 中共有 100 个国家和州的话，执行完上面这个查询后，表 `employees` 就将会有 100 个分区！

用户也可以混合使用动态和静态分区。如下这个例子中指定了 `country` 字段的值为静态

的 US，而分区字段 state 是动态值：

```
INSERT OVERWRITE TABLE employees
PARTITION (country = 'US', state)
SELECT ..., se.cnty, se.st
FROM staged_employees se
WHERE se.cnty = 'US';
```

静态分区键必须出现在动态分区键之前。

动态分区功能默认情况下没有开启。开启后，默认是以“严格”模式执行的，在这种模式下要求至少有一列分区字段是静态的。这有助于阻止因设计错误导致查询产生大量的分区。例如，用户可能错误地使用时间戳作为分区字段，然后导致每秒都对一个分区！而用户也许是期望按照天或者按照小时进行划分的。还有一些其他相关属性值用于限制资源利用。表 5-1 描述了这些属性。

表 5-1 动态分区属性

属性名称	缺省值	描述
hive.exec.dynamic.partition	false	设置成 true，表示开启动态分区功能
hive.exec.dynamic.partition.mode	strict	设置成 nonstrict，表示允许所有分区都是动态的
hive.exec.max.dynamic.partitions.pernode	100	每个 mapper 或 reducer 可以创建的最大动态分区个数。如果某个 mapper 或 reducer 尝试创建大于这个值的分区的话则会抛出一个致命错误信息
hive.exec.max.dynamic.partitions	+1000	一个动态分区创建语句可以创建的最大动态分区个数。如果超过这个值则会抛出一个致命错误信息
hive.exec.max.created.files	100000	全局可以创建的最大文件个数。有一个 Hadoop 计数器会跟踪记录创建了多少个文件，如果超过这个值则会抛出一个致命错误信息

因此，作为例子演示，前面我们使用的第一个使用动态分区的例子看上去应该是像下面这个样子的，这里我们不过在使用前设置了一些期望的属性：

```
hive> set hive.exec.dynamic.partition=true;
hive> set hive.exec.dynamic.partition.mode=nonstrict;
hive> set hive.exec.max.dynamic.partitions.pernode=1000;

hive> INSERT OVERWRITE TABLE employees
> PARTITION (country, state)
> SELECT ..., se.cnty, se.st
> FROM staged_employees se;
```

5.3 单个查询语句中创建表并加载数据

用户同样可以在一个语句中完成创建表并将查询结果载入这个表的操作：

```
CREATE TABLE ca_employees
AS SELECT name, salary, address
FROM employees
WHERE se.state = 'CA';
```

这张表只含有 `employee` 表中来自加利福尼亚州的雇员的 `name`、`salary` 和 `address` 3 个字段的信息。新表的模式是根据 `SELECT` 语句来生成的。

使用这个功能的常见情况是从一个大的宽表中选取部分需要的数据集。

这个功能不能用于外部表。可以回想下使用 `ALTER TABLE` 语句可以为外部表“引用”到一个分区，这里本身没有进行数据“装载”，而是将元数据中指定一个指向数据的路径。

5.4 导出数据

我们如何从表中导出数据呢？如果数据文件恰好是用户需要的格式，那么只需要简单地拷贝文件夹或者文件就可以了：

```
hadoop fs -cp source_path target_path
```

否则，用户可以使用 `INSERT ... DIRECTORY ...`，如下面例子所示：

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/ca_employees'
SELECT name, salary, address
FROM employees
WHERE se.state = 'CA';
```

关键字 `OVERWRITE` 和 `LOCAL` 和前面的说明是一致的，路径格式也和通常的规则一致。一个或者多个文件将会被写入到 `/tmp/ca_employees`，具体个数取决于调用的 `reducer` 个数。

这里指定的路径也可以写成全 URL 路径（例如，`hdfs://master-server/tmp/ca_employees`）。

不管在源表中数据实际是怎么存储的，Hive 会将所有的字段序列化或成字符串写入到文件中。Hive 使用和 Hive 内部存储的表相同的编码方式来生成输出文件。

作为提醒，我们可以在 hive CLI 中查看结果文件内容：

```
hive> ! ls /tmp/ca_employees;
000000_0
hive> ! cat /tmp/payroll/000000_0
John Doe100000.0201 San Antonio CircleMountain ViewCA94040
```

```
Mary Smith80000.01 Infinity LoopCupertinoCA95014
```

```
...
```

是的，文件名是 000000_0。如果有两个或者多个 reducer 来写输出的话，那么我们还可以看到其他相似命名的文件（例如，000001_1）。

如上输出内容看上去字段间没有分隔符，这是因为这里并没有把^A 和^B 显示出来。

和向表中插入数据一样，用户也是可以通过如下方式指定多个输出文件夹目录的：

```
FROM staged_employees se
INSERT OVERWRITE DIRECTORY '/tmp/or_employees'
  SELECT * WHERE se.cty = 'US' and se.st = 'OR'
INSERT OVERWRITE DIRECTORY '/tmp/ca_employees'
  SELECT * WHERE se.cty = 'US' and se.st = 'CA'
INSERT OVERWRITE DIRECTORY '/tmp/il_employees'
  SELECT * WHERE se.cty = 'US' and se.st = 'IL';
```

对于定制输出的数据是有一些限制的（当然，除非自己写一个定制的 OUTPUTFORMAT，这在第 4.4.2 节“自定义表存储格式”中有讲述）。为了格式化字段，Hive 提供了一些内置函数，其中包括那些用于字符串格式化的操作、例如转换操作，拼接输出操作等。详细信息请参考第 6.1.4 节中的“其他内置函数”。

表的字段分隔符可能是需要考量的。例如，如果其使用的是默认的^A 分隔符，而用户又经常导出数据的话，那么可能使用逗号或者制表键作为分隔符会更合适。

另一种变通的方式是定义一个“临时”表，这个表的存储方式配置成期望的输出格式（例如，使用制表键作为字段分隔符）。然后再从这个临时表中查询数据，并使用 INSERT OVERWRITE DIRECTORY 将查询结果写入到这个表中。和很多关系型数据库不同的是，Hive 中没有临时表的概念。用户需要手动删除任何创建了但又长期保留的表。

HiveQL: 查询

在了解了可以通过多种方式来定义和格式化表之后，让我们来学习一下如何运行查询。当然，我们将假设你已经具备了 SQL 的相关知识。为了说明一些概念我们已经使用了一些查询，比如在第 5 章介绍过的加载查询数据到其他表中的操作。现在，我们将完善大部分的细节。一些特殊的主题将会在以后的其他章节中进行介绍。

对于具有 SQL 使用经验的用户来说比较熟悉的细节我们将进行得很快，而会专注于其与 HiveQL 有何差异，包括语法和特性的差异，以及对性能的影响。

6.1 SELECT... FROM 语句

SELECT 是 SQL 中的射影算子。FROM 子句标识了从哪个表、视图或嵌套查询中选择记录（见第 7 章）。

对于一个给定的记录，SELECT 指定了要保存的列以及输出函数需要调用的一个或多个列（例如，像 count(*)这样的聚合函数）。我们再次回想下之前说明过的分区 employees 表：

```
CREATE TABLE employees (  
    name          STRING,  
    salary        FLOAT,  
    subordinates  ARRAY<STRING>,  
    deductions   MAP<STRING, FLOAT>,  
    address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>  
)  
PARTITIONED BY (country STRING, state STRING);
```

我们假定其和第 3.3 节“文本文件数据编码”中所介绍的具有相同的内容，也就是在美国伊利诺伊州（缩写为 IL）中有 4 名员工。下面是对这个表进行的查询语句以及其输出内容：

```
hive> SELECT name, salary FROM employees;
John Doe      100000.0
Mary Smith    80000.0
Todd Jones    70000.0
Bill King     60000.0
```

下面两个查询是等价的。第 2 个版本使用了一个表别名 `e`，在这个查询中不是很有用，但是如果查询中含有链接操作（参考第 6.4 节“JOIN 语句”）的话，会涉及到多个不同的表，那就很有用了。

```
hive> SELECT name, salary FROM employees;
hive> SELECT e.name, e.salary FROM employees e;
```

当用户选择的列是集合数据类型时，Hive 会使用 JSON（Java 脚本对象表示法）语法应用于输出。首先，让我们选择 `subordinates` 列，该列为一个数组，其值使用一个被括在 [...] 内的以逗号分隔的列表进行表示。注意，集合的字符串元素是加上引号的，而基本数据类型 `STRING` 的列值是不加引号的。

```
hive> SELECT name, subordinates FROM employees;
John Doe      ["Mary Smith","Todd Jones"]
Mary Smith    ["Bill King"]
Todd Jones    []
Bill King     []
```

`deductions` 列是一个 MAP，其使用 JSON 格式来表达 map，即使用一个被括在 {...} 内的以逗号分隔的键：值对列表进行表示：

```
hive> SELECT name, deductions FROM employees;
John Doe      {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}
Mary Smith    {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}
Todd Jones    {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}
Bill King     {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}
```

最后，`address` 列是一个 STRUCT，其也是使用 JSON map 格式进行表示的：

```
hive> SELECT name, address FROM employees;
John Doe      {"street":"1 Michigan Ave.,"city":"Chicago","state":"IL",
               "zip":60600}
Mary Smith    {"street":"100 Ontario St.,"city":"Chicago","state":"IL",
               "zip":60601}
Todd Jones    {"street":"200 Chicago Ave.,"city":"Oak Park","state":"IL",
               "zip":60700}
Bill King     {"street":"300 Obscure Dr.,"city":"Obscuria","state":"IL",
               "zip":60100}
```

接下来，让我们看看如何引用集合数据类型中的元素。

首先，数组索引是基于 0 的，这个和在 Java 中是一样的。这里是一个选择 `subordinates` 数

组中的第1个元素的查询：

```
hive> SELECT name, subordinates[0] FROM employees;
John Doe      Mary Smith
Mary Smith    Bill King
Todd Jones    NULL
Bill King     NULL
```

注意，引用一个不存在的元素将会返回 NULL。同时，提取出的 STRING 数据类型的值将不再加引号！

为了引用一个 MAP 元素，用户还可以使用 ARRAY[...]语法，但是使用的是键值而不是整数索引：

```
hive> SELECT name, deductions["State Taxes"] FROM employees;
John Doe      0.05
Mary Smith    0.05
Todd Jones    0.03
Bill King     0.03
```

最后，为了引用STRUCT中的一个元素，用户可以使用“点”符号，类似于前面提到的“表的别名.列名”这样的用法：

```
hive> SELECT name, address.city FROM employees;
John Doe      Chicago
Mary Smith    Chicago
Todd Jones    Oak Park
Bill King     Obscuria
```

WHERE 子句中同样可以使用这些引用方式，这个我们将在第 6.2 节“WHERE 语句”中进行讨论。

6.1.1 使用正则表达式来指定列

我们甚至可以使用正则表达式来选择我们想要的列。下面的查询将会从表 `stocks` 中选择 `symbol` 列和所有列名以 `price` 作为前缀的列¹：

```
hive> SELECT symbol, `price.*` FROM stocks;
AAPL  195.69  197.88  194.0   194.12  194.12
AAPL  192.63  196.0   190.85  195.46  195.46
AAPL  196.73  198.37  191.57  192.05  192.05
AAPL  195.17  200.2   194.42  199.23  199.23
AAPL  195.91  196.32  193.38  195.86  195.86
...
```

我们将在第 6.2.3 节“LIKE 和 RLIKE”中继续讨论在 Hive 中如何使用正则表达式。

1.在撰写本文时，Hive Wiki 中展示了一个不正确的语法来通过正则表达式指定列。

6.1.2 使用列值进行计算

用户不但可以选择表中的列，还可以使用函数调用和算术表达式来操作列值。

例如，我们可以查询得到转换为大写的雇员姓名、雇员对应的薪水、需要缴纳的联邦税收比例以及扣除税收后再进行取整所得的税后薪资。我们甚至可以通过调用内置函数 `map_values` 提取出 `deductions` 字段 `map` 类型值的所有元素，然后使用内置的 `sum` 函数对 `map` 中所有元素进行求和运算。

以下这个查询因为太长，所以我们将它划分成两行显示了。注意下第 2 行 Hive 所使用的提示符，那是一个缩进了的大于符号 (`>`)：

```
hive> SELECT upper(name), salary, deductions["Federal Taxes"],
> round(salary * (1 - deductions["Federal Taxes"])) FROM employees;
JOHN DOE      100000.0    0.2    80000
MARY SMITH    80000.0     0.2    64000
TODD JONES    70000.0     0.15   59500
BILL KING     60000.0     0.15   51000
```

让我们先来讨论一下算术运算符，然后再讨论如何在表达式中使用这些算术运算符。

6.1.3 算术运算符

Hive 中支持所有典型的算术运算符。表 6-1 描述了具体的细节。

表 6-1 算术运算符

运算符	类型	描述
A + B	数值	A 和 B 相加
A - B	数值	A 减去 B
A * B	数值	A 和 B 相乘
A / B	数值	A 除以 B。如果能整除，那么返回商数（译者注：商数是一个整数，表示在不考虑有余数的情况下，除数可以除被除数的次数）
A % B	数值	A 除以 B 的余数
A & B	数值	A 和 B 按位取与
A B	数值	A 和 B 按位取或
A ^ B	数值	A 和 B 按位取亦或
~A	数值	A 按位取反

算术运算符接受任意的数值类型。不过，如果数据类型不同，那么两种类型中值范围较小的那个数据类型将转换为其他范围更广的数据类型。（范围更广在某种意义上就是指一个类型具有更多的字节从而可以容纳更大范围的值。）例如，对于 `INT` 和 `BIGINT` 运算，`INT` 会将类型转换提升为 `BIGINT`。对于 `INT` 和 `FLOAT` 运算，`INT` 将提升为

FLOAT。可以注意到我们的查询语句中包含有(1 - deductions[...])这个运算。因为字段 deductions 是 FLOAT 类型的，因此数字 1 会提升为 FLOAT 类型。

当进行算术运算时，用户需要注意数据溢出或数据下溢问题。Hive 遵循的是底层 Java 中数据类型的规则，因此当溢出或下溢发生时计算结果不会自动转换为更广泛的数据类型。乘法和除法最有可能引发这个问题。

用户需要注意所使用的数值数据的数值范围，并确认实际数据是否接近表模式中定义的数据类型所规定的数值范围上限或者下限，还需要确认人们可能对这些数据进行什么类型的计算。

如果用户比较担心溢出和下溢，那么可以考虑在表模式中定义使用范围更广的数据类型。不过这样做的缺点是每个数据值会占用更多额外的内存。

用户也可以使用特定的表达式将值转换为范围更广的数据类型。详细信息请参考随后的表 6-2 和第 6.8 节“类型转换”。

有时使用函数将数据值按比例从一个范围缩放到另一个范围也是很有用的，例如按照 10 次方幂进行除法运算或取 log 值（指数值），等等。这种数据缩放也适用于某些机器学习计算中，用以提高算法的准确性和数值稳定性。

6.1.4 使用函数

我们前面的那个示例中还使用到了一个内置数学函数 round()，这个函数会返回一个 DOUBLE 类型的最近整数。

1. 数学函数

表 6-2 中描述了 Hive 内置数学函数，这是 Hive v0.8.0 版本中所提供的，而且是用于处理单个列的数据的。

表 6-2 数学函数

返回值类型	样式	描述
BIGINT	round(DOUBLE d)	返回 DOUBLE 型 d 的 BIGINT 类型的近似值
DOUBLE	round(DOUBLE d, INT n)	返回 DOUBLE 型 d 的保留 n 位小数的 DOUBLE 型的近似值
BIGINT	floor(DOUBLE d)	d 是 DOUBLE 类型的，返回 ≤d 的最大 BIGINT 型值
BIGINT	ceil(DOUBLE d) ceiling(DOUBLE d)	d 是 DOUBLE 类型的，返回 ≥d 的最小 BIGINT 型值
DOUBLE	rand() rand(INT seed)	每行返回一个 DOUBLE 型随机数，整数 seed 是随机因子
DOUBLE	exp(DOUBLE d)	返回 e 的 d 幂次方，返回的是个 DOUBLE 型值

续表

返回值类型	样式	描述
DOUBLE	ln(DOUBLE d)	以自然数为底 d 的对数, 返回 DOUBLE 型值
DOUBLE	log10(DOUBLE d)	以 10 为底 d 的对数, 返回 DOUBLE 型值
DOUBLE	log2(DOUBLE d)	以 2 为底 d 的对数, 返回 DOUBLE 型值
DOUBLE	log(DOUBLE base, DOUBLE d)	以 base 为底 d 的对数, 返回 DOUBLE 型值, 其中 base 和 d 都是 DOUBLE 型的
DOUBLE	pow(DOUBLE d, DOUBLE p) power(DOUBLE d, DOUBLE p)	计算 d 的 p 次幂, 返回 DOUBLE 值, 其中 d 和 p 都是 DOUBLE 型的
DOUBLE	sqrt(DOUBLE d)	计算 d 的平方根, 其中 d 是 DOUBLE 型的
STRING	bin(DOUBLE i)	计算二进制值 i 的 STRING 类型值, 其中 i 是 BIGINT 类型的
STRING	hex(BIGINT i)	计算十六进制值 i 的 STRING 类型值, 其中 i 是 BIGINT 类型的
STRING	hex(STRING str)	计算十六进制表达的值 str 的 STRING 类型值
STRING	hex(BINARY b)	计算二进制表达的值 b 的 STRING 类型值 (Hive 0.12.0 版本新增)
STRING	unhex(STRING i)	hex(STRING str)的逆方法
STRING	conv(BIGINT num, INT from_base, INT to_base)	将 BIGINT 类型的 num 从 from_base 进制转换成 to_base 进制, 并返回 STRING 类型结果
STRING	conv(STRING num, INT from_base, INT to_base)	将 STRING 类型的 num 从 from_base 进制转换成 to_base 进制, 并返回 STRING 类型结果
DOUBLE	abs(DOUBLE d)	计算 DOUBLE 型值 d 的绝对值, 返回结果也是 DOUBLE 型的
INT	pmod(INT i1, INT i2)	INT 值 i1 对 INT 值 i2 取模, 结果也是 INT 型的
DOUBLE	pmod(DOUBLE d1, DOUBLE d2)	DOUBLE 值 d1 对 DOUBLE 值 d2 取模, 结果也是 DOUBLE 型的
DOUBLE	sin(DOUBLE d)	在弧度度量中, 返回 DOUBLE 型值 d 的正弦值, 结果是 DOUBLE 型的
DOUBLE	asin(DOUBLE d)	在弧度度量中, 返回 DOUBLE 型值 d 的反正弦值, 结果是 DOUBLE 型的
DOUBLE	cos(DOUBLE d)	在弧度度量中, 返回 DOUBLE 型值 d 的余弦值, 结果是 DOUBLE 型的
DOUBLE	acos(DOUBLE d)	在弧度度量中, 返回 DOUBLE 型值 d 的反余弦值, 结果是 DOUBLE 型的

续表

返回值类型	样式	描述
DOUBLE	<code>tan (DOUBLE d)</code>	在弧度度量中, 返回 DOUBLE 型值 d 的正切值, 结果是 DOUBLE 型的
DOUBLE	<code>atan (DOUBLE d)</code>	在弧度度量中, 返回 DOUBLE 型值 d 的反正切值, 结果是 DOUBLE 型的
DOUBLE	<code>degrees (DOUBLE d)</code>	将 DOUBLE 型弧度值 d 转换成角度值, 结果是 DOUBLE 型的
DOUBLE	<code>radians (DOUBLE d)</code>	将 DOUBLE 型角度值 d 转换成弧度值, 结果是 DOUBLE 型的
INT	<code>positive (INT i)</code>	返回 INT 型值 i (其等价的有效表达式是 $\backslash+i$)
DOUBLE	<code>positive (DOUBLE d)</code>	返回 DOUBLE 型值 d (其等价的有效表达式是 $\backslash+d$)
INT	<code>negative (INT i)</code>	返回 INT 型值 i 的负数 (其等价的有效表达式是 $-i$)
DOUBLE	<code>negative (DOUBLE d)</code>	返回 DOUBLE 型值 d 的负数 (其等价的有效表达式是 $-d$)
FLOAT	<code>sign (DOUBLE d)</code>	如果 DOUBLE 型值 d 是正数的话, 则返回 FLOAT 值 1.0; 如果 d 是负数的话, 则返回 -1.0; 否则返回 0.0
DOUBLE	<code>e ()</code>	数学常数 e, 也就是超越数, 的 DOUBLE 型值
DOUBLE	<code>pi ()</code>	数学常数 pi, 也就是圆周率, 的 DOUBLE 型值

需要注意的是函数 `floor`、`round` 和 `ceil` (“向上取整”) 输入的是 DOUBLE 类型的值, 而返回值是 BIGINT 类型的, 也就是将浮点型数转换成整型了。在进行数据类型转换时, 这些函数是首选的处理方式, 而不是使用前面我们提到过的 `cast` 类型转换操作符。

同样地, 也存在基于不同的底 (例如十六进制) 将整数转换为字符串的函数。

2. 聚合函数

聚合函数是一类比较特殊的函数, 其可以对多行进行一些计算, 然后得到一个结果值。更确切地说, 这是用户自定义聚合函数, 在第 13.4 节“聚合函数”中会有详细的介绍。这类函数中最有名的两个例子就是 `count` 和 `avg`。函数 `count` 用于计算有多少行数据 (或者某列有多少值), 而函数 `avg` 可以返回指定列的平均值。

这里是一个查询示例表 `employees` 中有多少雇员, 以及计算这些雇员平均薪水的 HiveQL 语句:

```
hive> SELECT count (*), avg (salary) FROM employees;
```

当我们在第 6.3 节“GROUP BY 语句”中讨论 GROUP BY 时将会看到其他的例子。

表 6-3 列举了 Hive 的内置聚合函数。

表 6-3 聚合函数

返回值类型	样式	描述
BIGINT	count (*)	计算总行数，包括含有 NULL 值的行
BIGINT	count (expr)	计算提供的 expr 表达式的值非 NULL 的行数
BIGINT	count (DISTINCT expr [, expr_..])	计算提供的 expr 表达式的值排重后非 NULL 的行数
DOUBLE	sum (col)	计算指定行的值的和
DOUBLE	sum (DISTINCT col)	计算排重后值的和
DOUBLE	avg (col)	计算指定行的值的平均值
DOUBLE	avg (DISTINCT col)	计算排重后的值的平均值
DOUBLE	min (col)	计算指定行的最小值
DOUBLE	max (col)	计算指定行的最大值
DOUBLE	variance (col), var_pop (col)	返回集合 col 中的一组数值的方差
DOUBLE	var_samp (col)	返回集合 col 中的一组数值的样本方差
DOUBLE	stddev_pop (col)	返回一组数值的标准偏差
DOUBLE	stddev_samp (col)	返回一组数值的标准样本偏差
DOUBLE	covar_pop (col1, col2)	返回一组数值的协方差
DOUBLE	covar_samp (col1, col2)	返回一组数值的样本协方差
DOUBLE	corr (col1, col2)	返回两组数值的相关系数
DOUBLE	percentile (BIGINT int_expr, p)	int_expr 在 p (范围是: [0,1]) 处的对应的百分比, 其中 p 是一个 DOUBLE 型数值
ARRAY<DOUBLE>	percentile (BIGINT int_expr, ARRAY (P 1 [, P2] ...))	int_expr 在 p (范围是: [0,1]) 处的对应的百分比, 其中 p 是一个 DOUBLE 型数组
DOUBLE	percentile_approx (DOUBLE col, p [, NB])	col 在 p (范围是: [0,1]) 处的对应的百分比, 其中 p 是一个 DOUBLE 型数值, NB 是用于估计的直方图中的仓库数量 (默认是 10000)

返回值类型	样式	描述
ARRAY<DOUBLE>	percentile_approx (DOUBLE col, ARRAY (p1[,p2]...) [, NB])	col 在 p (范围是: [0,1]) 处的对应的百分比, 其中 p 是一个 DOUBLE 型数组, NB 是用于估计的直方图中的仓库数量 (默认是 10000)
ARRAY<STRUCT{x,y}>	histogram_numeric(col, NB)	返回 NB 数量的直方图仓库数组, 返回结果中的值 x 是中心, 值 y 是仓库的高
ARRAY	collect_set(col)	返回集合 col 元素排重后的数组

通常, 可以通过设置属性 `hive.map.aggr` 值为 `true` 来提高聚合的性能, 如下所示:

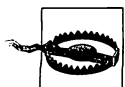
```
hive> SET hive.map.aggr=true;

hive> SELECT count(*), avg(salary) FROM employees;
```

正如这个例子所展示的, 这个设置会触发在 `map` 阶段进行的“顶级”聚合过程。(非顶级的聚合过程将会在执行一个 `GROUP BY` 后进行。) 不过, 这个设置将需要更多的内存。

如表 6-3 所示, 多个函数都可以接受 `DISTINCT ...` 表达式。例如, 我们可以通过这种方式计算排重后的孤僻交易码个数:

```
hive> SELECT count(DISTINCT symbol) FROM stocks;
0
```



警告

等一下, 结果为 0? 当使用 `count(DISTINCT col)` 而同时 `col` 是分区列时存在这个 bug。对于纳斯达克和纽约证券交易所来说答案应该是 743, 至少在我们使用的 2010 年初 `infochimps.org` 提供的数据集中是这个数。

注意, 在 Hive wiki 中, 目前不允许在一个查询语句中使用多于一个的函数(`DISTINCT ...`) 表达式。例如, 下面这个查询语句按说是不允许的, 但是实际上是可以执行的:

```
hive> SELECT count(DISTINCT ymd), count(DISTINCT volume) FROM stocks;
12110      26144
```

因此, 从查询结果中可以看到有 12 110 个交易日的数据, 即超过 40 年的价值。

3. 表生成函数

与聚合函数“相反的”一类函数就是所谓的表生成函数, 其可以将单列扩展成多列或者多行。我们将在第 13.5 节“表生成函数”中更全面地讨论这类函数, 这里我们将简要地讨论下, 然后列举出 Hive 目前所提供的一些内置表生成函数。

下面我们通过一个例子来进行讲解。如下的这个查询语句将 `employees` 表中每行记录中的 `subordinates` 字段内容转换成 0 个或者多个新的记录行。如果某行雇员记录

subordinates 字段内容为空的话，那么将不会产生新的记录；如果不为空的话，那么这个数组的每个元素都将产生一行新记录：

```
hive> SELECT explode(subordinates) AS sub FROM employees;
Mary Smith
Todd Jones
Bill King
```

上面的查询语句中，我们使用 AS sub 子句定义了列别名 sub。当使用表生成函数时，Hive 要求使用列别名。用户可能需要了解其他许多的特性细节才能正确地使用这些函数。第 13.5 节“表生成函数”中，我们将进行详细的讨论。

表 6-4 列举了 Hive 内置的表生成函数。

下面是一个使用函数 parse_url_tuple 的例子，其中我们假设存在一张名为 url_table 的表，而且表中含有一个名为 url 的列，列中存储有很多网址：

```
SELECT parse_url_tuple(url, 'HOST', 'PATH', 'QUERY') as (host, path,
query)
FROM url_table;
```

表 6-4 表生成函数

返回值类型	样式	描述
N 行结果	explode (ARRAY array)	返回 0 到多行结果，每行都对输入输入的 array 数组中的一个元素
N 行结果	explode (MAP map)	返回 0 到多行结果，每行对应每个 map 键-值对，其中一个字段是 map 的键，另一个字段对应 map 的值 (Hive 0.8.0 版本新增)
数组的类型	explode (ARRAY<TYPE> a)	对于 a 中的每个元素，explode() 会生成一行记录包含这个元素
结果插入表中	inline (ARRAY<STRUCT[, STRUCT]>)	将结构体数组提取出来并插入到表中 (Hive 0.10.0 版本新增)
TUPLE	json_tuple (STRING jsonStr, p1, p2, ..., pn)	本函数可以接受多个标签名称，对输入的 JSON 字符串进行处理，这个 get_json_object 这个 UDF 类似，不过更高效，其通过一次调用就可以获得多个键值
TUPLE	parse_url_tuple (url, partname1, partname2, ..., partnameN) 其中 N >= 1	从 URL 中解析出 N 个部分信息。其输入参数是：URL，以及多个要抽取的部分的名称。所有输入的参数类型都是 STRING。部分名称是大小写敏感的，而且不应该包含有空格：HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY:<KEY_NAME>
N 行结果	stack (INT n, col1, ..., colM)	把 M 列转换成 N 行，每行有 M/N 个字段。其中 n 必须是个常数

根据下面的表 6-5，用户可以比较下函数 `parse_url_tuple` 和函数 `parse_url` 的区别。

4. 其他内置函数

表 6-5 中描述了 Hive 中其余的内置函数，这些函数用于处理字符串、Map、数组、JSON 和时间戳，包含或者没有包含最近引入的 `TIMESTAMP` 数据类型（参考第 3.1 节“基本数据类型”中的内容）。

表 6-5 其他内置函数

返回值类型	样式	描述
STRING	<code>ascii (STRING s)</code>	返回字符串 <code>s</code> 中首个 ASCII 字符的整数
STRING	<code>base64 (BINARY bin)</code>	将二进制值 <code>bin</code> 转换成基于 64 位的字符串 (Hive 0.12.0 版本新增)
BINARY	<code>binary (STRING s)</code> <code>binary (BINARY b)</code>	将输入的值转换成二进制值 (Hive 0.12.0 版本新增)
返回类型就是 <code>type</code> 定义的类型	<code>cast (<expr> as <type>)</code>	将 <code>expr</code> 转换成 <code>type</code> 类型的。例如 <code>cast('1' as BIGINT)</code> 将会将字符串 '1' 转换成 <code>BIGINT</code> 数值类型。如果转换过程失败，则返回 <code>NULL</code>
STRING	<code>concat (BINARY s1, BINARY s2, ...)</code>	将二进制字节码按次序拼接成一个字符串 (Hive 0.12.0 版本新增)
STRING	<code>concat (STRING s1, STRING s2, ...)</code>	将字符串 <code>s1,s2</code> 等拼接成一个字符串。例如， <code>concat('ab', 'cd')</code> 的结果是 'abcd'
STRING	<code>concat_ws (STRING separator, STRING s1, STRING s2, ...)</code>	和 <code>concat</code> 类似，不过是使用指定的分隔符进行拼接的
STRING	<code>concat_ws (BINARY separator, BINARY s1, STRING s2, ...)</code>	和 <code>concat</code> 类似，不过是使用指定的分隔符进行拼接的 (Hive 0.12.0 版本新增)
ARRAY<STRUCT <STRING,DOUBLE>>	<code>context_ngrams (array<array<string>>, array<string>, int K, int pf)</code>	和 <code>ngrams</code> 类似，但是从每个外层数组的第二个单词数组来查找前 K 个字尾
STRING	<code>decode (BINARY bin, STRING charset)</code>	使用指定的字符集 <code>charset</code> 将二进制值 <code>bin</code> 解码成字符串 (支持的字符集有 : 'US_ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16')。如果任一输入参数为 <code>NULL</code> ，则结果为 <code>NULL</code> (Hive 0.12.0 版本新增)

返回值类型	样式	描述
BINARY	<code>encode(String src, String charset)</code>	使用指定的字符集 <code>charset</code> 将字符串 <code>src</code> 编码成二进制值(支持的字符集有: 'US_ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16')。如果任一输入参数为 NULL, 则结果为 NULL(Hive 0.12.0 版本新增)
INT	<code>find_in_set(String s, String commaSeparatedString)</code>	返回在以逗号分隔的字符串中 <code>s</code> 出现的位置, 如果没有找到则返回 NULL
STRING	<code>format_number(NUMBER x, INT d)</code>	将数值 <code>x</code> 转换成 '#,###,###.##' 格式字符串, 并保留 <code>d</code> 位小数。如果 <code>d</code> 为 0, 那么输出值就没有小数点后面的值
STRING	<code>get_json_object(String jsonString, String path)</code>	从给定路径上的 JSON 字符串中抽取出 JSON 对象, 并返回这个对象的 JSOM 字符串形式。如果输入的 JSON 字符串是非法的, 则返回 NULL
BOOLEAN	<code>in</code>	例如, <code>test in (val1, val2, ...)</code> , 其表示如果 <code>test</code> 值等于后面列表中的任一值的话, 则返回 true
BOOLEAN	<code>in_file(String s, String filename)</code>	如果文件名为 <code>filename</code> 的文件中有完整一行数据和字符串 <code>s</code> 完全匹配的话, 则返回 true
INT	<code>instr(String str, String substr)</code>	查找字符串 <code>str</code> 中子字符串 <code>substr</code> 第一次出现的位置
INT	<code>length(String s)</code>	计算字符串 <code>s</code> 的长度
INT	<code>locate(String substr, String str [, INT pos])</code>	查找在字符串 <code>str</code> 中的 <code>pos</code> 位置后字符串 <code>substr</code> 第一次出现的位置
STRING	<code>lower(String s)</code>	将字符串中所有字母转换成小写字母。例如, <code>upper('hivE')</code> 的结果是 'hive'
STRING	<code>lcase(String s)</code>	和 <code>lower()</code> 一样
STRING	<code>lpad(String s, INT len, String pad)</code>	从左边开始对字符串 <code>s</code> 使用字符串 <code>pad</code> 进行填充, 最终达到 <code>len</code> 长度为止。如果字符串 <code>s</code> 本身长度比 <code>len</code> 大的话, 那么多余的部分会被去除掉
STRING	<code>ltrim(String s)</code>	将字符串 <code>s</code> 前面出现的空格全部去除掉。例如 <code>trim('hive')</code> 的结果是 'hive'
ARRAY<STRUCT<STRING,DOUBLE>>	<code>ngrams(Array<Array<string>>, INT N, INT K, INT pf)</code>	估算文件中前 <code>K</code> 个字尾。 <code>pf</code> 是精度系数

返回值类型	样式	描述
STRING	<code>parse_url (STRING url, STRING partname [, STRING key])</code>	从 URL 中抽取指定部分的内容。参数 url 表示一个 URL 字符串, 参数 partname 表示要抽取的部分名称, 其是大小写敏感的, 可选的值有: HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY:<key>。如果 partname 是 QUERY 的话, 那么还需要指定第三个参数 key。可以和表 6-4 中的 parse_url_tuple 对比下
STRING	<code>printf (STRING format, Obj ... args)</code>	按照 printf 风格格式化输出输入的字符串 (Hive 0.9.0 版本新增)
STRING	<code>regexp_extract (STRING subject, STRING regex_pattern, STRING index)</code>	抽取字符串 subject 中符合正则表达式 regex_pattern 的第 index 个部分的子字符串
STRING	<code>regexp_replace (STRING s, STRING regex, STRING replacement)</code>	按照 Java 正则表达式 regex 将字符串 s 中符合条件的部分替换成 replacement 所指定的字符串 a。如果 replacement 部分是空的话, 那么符合正则的部门就会被去除掉。例如 <code>regexp_replace('hive', '[ie]', 'z')</code> 的结果是 'hzvz'
STRING	<code>repeat (STRING s, INT n)</code>	重复输出 n 次字符串 s
STRING	<code>reverse (STRING s)</code>	反转字符串
STRING	<code>rpad (STRING s, INT len, STRING pad)</code>	从右边开始对字符串 s 使用字符串 pad 进行填充, 最终达到 len 长度为止。如果字符串 s 本身长度比 len 大的话, 那么多余的部分会被去除掉
STRING	<code>rtrim (STRING s)</code>	将字符串 s 后面出现的空格全部去除掉。例如 <code>trim(' hive')</code> 的结果是 'hive'
ARRAY<ARRAY<STRING>>	<code>sentences (STRING s, STRING lang, STRING locale)</code>	将输入字符串 s 转换成句子数组, 每个句子又由一个单词数组构成。参数 lang 和 locale 是可选的, 如果没有使用的, 则使用默认的本地化信息
INT	<code>size (MAP<K.V>)</code>	返回 MAP 中元素的个数
INT	<code>size (ARRAY<T>)</code>	返回数组 ARRAY 的元素个数
STRING	<code>space (INT n)</code>	返回 n 个空格
ARRAY<STRING>	<code>split (STRING s, STRING pattern)</code>	按照正则表达式 pattern 分割字符串 s, 并将分割后的部分以字符串数组的方式返回

返回值类型	样式	描述
MAP<STRING, STRING>	<code>str_to_map (STRING s, STRING delim1, STRING delim2)</code>	将字符串 s 按照指定分隔符转换成 Map, 第一个参数是输入的字符串, 第二个参数是键值对之间的分隔符, 第三个分隔符是键和价值之间的分隔符
STRING	<code>substr (STRING s, STRING start_index)</code> <code>substring (STRING s, STRING start_index)</code>	对于字符串 s, 从 start 位置开始截取 length 长度的字符串, 作为子字符串。例如 <code>substr('abcdefgh', 3, 2)</code> 的结果是 'cd'
STRING	<code>substr (BINARY s, STRING start_index)</code> <code>substring (BINARY s, STRING start_index)</code>	对于二进制字节值 s, 从 start 位置开始截取 length 长度的字符串, 作为子字符串 (Hive 0.12.0 新增)
STRING	<code>translate (STRING input, STRING from, STRING to)</code>	
STRING	<code>trim (STRING A)</code>	将字符串 s 前后出现的空格全部去除掉。例如 <code>trim('hive')</code> 的结果是 'hive'
BINARY	<code>unbase64 (STRING str)</code>	将基于 64 位的字符串 str 转换成二进制值 (Hive 0.12.0 版本新增)
STRING	<code>upper (STRING A)</code> <code>ucase (STRING A)</code>	将字符串中所有字母转换成大写字母。例如, <code>upper('hIvE')</code> 的结果是 'HIVE'
STRING	<code>from_unixtime (BIGINT unixtime[, STRING format])</code>	将时间戳秒数转换成 UTC 时间, 并用字符串表示, 可以通过 format 规定的时间格式, 指定输出的时间格式
BIGINT	<code>unix_timestamp ()</code>	获取当前本地时区下的当前时间戳
BIGINT	<code>unix_timestamp (STRING date)</code>	输入的时间字符串格式必须是 yyyy-MM-dd HH:mm:ss, 不符合则返回 0, 如果符合则将此时间字符串转换成 Unix 时间戳。例如: <code>unix_timestamp('2009-03-20 11:30:01') = 1237573801</code>
BIGINT	<code>unix_timestamp (STRING date, STRING pattern)</code>	将指定时间字符串格式字符串转换成 Unix 时间戳, 如果格式不对则返回 0。例如: <code>unix_timestamp('2009-03-20', 'yyyy-MM-dd') = 1237532400</code>

续表

返回值类型	样式	描述
STRING	<code>to_date (STRING timestamp)</code>	返回时间字符串的日期部分, 例如: <code>to_date("1970-01-01 00:00:00") = "1970-01-01"</code>
INT	<code>year (STRING date)</code>	返回时间字符串中的年份并使用 INT 类型表示。例如: <code>year("1970-01-01 00:00:00") = 1970</code> , <code>year("1970-01-01") = 1970</code>
INT	<code>month (STRING date)</code>	返回时间字符串中的月份并使用 INT 类型表示。例如: <code>month("1970-11-01 00:00:00") = 11</code> , <code>month("1970-11-01") = 11</code>
INT	<code>day (STRING date)</code> <code>dayofmonth (STRING date)</code>	返回时间字符串中的天并使用 INT 类型表示。例如: <code>day("1970-11-01 00:00:00") = 1</code> , <code>day("1970-11-01") = 1</code>
INT	<code>hour (STRING date)</code>	返回时间戳字符串中的小时并使用 INT 类型表示。例如: <code>hour("2009-07-30 12:58:59") = 12</code> , <code>hour('12:58:59') = 12</code>
INT	<code>minute (STRING date)</code>	返回时间字符串中的分钟数
INT	<code>second (STRING date)</code>	返回时间字符串中的秒数
INT	<code>weekofyear (STRING date)</code>	返回时间字符串位于一年中第几个周内。例如: <code>weekofyear("1970-11-01 00:00:00") = 44</code> , <code>weekofyear("1970-11-01") = 44</code>
INT	<code>datediff (STRING enddate, STRING startdate)</code>	计算开始时间 <code>startdate</code> 到结束时间 <code>enddate</code> 相差的天数。例如: <code>datediff('2009-03-01', '2009-02-27') = 2</code>
STRING	<code>date_add (STRING startdate, INT days)</code>	为开始时间 <code>startdate</code> 增加 <code>days</code> 天。例如: <code>date_add('2008-12-31', 1) = '2009-01-01'</code>
STRING	<code>date_sub (STRING startdate, INT days)</code>	从开始时间 <code>startdate</code> 中减去 <code>days</code> 天。例如: <code>date_sub('2008-12-31', 1) = '2008-12-30'</code>
TIMESTAMP	<code>from_utc_timestamp (TIMESTAMP timestamp, STRING timezone)</code>	如果给定的时间戳并非 UTC, 则将其转化成指定的时区下的时间戳 (Hive 0.8.0 版本新增)
TIMESTAMP	<code>to_utc_timestamp (TIMESTAMP timestamp, STRING timezone)</code>	如果给定的时间戳是指定的时区下的时间戳, 则将其转化成 UTC 下的时间戳 (Hive 0.8.0 版本新增)

需要注意的是, 和时间相关的函数输入的是整型或者字符串类型参数。对于 Hive v0.8.0 版本, 这些函数同样接受 `TIMESTAMP` 类型参数, 同时为了向后兼容, 它们还将继续支持之前的整型和字符串类型参数。

6.1.5 LIMIT 语句

典型的查询会返回多行数据。LIMIT 子句用于限制返回的行数：

```
hive> SELECT upper(name), salary, deductions["Federal Taxes"],
> round(salary * (1 - deductions["Federal Taxes"])) FROM employees
> LIMIT 2;
JOHN DOE      100000.0  0.2  80000
MARY SMITH    80000.0  0.2  64000
```

6.1.6 列别名

前面的示例查询语句可以认为是返回一个由新列组成的新的关系，其中有些新产生的结果列对于表 `employees` 来说是不存在的。通常有必要给这些新产生的列起一个名称，也就是别名。下面这个例子对之前的那个查询进行了修改，为第 3 个和第 4 个字段起了别名，别名分别为 `fed_taxes` 和 `salary_minus_fed_taxes`。

```
hive> SELECT upper(name), salary, deductions["Federal Taxes"] as fed_taxes,
> round(salary * (1 - deductions["Federal Taxes"])) as salary_minus_
fed_taxes
> FROM employees LIMIT 2;
JOHN DOE      100000.0  0.2  80000
MARY SMITH    80000.0  0.2  64000
```

6.1.7 嵌套 SELECT 语句

对于嵌套查询语句来说，使用别名是非常有用的。下面，我们使用前面的示例作为一个嵌套查询：

```
hive> FROM (
>   SELECT upper(name), salary, deductions["Federal Taxes"] as fed_taxes,
>   round(salary * (1 - deductions["Federal Taxes"])) as salary_
minus_fed_taxes
>   FROM employees
> ) e
> SELECT e.name, e.salary_minus_fed_taxes
> WHERE e.salary_minus_fed_taxes > 70000;
JOHN DOE 100000.0 0.2 80000
```

从这个嵌套查询语句中可以看到，我们将前面的结果集起了个别名，称之为 `e`，在这个语句外面嵌套查询了 `name` 和 `salary_minus_fed_taxes` 两个字段，同时约束后者的值要大于 70,000。（在后面的第 6.2 节“WHERE 语句”中我们将会讨论 WHERE 相关内容。）

6.1.8 CASE ... WHEN ... THEN 句式

CASE ... WHEN ... THEN 语句和 if 条件语句类似，用于处理单个列的查询结果。

例如：

```
hive> SELECT name, salary,
> CASE
> WHEN salary < 50000.0 THEN 'low'
> WHEN salary >= 50000.0 AND salary < 70000.0 THEN 'middle'
> WHEN salary >= 70000.0 AND salary < 100000.0 THEN 'high'
> ELSE 'very high'
> END AS bracket FROM employees;
John Doe          100000.0 very high
Mary Smith        80000.0 high
Todd Jones        70000.0 high
Bill King         60000.0 middle
Boss Man          200000.0 very high
Fred Finance      150000.0 very high
Stacy Accountant  60000.0 middle
...
```

6.1.9 什么情况下 Hive 可以避免进行 MapReduce

对于本书中的查询，如果用户进行过执行的话，那么可能会注意到大多数情况下查询都会触发一个 MapReduce 任务 (job)。Hive 中对某些情况的查询可以不必使用 MapReduce，也就是所谓的本地模式，例如：

```
SELECT * FROM employees;
```

在这种情况下，Hive 可以简单地读取 employees 对应的存储目录下的文件，然后输出格式化后的内容到控制台。

对于 WHERE 语句中过滤条件只是分区字段这种情况（无论是否使用 LIMIT 语句限制输出记录条数），也是无需 MapReduce 过程的。

```
SELECT * FROM employees
WHERE country='US' AND state='CA'
LIMIT 100;
```

此外，如果属性 hive.exec.mode.local.auto 的值设置为 true 的话，Hive 还会尝试使用本地模式执行其他的操作：

```
set hive.exec.mode.local.auto=true;
```

否则，Hive 使用 MapReduce 来执行其他所有的查询。



提示

相信我，最好将 set hive.exec.mode.local.auto=true; 这个设置增加到你的 \$HOME/.hiverc 配置文件中。

6.2 WHERE 语句

SELECT 语句用于选取字段，WHERE 语句用于过滤条件，两者结合使用可以查找到符合过滤条件的记录。和 SELECT 语句一样，在介绍 WHERE 语句之前我们已经在很多的简单例子中使用过它了。之前都是假定用户是见过这样的语句的，现在我们将更多地探讨一些细节。

WHERE 语句使用谓词表达式，对于列应用在谓词操作符上的情况，稍后我们将进行讨论。有几种谓词表达式可以使用 AND 和 OR 相连接。当谓词表达式计算结果为 true 时，相应的行将被保留并输出。

我们刚才就使用了下面这个例子来限制查询的结果必须是美国的加利福尼亚州的：

```
SELECT * FROM employees
WHERE country = 'US' AND state = 'CA';
```

谓词可以引用和SELECT语句中相同的各种对于列值的计算。这里我们修改下之前的对于联邦税收的查询，过滤保留那些工资减去联邦税后总额大于70,000的查询结果：

```
hive> SELECT name, salary, deductions["Federal Taxes"],
> salary * (1 - deductions["Federal Taxes"])
> FROM employees
> WHERE round(salary * (1 - deductions["Federal Taxes"])) > 70000;
John Doe      100000.0  0.2  80000.0
```

这个查询语句有点难看，因为第 2 行的那个复杂的表达式和 WHERE 后面的表达式是一样的。下面的查询语句通过使用一个列别名消除了这里表达式重复的问题，但是不幸的是它不是有效的：

```
hive> SELECT name, salary, deductions["Federal Taxes"],
> salary * (1 - deductions["Federal Taxes"]) as salary_minus_fed_taxes
> FROM employees
> WHERE round(salary_minus_fed_taxes) > 70000;
FAILED: Error in semantic analysis: Line 4:13 Invalid table alias or
column reference 'salary_minus_fed_taxes': (possible column names are:
name, salary, subordinates, deductions, address)
```

正如错误信息所提示的，不能在 WHERE 语句中使用列别名。不过，我们可以使用一个嵌套的 SELECT 语句：

```
hive> SELECT e.* FROM
> (SELECT name, salary, deductions["Federal Taxes"] as ded,
> salary * (1 - deductions["Federal Taxes"]) as salary_minus_fed_taxes
> FROM employees) e
> WHERE round(e.salary_minus_fed_taxes) > 70000;
John Doe      100000.0  0.2  80000.0
Boss Man      200000.0  0.3  140000.0
Fred Finance  150000.0  0.3  105000.0
```

6.2.1 谓词操作符

表 6-6 描述了谓词操作符，这些操作符同样可以用于 JOIN... ON 和 HAVING 语句中。

表 6-6 谓词操作符

操作符	支持的数据类型	描述
A = B	基本数据类型	如果 A 等于 B 则返回 TRUE,反之返回 FALSE
A <=> B	基本数据类型	如果 A 和 B 都为 NULL 则返回 TRUE, 其他的和等号 (=) 操作符的结果一致, 如果任一为 NULL 则结果为 NULL (Hive 0.9.0 版本新增)
A == B	没有	这个是错误的语法! SQL 使用=, 而不是==
A <> B, A != B	基本数据类型	A 或者 B 为 NULL 则返回 NULL; 如果 A 不等于 B 则返回 TRUE, 反之返回 FALSE
A < B	基本数据类型	A 或者 B 为 NULL 则返回 NULL; 如果 A 小于 B 则返回 TRUE, 反之返回 FALSE
A <= B	基本数据类型	A 或者 B 为 NULL 则返回 NULL; 如果 A 小于或等于 B 则返回 TRUE, 反之返回 FALSE
A > B	基本数据类型	A 或者 B 为 NULL 则返回 NULL; 如果 A 大于 B 则返回 TRUE, 反之返回 FALSE
A >= B	基本数据类型	A 或者 B 为 NULL 则返回 NULL; 如果 A 大于或等于 B 则返回 TRUE, 反之返回 FALSE
A [NOT] BETWEEN B AND C	基本数据类型	如果 A, B 或者 C 任一为 NULL, 则结果为 NULL。如果 A 的值大于或等于 B 而且小于或等于 C, 则结果为 TRUE, 反之为 FALSE。如果使用 NOT 关键字则可达到相反的效果 (Hive 0.9.0 版本中新增)
A IS NULL	所有数据类型	如果 A 等于 NULL 则返回 TRUE;反之返回 FALSE
A IS NOT NULL	所有数据类型	如果 A 不等于 NULL 则返回 TRUE;反之返回 FALSE
A [NOT] LIKE B	STRING 类型	B 是一个 SQL 下的简单正则表达式, 如果 A 与其匹配的话, 则返回 TRUE;反之返回 FALSE。B 的表达式说明如下: 'x%'表示 A 必须以字母'x'开头, '%x'表示 A 必须以字母'x'结尾, 而'%x%'表示 A 包含有字母'x', 可以位于开头, 结尾或者字符串中间。类似地, 下划线 '_' 匹配单个字符。B 必须要和整个字符串 A 相匹配才行。如果使用 NOT 关键字则可达到相反的效果
A RLIKE B, A REGEXP B	STRING 类型	B 是一个正则表达式, 如果 A 与其相匹配, 则返回 TRUE;反之返回 FALSE。匹配使用的是 JDK 中的正则表达式接口实现的, 因为正则规则也依据其中的规则。例如, 正则表达式必须和整个字符串 A 相匹配, 而不是只需与其子字符串匹配。关于正则表达式请参阅后面更多信息

后面我们将详细讨论下 LIKE 和 RLIKE(请看第 6.2.3 节“LIKE 和 RLIKE”)。首先,我们先说明下用户应该明白的关于浮点数比较的内容。

6.2.2 关于浮点数比较

浮点数比较的一个常见陷阱出现在不同类型间作比较的时候(也就是 FLOAT 和 DOUBLE 比较)。思考下面这个对于员工表的查询语句,该语句将返回员工姓名、工资和联邦税,过滤条件是薪水的减免税款超过 0.2 (20%) :

```
hive> SELECT name, salary, deductions['Federal Taxes']
      > FROM employees WHERE deductions['Federal Taxes'] > 0.2;
John Doe      100000.0    0.2
Mary Smith    80000.0      0.2
Boss Man      200000.0    0.3
Fred Finance  150000.0    0.3
```

等一下! 为什么 deductions['Federal Taxes'] = 0.2 的记录也被输出了?

这是个 Hive 的 Bug 吗? 确实有个 issue 是关于这个问题的, 但是其实际上反映了内部是如何进行浮点数比较的, 这个问题几乎影响了在现在数字计算机中所有使用各种各样编程语言编写的软件(请参阅 <https://issues.apache.org/jira/browse/HIVE-2586>)。

当用户写一个浮点数时, 比如 0.2, Hive 会将该值保存为 DOUBLE 型的。我们之前定义 deductions 这个 map 的值的类型是 FLOAT 型的, 这意味着 Hive 将隐式地将税收减免值转换为 DOUBLE 类型后再进行比较。这样应该是可以的, 对吗?

事实上, 这样行不通。这里解释下为什么不能。数字 0.2 不能够使用 FLOAT 或 DOUBLE 进行准确表示。(参阅 http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html, 深入探讨浮点数问题)。在这个例子中, 0.2 的最近似的精确值应略大于 0.2, 也就是 0.2 后面的若干个 0 后存在非零的数值。

为了简化一点, 实际上我们可以说 0.2 对于 FLOAT 类型是 0.2000001, 而对于 DOUBLE 类型是 0.200000000001。这是因为一个 8 个字节的 DOUBLE 值具有更多的小数位(也就是小数点后的位数)。当表中的 FLOAT 值通过 Hive 转换为 DOUBLE 值时, 其产生的 DOUBLE 值是 0.200000100000, 这个值实际要比 0.200000000001 大。这就是为什么这个查询结果像是使用了 >= 而不是 >了。

这个问题并非仅仅存在于 Hive 中或 Java 中(Hive 是使用 Java 实现的)。而是所有使用 IEEE 标准进行浮点数编码的系统中存在的一个普遍的问题。

然而, Hive 中有两种规避这个问题的方法。

首先, 如果我们是从 TEXTFILE 文本文件(请参考第 15 章内容)中读取数据的话, 也就是目前为止我们所假定使用的存储格式, 那么 Hive 会从数据文件中读取字符串“0.2”, 然后将其转换为一个真实的数字。我们可以在表模式中定义对应的字段类型为

DOUBLE 而不是 FLOAT。这样我们就可以对 `deductions['Federal Taxes']` 这个 DOUBLE 值和 0.2 这个 DOUBLE 值进行比较。不过，这种变化会增加我们查询时所需的内存消耗。同时，如果存储格式是二进制文件格式（如 SEQUENCEFILE（第 15 章将会进行讨论））的话，我们也不能简单地进行这样的改变。

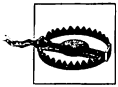
第2个规避方案是显式地指出0.2为FLOAT类型的。Java中有一个很好的方式能够达到这个目的：只需要在数值末尾加上字母F或f(例如，0.2f)。不幸的是，Hive并不支持这种语法，这里我们必须使用cast操作符。

下面这个是修改后的查询语句，其将 0.2 类型转换为 FLOAT 类型了。通过这个修改，返回结果是符合预期的：

```
hive> SELECT name, salary, deductions['Federal Taxes'] FROM employees
      > WHERE deductions['Federal Taxes'] > cast(0.2 AS FLOAT);
Boss Man      200000.0    0.3
Fred Finance  150000.0    0.3
```

注意 cast 操作符内部的语法：数值 AS FLOAT。

实际上，还有第 3 种解决方案，即：和钱相关的都避免使用浮点数。



警告

对浮点数进行比较时，需要保持极端谨慎的态度。要避免任何从窄类型隐式转换到更广泛类型的操作。

6.2.3 LIKE 和 RLIKE

表 6-6 描述了 LIKE 和 RLIKE 谓词操作符。用户可能在之前已经见过 LIKE 的使用了，因为其是一个标准的 SQL 操作符。其可以让我们通过字符串的开头或结尾，以及指定特定的子字符串，或当子字符串出现在字符串内的任何位置时进行匹配。

例如，下面 3 个查询依次分别选择出了住址中街道是以字符串 Ave 结尾的雇员名称和住址、城市是以 O 开头的雇员名称和住址和街道名称中包含有 Chicago 的雇员名称和住址：

```
hive> SELECT name, address.street FROM employees WHERE address.street LIKE
'%Ave.';
John Doe      1 Michigan Ave.
Todd Jones    200 Chicago Ave.

hive> SELECT name, address.city FROM employees WHERE address.city LIKE
'O%';
Todd Jones    Oak Park
Bill King     Obscuria

hive> SELECT name, address.street FROM employees WHERE address.street LIKE
'%Chi%';
```


Todd Jones 200 Chicago Ave.

RLIKE 子句是 Hive 中这个功能的一个扩展,其可以通过 Java 的正则表达式这个更强大的语言来指定匹配条件。不过本书中不会介绍正则表达式的语法和功能。表 6-6 中的 RLIKE 项有关于正则表达更详细信息介绍的链接。这里,我们通过一个例子来展示它们的用法,这个例子会从 employees 表中查找所有住址的街道名称中含有单词 Chicago 或 Ontario 的雇员名称和街道信息:

```
hive> SELECT name, address.street
> FROM employees WHERE address.street RLIKE '.*(Chicago|Ontario).*';
Mary Smith 100 Ontario St.
Todd Jones 200 Chicago Ave.
```

关键字 RLIKE 后面的的字符串表达如下含义:字符串中的点号(.)表示和任意的字符匹配,星号(*)表示重复“左边的字符串”(在以上所示 2 个例子中为点号)零次到无数次。表达式(x|y)表示和 x 或者 y 匹配。

不过,“Chicago”或者“Ontario”字符串前可能没有其他任何字符,而且它们后面也可能不含有其他任何字符。当然,我们也可以通过 2 个 LIKE 子句来改写这个例子为如下这个样子:

```
SELECT name, address FROM employees
WHERE address.street LIKE '%Chicago%' OR address.street LIKE '%Ontario%';
```

通过正则表达式可以比如上这种通过多个 LIKE 子句进行过滤表达更丰富的匹配条件。

关于 Hive 中通过 Java 实现的正则表达式的更详细信息,请查看如下链接中关于 Java 的正则表达式语法部分的介绍:<http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html>, 或者可以参考 Tony Stubblebine (O’ Reilly)所著的《正则表达式参考手册》以及 Jan Goyvaerts 和 Steven Levithan (O’ Reilly)所著的《正则表达式 Cookbook》,也可以参考 Jeffrey E.F. Friedl (O’ Reilly)所著的《精通正则表达式- (第三版)》。

6.3 GROUP BY 语句

GROUP BY 语句通常会和聚合函数一起使用,按照一个或者多个列对结果进行分组,然后对每个组执行聚合操作。

我们重新看下第 4.3.2 节“外部表”中所介绍的股价交易表 stocks。如下这个查询语句按照苹果公司股票(股票代码 APPL)的年份对股票记录进行分组,然后计算每年的平均收盘价:

```
hive> SELECT year(ymd), avg(price_close) FROM stocks
> WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
> GROUP BY year(ymd);
```

```

1984      25.578625440597534
1985      20.193676221040867
1986      32.46102808021274
1987      53.88968399108163
1988      41.540079275138766
1989      41.65976212516664
1990      37.56268799823263
1991      52.49553383386182
1992      54.80338610251119
1993      41.02671956450572
1994      34.0813495847914
...

```

HAVING 语句

HAVING子句允许用户通过一个简单的语法完成原本需要通过子查询才能对GROUP BY语句产生的分组进行条件过滤的任务。如下是对前面的查询语句增加一个HAVING语句来限制输出结果中年平均收盘价要大于\$50.0:

```

hive> SELECT year(ymd), avg(price_close) FROM stocks
      > WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
      > GROUP BY year(ymd)
          > HAVING avg(price_close) > 50.0;
1987      53.88968399108163
1991      52.49553383386182
1992      54.80338610251119
1999      57.77071460844979
2000      71.74892876261757
2005      52.401745992993554
...

```

如果没使用 HAVING 子句, 那么这个查询将需要使用一个嵌套 SELECT 子查询:

```

hive> SELECT s2.year, s2.avg FROM
      > (SELECT year(ymd) AS year, avg(price_close) AS avg FROM stocks
      > WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
      > GROUP BY year(ymd)) s2
      > WHERE s2.avg > 50.0;
1987      53.88968399108163
...

```

6.4 JOIN 语句

Hive 支持通常的 SQL JOIN 语句, 但是只支持等值连接。

6.4.1 INNER JOIN

内连接 (INNER JOIN) 中, 只有进行连接的两个表中都存在与连接标准相匹配的数据

才会被保留下来。例如，如下这个查询对苹果公司的股价（股票代码 AAPL）和 IBM 公司的股价（股票代码 IBM）进行比较。股票表 stocks 进行自连接，连接条件是 ymd 字段（也就是 year-month-day）内容必须相等。我们也称 ymd 字段是这个查询语句中的连接关键字。

```
hive> SELECT a.ymd, a.price_close, b.price_close
> FROM stocks a JOIN stocks b ON a.ymd = b.ymd
> WHERE a.symbol = 'AAPL' AND b.symbol = 'IBM';
2010-01-04    214.01 132.45
2010-01-05    214.38 130.85
2010-01-06    210.97 130.0
2010-01-07    210.58 129.55
2010-01-08    211.98 130.85
2010-01-11    210.11 129.48
...
```

ON 子句指定了两个表间数据进行连接的条件。WHERE 子句限制了左边表是 AAPL 的记录，右边表是 IBM 的记录。同时用户可以看到这个查询中需要为两个表分别指定表别名。

众所周知，IBM 要比 Apple 老得多。IBM 也比 Apple 具有更久的股票交易记录。不过，既然这是一个内连接（INNER JOIN），IBM 的 1984 年 9 月 7 日前的记录就会被过滤掉，也就是 Apple 股票交易日的第一天算起！

标准 SQL 是支持对连接关键词进行非等值连接的，例如下面这个显示 Apple 和 IBM 对比数据的例子，连接条件是 Apple 的股票交易日期要比 IBM 的股票交易日期早。这个将会返回很少数据（如例 6-1 所示）！

例 6-1 Hive 中不支持的查询语句

```
SELECT a.ymd, a.price_close, b.price_close
FROM stocks a JOIN stocks b
ON a.ymd <= b.ymd
WHERE a.symbol = 'AAPL' AND b.symbol = 'IBM';
```

这个语句在 Hive 中是非法的，主要原因是通过 MapReduce 很难实现这种类型的连接。不过因为 Pig 提供了一个交叉生产功能，所以在 Pig 中是可以实现这种连接的，尽管 Pig 的原生连接功能并不支持这种连接。

同时，Hive 目前还不支持在 ON 子句中的谓词间使用 OR。

通过下面的例子我们来看下非自连接操作。dividends 表的数据同样来自于 infochimps.org，正如在第 4.3.2 节中所介绍的：

```
CREATE EXTERNAL TABLE IF NOT EXISTS dividends (
  ymd          STRING,
  dividend    FLOAT
```

```
)  
PARTITIONED BY (exchange STRING, symbol STRING)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

下面这个例子就是苹果公司的 `stocks` 表和 `dividends` 表按照字段 `ymd` 和字段 `symbol` 作为等值连接键的内连接 (INNER JOIN) :

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend  
      > FROM stocks s JOIN dividends d ON s.ymd = d.ymd AND s.symbol = d.symbol  
      > WHERE s.symbol = 'AAPL';  
1987-05-11  AAPL    77.0    0.015  
1987-08-10  AAPL    48.25   0.015  
1987-11-17  AAPL    35.0    0.02  
...  
1995-02-13  AAPL    43.75   0.03  
1995-05-26  AAPL    42.69   0.03  
1995-08-16  AAPL    44.5    0.03  
1995-11-21  AAPL    38.63   0.03
```

是的, 几年前 Apple 公司支付过一次股息, 而且近期刚刚宣布会再次支付股息! 注意到因为我们使用了内连接, 我们只看到每隔 3 个月的记录。通常支付股息的时间表会在宣布季度业绩报告时进行公布。

用户可以对多于 2 张表的多张表进行连接操作。下面我们来对 Apple 公司、IBM 公司和 GE 公司并排进行比较:

```
hive> SELECT a.ymd, a.price_close, b.price_close, c.price_close  
      > FROM stocks a JOIN stocks b ON a.ymd = b.ymd  
      > JOIN stocks c ON a.ymd = c.ymd  
      > WHERE a.symbol = 'AAPL' AND b.symbol = 'IBM' AND c.symbol = 'GE';  
2010-01-04  214.01  132.45  15.45  
2010-01-05  214.38  130.85  15.53  
2010-01-06  210.97  130.0   15.45  
2010-01-07  210.58  129.55  16.25  
2010-01-08  211.98  130.85  16.6  
2010-01-11  210.11  129.48  16.76  
...
```

大多数情况下, Hive 会对每对 JOIN 连接对象启动一个 MapReduce 任务。本例中, 会首先启动一个 MapReduce job 对表 a 和表 b 进行连接操作, 然后再启动一个 MapReduce job 将第一个 MapReduce job 的输出和表 c 进行连接操作。



提示

为什么不是表 b 和表 c 先进行连接操作呢? 这是因为 Hive 总是按照从左到右的顺序执行的。

不过，这个例子实际上得益于一个优化，这个后面我们将进行讨论。

6.4.2 JOIN 优化

在前面的那个例子中，每个 ON 子句中都使用到了 a.ymd 作为其中一个 JOIN 连接键。在这种情况下，Hive 通过一个优化可以在同一个 MapReduce job 中连接 3 张表。同样，如果 b.ymd 也用于 ON 子句中的话，那么也会应用到这个优化。



提示

当对 3 个或者更多个表进行 JOIN 连接时，如果每个 ON 子句都使用相同的连接键的话，那么只会产生一个 MapReduce job。

Hive 同时假定查询中最后一个表是最大的那个表。在对每行记录进行连接操作时，它会尝试将其他表缓存起来，然后扫描最后那个表进行计算。因此，用户需要保证连续查询中的表的大小从左到右是依次增加的。

回想下之前对表 stocks 和表 dividends 进行的连接操作。我们错误地将最小的表 dividends 放在了最后面：

```
SELECT s.ymd, s.symbol, s.price_close, d.dividend
FROM stocks s JOIN dividends d ON s.ymd = d.ymd AND s.symbol = d.symbol
WHERE s.symbol = 'AAPL';
```

应该交换下表 stocks 和表 dividends 的位置，如下所示：

```
SELECT s.ymd, s.symbol, s.price_close, d.dividend
FROM dividends d JOIN stocks s ON s.ymd = d.ymd AND s.symbol = d.symbol
WHERE s.symbol = 'AAPL';
```

不过因为这个数据集并不大，所以并没有明显地看出和之前执行的性能的差别，但是对于大数据集，用户将会感知到这个优化。

幸运的是，用户并非总是要将最大的表放置在查询语句的最后面的。这是因为 Hive 还提供了一个“标记”机制来显式地告诉查询优化器哪张表是大表，使用方式如下：

```
SELECT /*+STREAMTABLE(s)*/s.ymd, s.symbol, s.price_close, d.dividend
FROM stocks s JOIN dividends d ON s.ymd = d.ymd AND s.symbol = d.symbol
WHERE s.symbol = 'AAPL';
```

现在 Hive 将会尝试将表 stocks 作为驱动表，即使其在查询中不是位于最后面的。

还有另外一个类似的非常重要的优化叫做 map-side JOIN,用户可以参考第 6.4.9 节中的内容。

6.4.3 LEFT OUTER JOIN

左外连接通过关键字 LEFT OUTER 进行标识：

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
> FROM stocks s LEFT OUTER JOIN dividends d ON s.ymd = d.ymd AND s.symbol
= d.symbol
> WHERE s.symbol = 'AAPL';
...
1987-05-01    AAPL    80.0    NULL
1987-05-04    AAPL    79.75   NULL
1987-05-05    AAPL    80.25   NULL
1987-05-06    AAPL    80.0    NULL
1987-05-07    AAPL    80.25   NULL
1987-05-08    AAPL    79.0    NULL
1987-05-11    AAPL    77.0    0.015
1987-05-12    AAPL    75.5    NULL
1987-05-13    AAPL    78.5    NULL
1987-05-14    AAPL    79.25   NULL
1987-05-15    AAPL    78.25   NULL
1987-05-18    AAPL    75.75   NULL
1987-05-19    AAPL    73.25   NULL
1987-05-20    AAPL    74.5    NULL
...
```

在这种 JOIN 连接操作中，JOIN 操作符左边表中符合 WHERE 子句的所有记录将会被返回。JOIN 操作符右边表中如果没有符合 ON 后面连接条件的记录时，那么从右边表指定选择的列的值将会是 NULL。

因此，在这个结果集中，我们看到 Apple 公司的股票记录都返回了，而 d.dividend 字段的值通常是 NULL，除了当天有支付股息的那条记录（也就是输出中的 1987 年 5 月 11 日那天的记录）。

6.4.4 OUTER JOIN

在我们讨论其他外连接之前，让我们来讨论一个用户应该明白的问题。

回想下，前面我们说过，通过在 WHERE 子句中增加分区过滤器可以加快查询速度。为了提高前面那个查询的执行速度，我们可以对两张表的 exchange 字段增加谓词限定：

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
> FROM stocks s LEFT OUTER JOIN dividends d ON s.ymd = d.ymd AND s.symbol
= d.symbol
> WHERE s.symbol = 'AAPL'
> AND s.exchange = 'NASDAQ' AND d.exchange = 'NASDAQ';
1987-05-11    AAPL    77.0    0.015
1987-08-10    AAPL    48.25   0.015
1987-11-17    AAPL    35.0    0.02
```

```

1988-02-12  AAPL  41.0  0.02
1988-05-16  AAPL  41.25  0.02
...

```

不过，这时我们发现输出结果改变了，虽然我们可能认为我们不过增加了一个优化！我们重新获得每年 4 条左右的股票交易记录，而且我们发现每年对应的股息值都是非 NULL 的。换句话说，这个效果和之前的内连接（INNER JOIN）是一样的！

在大多数的 SQL 实现中，这种现象实际上是比较常见的。之所以发生这种情况，是因为会先执行 JOIN 语句，然后再将结果通过 WHERE 语句进行过滤。在到达 WHERE 语句时，d.exchange 字段中大多数值为 NULL，因此这个“优化”实际上过滤掉了那些非股息支付日的所有记录。

一个直接有效的解决方式是：移除掉 WHERE 语句中对 dividends 表的过滤条件，也就是去除掉 d.exchange = 'NASDAQ' 这个限制条件。

```

hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
> FROM stocks s LEFT OUTER JOIN dividends d ON s.ymd = d.ymd AND s.symbol
= d.symbol
> WHERE s.symbol = 'AAPL' AND s.exchange = 'NASDAQ';
...
1987-05-07  AAPL  80.25  NULL
1987-05-08  AAPL  79.0  NULL
1987-05-11  AAPL  77.0  0.015
1987-05-12  AAPL  75.5  NULL
1987-05-13  AAPL  78.5  NULL
...

```

这种方式并非很令人满意。用户可能会想知道是否可以将 WHERE 语句中的内容放置到 ON 语句里，至少知道分区过滤条件是否可以放置在 ON 语句中。对于外连接（OUTER JOIN）来说是不可以这样的，尽管 Hive Wiki 中的文章宣传其应该是可以工作的（参考链接：<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins>）。

```

hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
> FROM stocks s LEFT OUTER JOIN dividends d
> ON s.ymd = d.ymd AND s.symbol = d.symbol
> AND s.symbol = 'AAPL' AND s.exchange = 'NASDAQ' AND d.exchange = 'NASDAQ';
...
1962-01-02  GE      74.75  NULL
1962-01-02  IBM     572.0  NULL
1962-01-03  GE      74.0  NULL
1962-01-03  IBM     577.0  NULL
1962-01-04  GE      73.12  NULL
1962-01-04  IBM     571.25  NULL
1962-01-05  GE      71.25  NULL
1962-01-05  IBM     560.0  NULL
...

```

对于外连接(OUTER JOIN)会忽略掉分区过滤条件。不过,对于内连接 (INNER JOIN)使用这样的过滤谓词确实是起作用的!

幸运的是,有一个适用于所有种类连接的解决方案,那就是使用嵌套 SELECT 语句:

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend FROM
  > (SELECT * FROM stocks WHERE symbol = 'AAPL' AND exchange = 'NASDAQ') s
  > LEFT OUTER JOIN
  > (SELECT * FROM dividends WHERE symbol = 'AAPL' AND exchange = 'NASDAQ') d
  > ON s.ymd = d.ymd;
...
1988-02-10  AAPL    41.0    NULL
1988-02-11  AAPL    40.63   NULL
1988-02-12  AAPL    41.0    0.02
1988-02-16  AAPL    41.25   NULL
1988-02-17  AAPL    41.88   NULL
...
```

嵌套 SELECT 语句会按照要求执行“下推”过程,在数据进行连接操作之前会先进行分区过滤。



提示

WHERE 语句在连接操作执行后才会执行,因此 WHERE 语句应该只用于过滤那些非 NULL 值的列值。同时,和 Hive 的文档说明中相反的是, ON 语句中的分区过滤条件外连接(OUTER JOIN)中是无效的,不过在内连接 (INNER JOIN)中是有效的。

6.4.5 RIGHT OUTER JOIN

右外连接 (RIGHT OUTER JOIN) 会返回右边表所有符合 WHERE 语句的记录。左表中匹配不上的字段值用 NULL 代替。

这里我们调整下 stocks 表和 divideneds 表的位置来执行右外连接,并保留 SELECT 语句不变:

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
  > FROM dividends d RIGHT OUTER JOIN stocks s ON d.ymd = s.ymd AND d.symbol
= s.symbol
  > WHERE s.symbol = 'AAPL';
...
1987-05-07  AAPL    80.25   NULL
1987-05-08  AAPL    79.0    NULL
1987-05-11  AAPL    77.0    0.015
1987-05-12  AAPL    75.5    NULL
1987-05-13  AAPL    78.5    NULL
...
```


6.4.6 FULL OUTER JOIN

最后介绍的完全外连接 (FULL OUTER JOIN) 将会返回所有表中符合 WHERE 语句条件的所有记录。如果任一表的指定字段没有符合条件的值的话, 那么就使用 NULL 值替代。

如果我们将前面的查询改写成一个完全外连接查询的话, 事实上获得的结果和之前的一样。这是因为不可能存在有股息支付记录而没有对应的股票交易记录的情况。

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
      > FROM dividends d FULL OUTER JOIN stocks s ON d.ymd = s.ymd AND d.symbol
= s.symbol
      > WHERE s.symbol = 'AAPL';
...
1987-05-07    AAPL    80.25    NULL
1987-05-08    AAPL    79.0     NULL
1987-05-11    AAPL    77.0     0.015
1987-05-12    AAPL    75.5     NULL
1987-05-13    AAPL    78.5     NULL
...
```

6.4.7 LEFT SEMI-JOIN

左半开连接 (LEFT SEMI-JOIN) 会返回左边表的记录, 前提是其记录对于右边表满足 ON 语句中的判定条件。对于常见的内连接 (INNER JOIN) 来说, 这是一个特殊的、优化了的情况。大多数的 SQL 方言会通过 IN ... EXISTS 结构来处理这种情况。例如下面的例 6-2 中所示的查询, 其将试图返回限定的股息支付日内的股票交易记录, 不过这个查询 Hive 是不支持的。

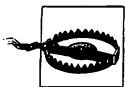
例 6-2 Hive 中不支持的查询

```
SELECT s.ymd, s.symbol, s.price_close FROM stocks s
WHERE s.ymd, s.symbol IN
(SELECT d.ymd, d.symbol FROM dividends d);
```

不过, 用户可以使用如下的 LEFT SEMI JOIN 语法达到同样的目的:

```
hive> SELECT s.ymd, s.symbol, s.price_close
      > FROM stocks s LEFT SEMI JOIN dividends d ON s.ymd = d.ymd AND s.symbol
= d.symbol;
...
1962-11-05    IBM     361.5
1962-08-07    IBM     373.25
1962-05-08    IBM     459.5
1962-02-06    IBM     551.5
```

请注意, SELECT 和 WHERE 语句中不能引用到右边表中的字段。



警告

Hive 不支持右半开连接 (RIGHT SEMI-JOIN)。

SEMI-JOIN 比通常的 INNER JOIN 要更高效, 原因如下: 对于左表中一条指定的记录, 在右边表中一旦找到匹配的记录, Hive 就会立即停止扫描。从这点来看, 左边表中选择的列是可以预测的。

6.4.8 笛卡尔积 JOIN

笛卡尔积是一种连接, 表示左边表的行数乘以右边表的行数等于笛卡尔结果集的大小。也就是说如果左边表有 5 行数据, 而右边表有 6 行数据, 那么产生的结果将是 30 行数据:

```
SELECTS * FROM stocks JOIN dividends;
```

如上面的查询, 以 stocks 表和 dividends 表为例, 实际上很难找到合适的理由来执行这类连接, 因为一只股票的股息通常并非和另一只股票配对。此外, 笛卡尔积会产生大量的数据。和其他连接类型不同, 笛卡尔积不是并行执行的, 而且使用 MapReduce 计算架构的话, 任何方式都无法进行优化。

这里非常有必要指出, 如果使用了错误的连接 (JOIN) 语法可能会导致产生一个执行时间长、运行缓慢的笛卡尔积查询。例如, 如下这个查询在很多数据库中会被优化成内连接 (INNER JOIN), 但是在 Hive 中没有此优化:

```
hive > SELECT * FROM stocks JOIN dividends  
> WHERE stock.symbol = dividends.symbol and stock.symbol='AAPL';
```

在 Hive 中, 这个查询在应用 WHERE 语句中的谓词条件前会先进行完全笛卡尔积计算。这个过程将会消耗很长的时间。如果设置属性 hive.mapred.mode 值为 strict 的话, Hive 会阻止用户执行笛卡尔积查询。第 10 章“调整”中我们将更详尽地讨论这个功能。



提示

笛卡尔积在一些情况下是很有用的。例如, 假设有一个表表示用户偏好, 另有一个表表示新闻文章, 同时有一个算法会推测出用户可能会喜欢读哪些文章。这个时候就需要使用笛卡尔积生成所有用户和所有网页的对应关系的集合。

6.4.9 map-side JOIN

如果所有表中只有一张表是小表, 那么可以在最大的表通过 mapper 的时候将小表完全放到内存中。Hive 可以在 map 端执行连接过程 (称为 map-side JOIN), 这是因为 Hive

可以和内存中的小表进行逐一匹配，从而省略掉常规连接操作所需要的 reduce 过程。即使对于很小的数据集，这个优化也明显地要快于常规的连接操作。其不仅减少了 reduce 过程，而且有时还可以同时减少 map 过程的执行步骤。

stocks 表和 dividends 表之间的连接操作也可以利用到这个优化，因为 dividends 表中的数据集很小，已经可以全部放在内存中缓存起来了。

在 Hive v0.7 之前的版本中，如果想使用这个优化，需要在查询语句中增加一个标记来进行触发。如下面的这个内连接 (INNER JOIN) 的例子所示：

```
SELECT /*+ MAPJOIN(d) */ s.ymd, s.symbol, s.price_close, d.dividend
FROM stocks s JOIN dividends d ON s.ymd = d.ymd AND s.symbol = d.symbol
WHERE s.symbol = 'AAPL';
```

在一个比较快的 MacBook Pro 笔记本电脑上执行上面这个优化后的查询大约需要 23s，这明显比优化前执行所消耗的 33s 要快。在相同的股票样本数据集上执行，执行速度提高了大约 30%。

从 Hive v0.7 版本开始，废弃了这种标记的方式，不过如果增加了这个标记同样也是有效的。如果不加上这个标记，那么这时用户需要设置属性 `hive.auto.convert.JOIN` 的值为 `true`，这样 Hive 才会在必要的时候启动这个优化。默认情况下这个属性的值是 `false`。

```
hive> set hive.auto.convert.join=true;

hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
> FROM stocks s JOIN dividends d ON s.ymd = d.ymd AND s.symbol = d.symbol
> WHERE s.symbol = 'AAPL';
```

需要注意的是，用户也可以配置能够使用这个优化的小表的大小。如下是这个属性的默认值（单位是字节）：

```
hive.mapjoin.smalltable.filesize=25000000
```

如果用户期望 Hive 在必要的时候自动启动这个优化的话，那么可以将这一个（或两个）属性设置在 `$HOME/.hiverc` 文件中。

Hive 对于右外连接 (RIGHT OUTER JOIN) 和全外连接 (FULL OUTER JOIN) 不支持这个优化。

如果所有表中的数据是分桶的，那么对于大表，在特定的情况下同样可以使用这个优化，详细介绍请参见第 9.6 节“分桶表数据存储”中的介绍。简单地说，表中的数据必须是按照 ON 语句中的键进行分桶的，而且其中一张表的分桶的个数必须是另一张表分桶个数的若干倍。当满足这些条件时，那么 Hive 可以在 map 阶段按照分桶数据进行连接。因此这种情况下，不需要先获取到表中所有的内容，之后才去和另一张表中每

个分桶进行匹配连接。

不过，这个优化同样默认是没有开启的。需要设置参数 `hive.optimize.bucketmapJOIN` 为 `true` 才可以开启此优化：

```
set hive.optimize.bucketmapJOIN=true;
```

如果所涉及的分桶表都具有相同的分桶数，而且数据是按照连接键或桶的键进行排序的，那么这时 Hive 可以执行一个更快的分类-合并连接（`sort-merge JOIN`）。同样地，这个优化需要需要设置如下属性才能开启：

```
set hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;  
set hive.optimize.bucketmapjoin=true;  
set hive.optimize.bucketmapjoin.sortedmerge=true;
```

6.5 ORDER BY 和 SORT BY

Hive 中 `ORDER BY` 语句和其他的 SQL 方言中的定义是一样的。其会对查询结果集执行一个全局排序。这也就是说会有一个所有的数据都通过一个 `reducer` 进行处理的过程。对于大数据集，这个过程可能会消耗太过漫长的时间来执行。

Hive 增加了一个可供选择的方式，也就是 `SORT BY`，其只会在每个 `reducer` 中对数据进行排序，也就是执行一个局部排序过程。这可以保证每个 `reducer` 的输出数据都是有序的（但并非全局有序）。这样可以提高后面进行的全局排序的效率。

对于这两种情况，语法区别仅仅是，一个关键字是 `ORDER`，另一个关键字是 `SORT`。用户可以指定任意期望进行排序的字段，并可以在字段后面加上 `ASC` 关键字（默认的），表示按升序排序，或加 `DESC` 关键字，表示按降序排序。

下面是一个使用 `ORDER BY` 的例子：

```
SELECT s.ymd, s.symbol, s.price_close  
FROM stocks s  
ORDER BY s.ymd ASC, s.symbol DESC;
```

下面是一个类似的例子，不过使用的是 `SORT BY`：

```
SELECT s.ymd, s.symbol, s.price_close  
FROM stocks s  
SORT BY s.ymd ASC, s.symbol DESC;
```

上面介绍的两个查询看上去几乎一样，不过如果使用的 `reducer` 的个数大于 1 的话，那么输出结果的排序就大不一样了。既然只保证每个 `reducer` 的输出是局部有序的，那么不同 `reducer` 的输出就可能会有重叠的。

因为 `ORDER BY` 操作可能会导致运行时间过长，如果属性 `hive.mapred.mode` 的值是 `strict` 的话，那么 Hive 要求这样的语句必须加有 `LIMIT` 语句进行限制。默认情况下，

这个属性的值是 `nonstrict`，也就是不会有这样的限制。

6.6 含有 SORT BY 的 DISTRIBUTE BY

`DISTRIBUTE BY` 控制 `map` 的输出在 `reducer` 中是如何划分的。`MapReduce` job 中传输的所有数据都是按照键-值对的方式进行组织的，因此 `Hive` 在将用户的查询语句转换成 `MapReduce` job 时，其必须在内部使用这个功能。

通常，用户不需要担心这个特性。不过对于使用了 `Streaming` 特性（请参考第 14 章内容）以及一些状态为 `UDAF`（用户自定义聚合函数，参见第 13.4 节“聚合函数”中的介绍）的查询是个例外。还有，在另外一个场景下，使用这些语句是有用的。

默认情况下，`MapReduce` 计算框架会依据 `map` 输入的键计算相应的哈希值，然后按照得到的哈希值将键-值对均匀分发到多个 `reducer` 中去。不过不幸的是，这也就意味着当我们使用 `SORT BY` 时，不同 `reducer` 的输出内容会有明显的重叠，至少对于排列顺序而言是这样，即使每个 `reducer` 的输出的数据都是有序的。

假设我们希望具有相同股票交易码的数据在一起处理。那么我们可以使用 `DISTRIBUTE BY` 来保证具有相同股票交易码的记录会分发到同一个 `reducer` 中进行处理，然后使用 `SORT BY` 来按照我们的期望对数据进行排序。如下这个例子就演示了这种用法：

```
hive> SELECT s.ymd, s.symbol, s.price_close
> FROM stocks s
> DISTRIBUTE BY s.symbol
> SORT BY s.symbol ASC, s.ymd ASC;
1984-09-07 AAPL 26.5
1984-09-10 AAPL 26.37
1984-09-11 AAPL 26.87
1984-09-12 AAPL 26.12
1984-09-13 AAPL 27.5
1984-09-14 AAPL 27.87
1984-09-17 AAPL 28.62
1984-09-18 AAPL 27.62
1984-09-19 AAPL 27.0
1984-09-20 AAPL 27.12
...
```

当然，上面例子中的 `ASC` 关键字是可以省略掉的，因为其就是缺省值。这里加上了 `ASC` 关键字的原因，稍后我们将会进行简要的阐述。

`DISTRIBUTE BY` 和 `GROUP BY` 在其控制着 `reducer` 是如何接受一行行数据进行处理这方面是类似的，而 `SORT BY` 则控制着 `reducer` 内的数据是如何进行排序的。

需要注意的是，`Hive` 要求 `DISTRIBUTE BY` 语句要写在 `DORT BY` 语句之前。

6.7 CLUSTER BY

在前面的例子中, `s.symbol` 列被用在了 `DISTRIBUTE BY` 语句中, 而 `s.symbol` 列和 `s.ymd` 位于 `SORT BY` 语句中。如果这 2 个语句中涉及到的列完全相同, 而且采用的是升序排序方式 (也就是默认的排序方式), 那么在这种情况下, `CLUSTER BY` 就等价于前面的 2 个语句, 相当于是前面 2 个句子的一个简写方式。

如下面的例子所示, 我们将前面的查询语句中 `SORT BY` 后面的 `s.ymd` 字段去掉而只对 `s.symbol` 字段使用 `CLUSTER BY` 语句:

```
hive> SELECT s.ymd, s.symbol, s.price_close
      > FROM stocks s
      > CLUSTER BY s.symbol;
2010-02-08 AAPL 194.12
2010-02-05 AAPL 195.46
2010-02-04 AAPL 192.05
2010-02-03 AAPL 199.23
2010-02-02 AAPL 195.86
2010-02-01 AAPL 194.73
2010-01-29 AAPL 192.06
2010-01-28 AAPL 199.29
2010-01-27 AAPL 207.88
...
```

因此排序限制中去除掉了 `s.ymd` 字段, 所以输出中展示的是股票数据的原始排序方式, 也就是降序排列。

使用 `DISTRIBUTE BY ... SORT BY` 语句或其简化版的 `CLUSTER BY` 语句会剥夺 `SORT BY` 的并行性, 然而这样可以实现输出文件的数据是全局排序的。

6.8 类型转换

在第 3.1 节“基本数据类型”中我们简要提及了 Hive 会在适当的时候, 对数值型数据类型进行隐式类型转换, 其关键字是 `cast`。例如, 对不同类型的 2 个数值进行比较操作时就会有这种隐式类型转换。关于这话题我们在第 6.2.1 节“谓词操作符”和第 6.2.2 节“关于浮点数比较”中已经进行了比较全面的讨论。

这里我们讨论下 `cast()` 函数, 用户可以使用这个函数对指定的值进行显式的类型转换。

回想一下, 前面我们介绍过的 `employees` 表中 `salary` 列是使用 `FLOAT` 数据类型的。现在, 我们假设这个字段使用的数据类型是 `STRING` 的话, 那么我们如何才能将其作为 `FLOAT` 值进行计算呢?

如下这个例子会先将值转换为 `FLOAT` 类型, 然后才会执行数值大小比较过程:

```
SELECT name, salary FROM employees
WHERE cast(salary AS FLOAT) < 100000.0;
```

类型转换函数的语法是 `cast(value AS TYPE)`。如果例子中的 `salary` 字段的值不是合法的浮点数字符串的话，那么结果会怎么样呢？这种情况下，Hive 会返回 `NULL`。

需要注意的是，将浮点数转换成整数的推荐方式是使用表 6-2 中列举的 `round()` 或者 `floor()` 函数，而不是使用类型转换操作符 `cast`。

类型转换 BINARY 值

Hive v0.8.0 版本中新引入的 `BINARY` 类型只支持将 `BINARY` 类型转换为 `STRING` 类型。不过，如果用户知道其值是数值的话，那么可以通过嵌套 `cast()` 的方式对其进行类型转换，如下面例子所示，其中 `b` 字段类型是 `BINARY`：

```
SELECT (2.0*cast(cast(b as string) as double)) from src;
```

用户同样可以将 `STRING` 类型转换为 `BINARY` 类型。

6.9 抽样查询

对于非常大的数据集，有时用户需要使用的是一个具有代表性的查询结果而不是全部结果。Hive 可以通过对表进行分桶抽样来满足这个需求。

在下面这个例子中，假设 `numbers` 表只有 `number` 字段，其值是 1 到 10。

我们可以使用 `rand()` 函数进行抽样，这个函数会返回一个随机值。前两个查询都返回了两个不相等的值，而第 3 个查询语句无返回结果：

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;
2
4
```

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;
7
10
```

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;
```

如果我们是按照指定的列而非 `rand()` 函数进行分桶的话，那么同一语句多次执行的返回值是相同的：

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 3 OUT OF 10 ON number) s;
2
```

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 5 OUT OF 10 ON number) s;
4
```

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 3 OUT OF 10 ON number) s;  
2
```

分桶语句中的分母表示的是数据将会被散列的桶的个数，而分子表示将会选择的桶的个数：

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 1 OUT OF 2 ON number) s;  
2  
4  
6  
8  
10
```

```
hive> SELECT * from numbers TABLESAMPLE(BUCKET 2 OUT OF 2 ON number) s;  
1  
3  
5  
7  
9
```

6.9.1 数据块抽样

Hive 提供了另外一种按照抽样百分比进行抽样的方式，这种是基于行数的，按照输入路径下的数据块百分比进行的抽样：

```
hive> SELECT * FROM numbersflat TABLESAMPLE(0.1 PERCENT) s;
```



提示

这种抽样方式不一定适用于所有的文件格式。另外，这种抽样的最小抽样单元是一个 HDFS 数据块。因此，如果表的数据大小小于普通的块大小 128MB 的话，那么将会返回所有行。

基于百分比的抽样方式提供了一个变量，用于控制基于数据块的调优的种子信息：

```
<property>  
  <name>hive.sample.seednumber</name>  
  <value>0</value>  
  <description>A number used for percentage sampling. By changing this  
  number, user will change the subsets of data sampled.</description>  
</property>
```

6.9.2 分桶表的输入裁剪

从第一次看 TABLESAMPLE 语句，精明的用户可能会得出“如下的查询和 TABLESAMPLE 操作相同”的结论：

```
hive> SELECT * FROM numbersflat WHERE number % 2 = 0;
```



```
2
4
6
8
10
```

对于大多数类型的表确实是这样的。抽样会扫描表中所有的数据，然后在每 N 行中抽取一行数据。不过，如果 TABLESAMPLE 语句中指定的列和 CLUSTERED BY 语句中指定的列相同，那么 TABLESAMPLE 查询就只会扫描涉及到的表的哈希分区下的数据：

```
hive> CREATE TABLE numbers_bucketed (number int) CLUSTERED BY (number) INTO
3 BUCKETS;
```

```
hive> SET hive.enforce.bucketing=true;
```

```
hive> INSERT OVERWRITE TABLE numbers_bucketed SELECT number FROM numbers;
```

```
hive> dfs -ls /user/hive/warehouse/mydb.db/numbers_bucketed;
/user/hive/warehouse/mydb.db/numbers_bucketed/000000_0
/user/hive/warehouse/mydb.db/numbers_bucketed/000001_0
/user/hive/warehouse/mydb.db/numbers_bucketed/000002_0
```

```
hive> dfs -cat /user/hive/warehouse/mydb.db/numbers_bucketed/000001_0;
1
7
10
4
```

因为这个表已经聚集成 3 个数据桶了，下面的这个查询可以高效地仅对其中一个数据桶进行抽样：

```
hive> SELECT * FROM numbers_bucketed TABLESAMPLE (BUCKET 2 OUT OF 3 ON NUMBER) s;
1
7
10
4
```

6.10 UNION ALL

UNION ALL 可以将 2 个或多个表进行合并。每一个 union 子查询都必需具有相同的列，而且对应的每个字段的字段类型必须是一致的。例如，如果第 2 个字段是 FLOAT 类型的，那么所有其他子查询的第 2 个字段必须都是 FLOAT 类型的。

下面是个将日志数据进行合并的例子：

```
SELECT log.ymd, log.level, log.message
FROM (
    SELECT l1.ymd, l1.level,
```

```

        l1.message, 'Log1' AS source
    FROM log1 l1
UNION ALL
    SELECT l2.ymd, l2.level,
           l2.message, 'Log2' AS source
    FROM log1 l2
) log
SORT BY log.ymd ASC;

```

UNION 也可用于同一个源表的数据合并。从逻辑上讲，可以使用一个 SELECT 和 WHERE 语句来获得相同的结果。这个技术便于将一个长的复杂的 WHERE 语句分割成 2 个或多个 UNION 子查询。不过，除非源表建立了索引，否则，这个查询将会对同一份源数据进行多次拷贝分发。例如：

```

FROM (
    FROM src SELECT src.key, src.value WHERE src.key < 100
    UNION ALL
    FROM src SELECT src.* WHERE src.key > 110
) unioninput
INSERT OVERWRITE DIRECTORY '/tmp/union.out' SELECT unioninput.*

```

HiveQL: 视图

视图可以允许保存一个查询并像对待表一样对这个查询进行操作。这是一个逻辑结构，因为它不像一个表会存储数据。换句话说，Hive 目前暂不支持物化视图。

当一个查询引用一个视图时，这个视图所定义的查询语句将和用户的查询语句组合在一起，然后供 Hive 制定查询计划。从逻辑上讲，可以想象为 Hive 先执行这个视图，然后使用这个结果进行余下后续的查询。

7.1 使用视图来降低查询复杂度

当查询变得长或复杂的时候，通过使用视图将这个查询语句分割成多个小的、更可控的片段可以降低这种复杂度。这点和在编程语言中使用函数或者软件设计中的分层设计的概念是一致的。封装复杂的部分可以是最终用户通过重用重复的部分来构建复杂的查询。例如，如下是一个具有嵌套子查询的查询：

```
FROM (
  SELECT * FROM people JOIN cart
    ON (cart.people_id=people.id) WHERE firstname='john'
) a SELECT a.lastname WHERE a.id=3;
```

Hive 查询语句中含有多层嵌套是非常常见的。在下面这个例子中，嵌套子查询变成了一个视图：

```
CREATE VIEW shorter_join AS
SELECT * FROM people JOIN cart
ON (cart.people_id=people.id) WHERE firstname='john';
```

现在就可以像操作表一样来操作这个视图了。本次的查询语句中我们为 SELECT 语句增加了一个 WHERE 子句，这样就大大简化了之前的那个查询语句：

```
SELECT lastname FROM shorter_join WHERE id=3;
```

7.2 使用视图来限制基于条件过滤的数据

对于视图来说一个常见的使用场景就是基于一个或多个列的值来限制输出结果。有些数据库允许将视图作为一个安全机制，也就是不给用户直接访问具有敏感数据的原始表，而是提供给用户一个通过 WHERE 子句限制了视图，以供访问。Hive 目前并不支持这个功能，因为用户必须具有能够访问整个底层原始表的权限，这时视图才能工作。然而，通过创建视图来限制数据访问可以用来保护信息不被随意查询：

```
hive> CREATE TABLE userinfo (  
  > firstname string, lastname string, ssn string, password string);  
hive> CREATE VIEW safer_user_info AS  
  > SELECT firstname,lastname FROM userinfo;
```

如下是通过 WHERE 子句限制数据访问的视图的另一个例子。在这种情况下，我们希望提供一个员工表视图，只暴露来自特定部门的员工信息：

```
hive> CREATE TABLE employee (firstname string, lastname string,  
  > ssn string, password string, department string);  
hive> CREATE VIEW techops_employee AS  
  > SELECT firstname,lastname,ssn FROM userinfo WHERE department='techops';
```

7.3 动态分区中的视图和 map 类型

记得在第 3 章我们就介绍过 Hive 支持 array、map 和 struct 数据类型。这些数据类型在传统数据库中并不常见，因为它们破坏了第一范式。Hive 可将一行文本作为一个 map 而非一组固定的列的能力，加上视图功能，就允许用户可以基于同一个物理表构建多个逻辑表。

思考下面这个示例文件。其将整行作为一个 map 处理，而不是一列固定的列。这里没有使用 Hive 的默认分隔符，这个文件使用 ^A(Control + A) 作为集合内元素间的分隔符（例如，本例中 map 的多个键-值对之间的分隔符），然后使用 ^B (Control + B) 作为 map 中的键和值之间的分隔符。因为这条记录较长，所以为了更清晰地表达，我们人为地增加了空行：

```
time^B1298598398404^Atype^Brequest^Astate^Bny^Acity^Bwhite  
plains^Apart^Bmuffler  
  
time^B1298598398432^Atype^Bresponse^Astate^Bny^Acity^Barrystown^  
Apart^Bmuffler  
  
time^B1298598399404^Atype^Brequest^Astate^Btx^Acity^Baustin^  
Apart^Bheadlight
```

下面我们来创建表：

```
CREATE EXTERNAL TABLE dynamictable(cols map<string,string>)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\004'
  COLLECTION ITEMS TERMINATED BY '\001'
  MAP KEYS TERMINATED BY '\002'
STORED AS TEXTFILE;
```

上面的例子中，因为每行只有一个字段，因此 FIELDS TERMINATED BY 语句所指定的分隔符实际上没有任何影响。

现在我们可以创建这样一个视图，其仅取出 type 值等于 request 的 city、state 和 part 3 个字段，并将视图命名为 orders。视图 orders 具有 3 个字段：state、city 和 part。

```
CREATE VIEW orders(state, city, part) AS
SELECT cols["state"], cols["city"], cols["part"]
FROM dynamictable
WHERE cols["type"] = "request";
```

我们创建的第 2 个视图名为 shipments。这个视图返回 time 和 part 2 个字段作为列，限制条件是 type 的值为 response：

```
CREATE VIEW shipments(time, part) AS
SELECT cols["time"], cols["parts"]
FROM dynamictable
WHERE cols["type"] = "response";
```

关于这个特性的另一个例子，请查看下如下链接：<http://dev.bizo.com/2011/02/columns-in-hive.html#!/2011/02/columns-in-hive.html>。

7.4 视图零零碎碎相关的事情

我们说过，Hive 会先解析视图，然后使用解析结果再来解析整个查询语句。然而，作为 Hive 查询优化器的一部分，查询语句和视图语句可能会合并成一个单一的实际查询语句。

然而，这个概念视图仍然适用于视图和使用这个视图的查询语句都包含了一个 ORDER BY 子句或一个 LIMIT 子句的情况。这时会在使用这个视图的查询语句之前对该视图进行解析。

例如，如果视图语句含有一个 LIMIT 100 子句，而同时使用到这个视图的查询含有一个 LIMIT 200 子句，那么用户最终最多只能获取 100 条结果记录。

因为定义一个视图实际上并不会“具体化”操作任何实际数据，所以视图实际上是对

其所使用到的表和列的一个查询语句固化过程。因此，如果视图所涉及的表或者列不再存在时，会导致视图查询失败。

创建视图时用户还可以使用其他一些子句。如下例子修改了我们前面那个例子：

```
CREATE VIEW IF NOT EXISTS shipments(time, part)
COMMENT 'Time and parts for shipments.'
TBLPROPERTIES ('creator' = 'me')
AS SELECT ...;
```

对于表来说，IF NOT EXISTS 和 COMMENT... 子句是可选的，而且和表创建语句具有相同的含义。

一个视图的名称要和这个视图所在的数据库下的其他所有表和视图的名称不同。

用户还可以为所有的新列或部分新列增加一个 COMMENT 子句，进行写注释。这些注释并非“继承”原始表中的定义。

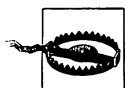
同样地，如果 AS SELECT 子句中包含没有命名别名的表达式的话，例如 size(cols)(计算 cols 中元素的个数)，那么 Hive 将会使用_CN 作为新的列名，其中 N 表示从 0 开始的一个整数。如果 AS SELECT 语句不合法的话，那么创建视图过程将失败。

在 AS SELECT 子句之前，用户可以通过定义 TBLPROPERTIES 来定义表属性信息，这点和表相同。上例中，我们定义的属性为“creator”，表示这个视图的创建者名称。

第 4.3 节“创建表”中讨论过的 CREATE TABLE ... LIKE ... 结构同样适用于复制视图，只需要在 LIKE 表达式里面写视图名就可以了：

```
CREATE TABLE shipments2
LIKE shipments;
```

用户也可以像以前一样选择性使用 EXTERNAL 关键字和 LOCATION... 子句。



警告

对于 Hive v0.8.0 版本和这个版本前的其他版本来说，这个语句的行为是不同的。对于 v0.8.0 版本，这个命令会创建一个新的表，而不是一个新的视图，同时使用默认的 SerDe 方式和文件格式。而对于之前的早期版本来说，会创建一个新的视图。

删除视图的方式和删除表的方式类似：

```
DROP VIEW IF EXISTS shipments;
```

和往常一样，上面例子中的 IF EXISTS 语句是可选的。

通过 SHOW TABLES 语句（没有 SHOW VIEWS 这样的语句）同样可以查看到视图，

但是不能使用 DROP TABLE 语句来删除视图。

和表一样，DESCRIBE 和 DESCRIBE EXTENDED 语句可以显示视图的元数据信息。如果使用后面那个命令，输出信息中的“Detailed Table Information”部分会有一个 tableType 字段，字段值显示的是“VIRTUAL_VIEW”。

视图不能够作为 INSERT 语句或 LOAD 命令的目标表。

最后要说明的是，视图是只读的。对于视图只允许改变元数据中 TBLPROPERTIES 属性信息：

```
ALTER VIEW shipments SET TBLPROPERTIES ('created_at' = 'some_timestamp');
```

HiveQL: 索引

Hive 只有有限的索引功能。Hive 中没有普通关系型数据库中键的概念，但是还是可以对一些字段建立索引来加速某些操作的。一张表的索引数据存储在另外一张表中。

同时，因为这是一个相对比较新的功能，所以目前还没有提供很多的选择。然而，索引处理模块被设计成为可以定制的 Java 编码的插件，因此，用户可以根据需要对其进行实现，以满足自身的需求。

当逻辑分区实际上太多太细而几乎无法使用时，建立索引也就成为分区的另一个选择。建立索引可以帮助裁剪掉一张表的一些数据块，这样能够减少 MapReduce 的输入数据量。并非所有的查询都可以通过建立索引获得好处。通过 EXPLAIN 命令可以查看某个查询语句是否用到了索引。

Hive 中的索引和那些关系型数据库中的一样，需要进行仔细评估才能使用。维护索引也需要额外的存储空间，同时创建索引也需要消耗计算资源。用户需要在建立索引为查询带来的好处和因此而需要付出的代价之间做出权衡。

8.1 创建索引

现在我们来为我们在第 56 页“分区表、管理表”章节讲述过的管理分区表中的 employees 表建立一个索引。下面这段是我们之前的表定义语句，作为参照：

```
CREATE TABLE employees (  
    name          STRING,  
    salary        FLOAT,  
    subordinates  ARRAY<STRING>,  
    deductions    MAP<STRING, FLOAT>,  
    address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>  
)  
PARTITIONED BY (country STRING, state STRING);
```


下面我们仅对分区字段 `country` 建立索引：

```
CREATE INDEX employees_index
ON TABLE employees (country)
AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'
WITH DEFERRED REBUILD
IDXPROPERTIES ('creator = 'me', 'created_at' = 'some_time')
IN TABLE employees_index_table
PARTITIONED BY (country, name)
COMMENT 'Employees indexed by country and name.';
```

在这种情况下，我们没有像原表那样对索引表进行同一粒度的分区划分。其实我们是那么做的。如果我们完全省略掉 `PARTITIONED BY` 语句的话，那么索引将会包含原始表的所有分区。

`AS ...` 语句指定了索引处理器，也就是一个实现了索引接口的 Java 类。Hive 本身包含了一些典型的索引实现。这里所展示的 `CompactIndexHandler` 就是其中的一个实现。可以通过第三方的实现来最优化地处理特定的场景，支持特定的文件格式，等等。在第 121 页的“实现一个定制化的索引处理器”章节，我们将对如何实现一个用户定制的索引处理器进行详细的说明。

下一节我们将讨论 `WITH DEFERRED REBUILD` 这个子句的含义。

并非一定要求索引处理器在一张新表中保留索引数据，但是如果需要的话，会使用到 `IN TABLE ...` 子句。这个句式提供了和创建其他类型表一样的很多功能。特别地，例子中没有使用到 `ROW FORMAT`、`STORED AS`、`STORED BY`、`LOCATION` 等我们在第 4 章所讨论过的选项。这些其实都是可以在最后的 `COMMENT` 语句前增加的。

目前，除了 S3 中的数据，对外部表和视图都是可以建立索引的。

Bitmap 索引

Hive v0.8.0 版本中新增了一个内置 `bitmap` 索引处理器。`bitmap` 索引普遍应用于排重后值较少的列。下面是对前面的例子使用 `bitmap` 索引处理器重写后的语句：

```
CREATE INDEX employees_index
ON TABLE employees (country)
AS 'BITMAP'
WITH DEFERRED REBUILD
IDXPROPERTIES ('creator' = 'me', 'created_at' = 'some_time')
IN TABLE employees_index_table
PARTITIONED BY (country, name)
COMMENT 'Employees indexed by country and name.';
```

8.2 重建索引

如果用户指定了 DEFERRED REBUILD，那么新索引将呈现空白状态。在任何时候，都可以进行第一次索引创建或者使用 ALTER INDEX 对索引进行重建：

```
ALTER INDEX employees_index
ON TABLE employees
PARTITION (country = 'US')
REBUILD;
```

如果省略掉 PARTITION，那么将会对所有分区进行重建索引。

还没有一个内置的机制能够在底层的表或者某个特定的分区发生改变时，自动触发重建索引。但是，如果用户具有一个 workflow 来更新表分区中的数据的话，那么用户可能已经在其中某处使用到了第 69 页“众多的 Alter Table 语句”章节中所说明的 ALTER TABLE ... TOUCH PARTITION(...)功能，同样地，在这个 workflow 中也可以对对应的索引执行重建索引语句 ALTER INDEX ... REBUILD。

如果重建索引失败，那么在重建开始之前，索引将停留在之前的版本状态。从这种意义上看，重建索引操作是原子性的。

8.3 显示索引

下面这个命令将显示对于这个索引表对所有列所建立的索引：

```
SHOW FORMATTED INDEX ON employees;
```

关键字 FORMATTED 是可选的。增加这个关键字可以使输出中包含有列名称。用户还可以替换 INDEX 为 INDEXES，这样输出中就可以列举出多个索引信息了。

8.4 删除索引

如果有索引表的话，删除一个索引将会删除这个索引表：

```
DROP INDEX IF EXISTS employees_index ON TABLE employees;
```

Hive 不允许用户直接使用 DROP TABLE 语句之前删除索引表。而通常情况下，IF EXISTS 都是可选的，其用于当索引不存在时避免抛出错误信息。

如果被索引的表被删除了，那么其对应的索引和索引表也会被删除。同样地，如果原始表的某个分区被删除了，那么这个分区对应的分区索引也同时会被删除掉。

8.5 实现一个定制化的索引处理器

在 Hive Wiki 页面具有实现一个定制化的索引处理器的完整的例子，链接是 https://cwiki.apache.org/confluence/display/Hive/IndexDev#CREATE_INDEX。其中还包含了索引的初步设计文档。当然，用户也是可以使用 `org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler` 中的源代码作为例子来学习的。

创建好索引后，实现索引处理器的 Java 代码中也要做一些初始验证和为索引表（如果使用到索引表的话）定义模式的过程。同时还要实现重建索引处理逻辑，其会读取需要创建索引的表，然后写索引存储（例如索引表）。但索引被删除后，处理器还需要清理掉所有为索引所使用到的非表存储。如果需要，还要依赖 Hive 去删除索引表。处理器必须能够参与到优化查询中。

Hive 看上去以及实际行为都像一个关系型数据库。用户对如表和列这类术语比较熟悉，而且 Hive 提供的查询语言和用户之前使用过的 SQL 方言非常地相似。不过，Hive 实现和使用的方式和传统的关系型数据库是非常不同的。通常，用户视图移植关系型数据库中的模式，而事实上 Hive 是反模式的。本章将重点介绍 Hive 中哪些模式是用户应该使用的，而又有哪些反模式是应该避免使用的。

9.1 按天划分的表

按天划分表就是一种模式，其通常会在表名中加入一个时间戳，例如表名为 `upply_2011_01_01`、`supply_2011_01_02`，等等。这种每天一张表的方式在数据库领域是反模式的一种方式，但是因为实际情况下数据集增长得很快，这种方式应用还是比较广泛的。

```
hive> CREATE TABLE supply_2011_01_02 (id int, part string, quantity int);

hive> CREATE TABLE supply_2011_01_03 (id int, part string, quantity int);

hive> CREATE TABLE supply_2011_01_04 (id int, part string, quantity int);

hive> .... load data ...

hive> SELECT part,quantity supply_2011_01_02
> UNION ALL
> SELECT part,quantity from supply_2011_01_03
> WHERE quantity < 4;
```

对于 Hive，这种情况下应该使用分区表。Hive 通过 WHERE 子句中的表达式来选择查询所需要的指定的分区。这样的查询执行效率高，而且看起来清晰明了：

```

hive> CREATE TABLE supply (id int, part string, quantity int)
> PARTITIONED BY (int day);

hive> ALTER TABLE supply add PARTITION (day=20110102);

hive> ALTER TABLE supply add PARTITION (day=20110103);

hive> ALTER TABLE supply add PARTITION (day=20110102);

hive> .... load data ...

hive> SELECT part,quantity FROM supply
> WHERE day>=20110102 AND day<20110103 AND quantity < 4;

```

9.2 关于分区

Hive 中分区的功能是非常有用的。这是因为 Hive 通常要对输入进行全盘扫描，来满足查询条件（这里我们先忽略掉 Hive 的索引功能）。通过创建很多的分区确实可以优化一些查询，但是同时可能会对其他一些重要的查询不利：

```

hive> CREATE TABLE weblogs (url string, time long )
> PARTITIONED BY (day int, state string, city string);

hive> SELECT * FROM weblogs WHERE day=20110102;

```

HDFS 用于设计存储数百万的大文件，而非数十亿的小文件。使用过多分区可能导致的一个问题就是会创建大量的非必须的 Hadoop 文件和文件夹。一个分区就对应着一个包含有多个文件的文件夹。如果指定的表存在数百个分区，那么可能每天都会创建好几万个文件。如果保持这样的表很多年，那么最终就会超出 NameNode 对系统云数据信息的处理能力。因为 NameNode 必须要将所有的系统文件的元数据信息保存在内存中。虽然每个文件只需要少量字节大小的元数据（大约是 150 字节/文件），但是这样也会限制一个 HDFS 实例所能管理的文件总数的上限。而其他的文件系统，比如 MapR 和 Amazon S3 就没有这个限制。

MapReduce 会将一个任务 (job) 转换成多个任务 (task)。默认情况下，每个 task 都是一个新的 JVM 实例，都需要开启和销毁的开销。对于小文件，每个文件都会对应一个 task。在一些情况下，JVM 开启和销毁的时间中销毁可能会比实际处理数据的时间消耗要长！

因此，一个理想的分区方案不应该导致产生太多的分区和文件夹目录，并且每个目录下的文件应该足够得大，应该是文件系统中块大小的若干倍。

按时间范围进行分区的一个好的策略就是按照不同的时间粒度来确定合适大小的数据积累量，而且安装这个时间粒度。随着时间的推移，分区数量的增长是“均匀的”，而且每个分区下包含的文件大小至少是文件系统中块的大小或块大小的数倍。这个平衡可

以保持使分区足够大，从而优化一般情况下查询的数据吞吐量。同时有必要考虑这种粒度级别在未来是否是适用的，特别是查询中 WHERE 子句选择较小粒度的范围的情况：

```
hive> CREATE TABLE weblogs (url string, time long, state string, city
string )
    > PARTITIONED BY (day int);
hive> SELECT * FROM weblogs WHERE day=20110102;
```

另一个解决方案是使用两个级别的分区并且使用不同的维度。例如，第一个分区可能是按照天(day)进行划分的，而二级分区可能通过如州名 (state) 这样的地理区域进行划分：

```
hive> CREATE TABLE weblogs (url string, time long, city string )
    > PARTITIONED BY (day int, state string);
hive> SELECT * FROM weblogs WHERE day=20110102;
```

然而，由于一些州可能会比其他州具有更多的数据，用户可能会发现 map task 处理数据时会出现不均，这是因为处理数据量多的州需要比处理数据量小的州要消耗更多的时间。

如果用户不能够找到好的、大小相对合适的分区方式的话，那么可以考虑使用第 9.6 节“分桶表数据存储”中介绍的分桶存储。

9.3 唯一键和标准化

关系型数据库通常使用唯一键、索引和标准化来存储数据集，通常是全部或者大部分存储到内存的。然而，Hive 没有主键或基于序列密钥生成的自增键的概念。如果可以的话，应避免对非标准化数据进行连接 (JOIN) 操作。复杂的数据类型，如 array、map 和 struct，有助于实现在单行中存储一对多数据。这并不是说不应该进行标准化，但是星形架构类型设计并非最优的。

避免标准化的主要的原因是为了最小化磁盘寻道，比如那些通常需要外键关系的情况。非标准化数据允许被扫描或写入到大的、连续的磁盘存储区域，从而优化磁盘驱动器的 I/O 性能。然而，非标准化数据可能导致数据重复，而且有更大的导致数据不一致的风险。

例如，思考下我们的运行示例——员工(employees)表。为了清晰地表述，这里再次做了一些修改：

```
CREATE TABLE employees (
name          STRING,
salary       FLOAT,
subordinates  ARRAY<STRING>,
```

```
deductions    MAP<STRING, FLOAT>
address       STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>;
```

这个例子中的数据模型从很多方面打破了传统的设计原则。

首先，我们非正式地使用 `name` 作为主键，而我们都知道名字往往不是唯一的！现在暂且忽略这个问题。一个关系模型中如果使用 `name` 作为键，那么从一个员工记录到经理记录应该有唯一的一个外键关系。这里我们使用了另一种方式来表达这个关系，即在 `subordinates` 数组字段中保存了这个员工所有的下属的名字。

其次，对于每名员工来说，其各项税收扣除额都是不同的，但是 `map` 的键都是一样的，即用户使用“标记”（例如，整数）来作为键实际对应的值。一个常规的关系模型通常使用一个单独的、具有 2 个列的表来记录税收扣除项的扣除名称(或标志)和具体的值，而员工表和这个税收扣除项之间是一对多的关系。

最后，有些雇员还是有可能住在同一个地址的，但是我们为每个雇员都记录了其对应的住址，而不是使用一个雇员住址表，然后和雇员表建立一对一的关系。

下面轮到管理引用完整性（或处理结果），然后解决特定的发生了改变的数据中的重复数据。Hive 本身没有提供方便的方式来对单行数据执行 `UPDATE` 操作。

不过，当用户的数据量达到数十 TB 到 PB 级别时，相对于这些局限性而言，优化执行速度显得更加重要。

9.4 同一份数据多种处理

Hive 本身提供了一个独特的语法，它可以从一个数据源产生多个数据聚合，而无需每次聚合都要重新扫描一次。对于大的数据输入集来说，这个优化可以节约非常可观的时间。我们会在第 5 章详细讨论这个问题。

例如，下面这 2 个查询都会从源表 `history` 表读取数据，然后导入到 2 个不同的表中：

```
hive> INSERT OVERWRITE TABLE sales
> SELECT * FROM history WHERE action='purchased';
hive> INSERT OVERWRITE TABLE credits
> SELECT * FROM history WHERE action='returned';
```

上面的查询，语法是正确的，不过执行效率低下。而如下这个查询可以达到同样的目的，却只需要扫描 `history` 表一次就可以：

```
hive> FROM history
> INSERT OVERWRITE sales SELECT * WHERE action='purchased'
> INSERT OVERWRITE credits SELECT * WHERE action='returned';
```

9.5 对于每个表的分区

很多的 ETL 处理过程会涉及到多个处理步骤，而每个步骤可能会产生一个或多个临时表，这些表仅供下一个 job 使用。起先可能会觉得将这些临时表进行分区不是那么有必要的。不过，想象一下这样的场景：由于查询或者原始数据处理的某个步骤出现问题而导致需要对好几天的输入数据重跑 ETL 过程。这时用户可能就需要执行那些一天执行一次的处理过程，来保证在所有的任务都完成之前不会有 job 将临时表覆盖重写。

例如，下面这个例子中设计了一个名为 `distinct_ip_in_logs` 的中间表，其会在后续处理步骤中使用到：

```
$ hive -hiveconf dt=2011-01-01
hive> INSERT OVERWRITE table distinct_ip_in_logs
  > SELECT distinct(ip) as ip from weblog
  > WHERE hit_date='${hiveconf:dt}';

hive> CREATE TABLE state_city_for_day (state string,city string);

hive> INSERT OVERWRITE state_city_for_day
  > SELECT distinct(state,city) FROM distinct_ip_in_logs
  > JOIN geodata ON (distinct_ip_in_logs.ip=geodata.ip);
```

这种方式是有效的，不过当计算某一天的数据时会导致前一天的数据被 `INSERT OVERWRITE` 语句覆盖掉。如果同时运行两个这样的实例，用于处理不同日期的数据的话，那么它们就可能会相互影响到对方的结果数据。

一个更具鲁棒性的处理方法是在整个过程中使用分区。这样就不会存在同步问题。同时，这样还能带来一个好处，那就是可以允许用户对中间数据按日期进行比较：

```
$ hive -hiveconf dt=2011-01-01
hive> INSERT OVERWRITE table distinct_ip_in_logs
  > PARTITION (hit_date=${dt})
  > SELECT distinct(ip) as ip from weblog
  > WHERE hit_date='${hiveconf:dt}';

hive> CREATE TABLE state_city_for_day (state string,city string)
  > PARTITIONED BY (hit_date string);

hive> INSERT OVERWRITE table state_city_for_day PARTITION('${hiveconf:dt}')
  > SELECT distinct(state,city) FROM distinct_ip_in_logs
  > JOIN geodata ON (distinct_ip_in_logs.ip=geodata.ip)
  > WHERE (hit_date='${hiveconf:dt}');
```

这种方法的一个缺点是，用户将需要管理中间表并删除旧分区，不过这些任务也很容易实现自动化处理。

9.6 分桶表数据存储

分区提供一个隔离数据和优化查询的便利的方式。不过，并非所有的数据集都可形成合理的分区，特别是之前所提到过的要确定合适的划分大小这个疑虑。

分桶是将数据集分解成更容易管理的若干部分的另一个技术。

例如，假设有个表的一级分区是 `dt`，代表日期，二级分区是 `user_id`，那么这种划分方式可能会导致太多的小分区。回想下，如果用户是使用动态分区来创建这些分区的话，那么默认情况下，Hive 会限制动态分区可以创建的最大分区数，用来避免由于创建太多的分区导致超过了文件系统的处理能力以及其他一些问题。因此，如下命令可能会执行失败：

```
hive> CREATE TABLE weblog (url STRING, source_ip STRING)
> PARTITIONED BY (dt STRING, user_id INT);

hive> FROM raw_weblog
> INSERT OVERWRITE TABLE page_view PARTITION(dt='2012-06-08', user_id)
> SELECT server_name, url, source_ip, dt, user_id;
```

不过，如果我们对表 `weblog` 进行分桶，并使用 `user_id` 字段作为分桶字段，则字段值会根据用户指定的值进行哈希分发到桶中。同一个 `user_id` 下的记录通常会存储到同一个桶内。假设用户数要比桶数多得多，那么每个桶内就将会包含多个用户的记录：

```
hive> CREATE TABLE weblog (user_id INT, url STRING, source_ip STRING)
> PARTITIONED BY (dt STRING)
> CLUSTERED BY (user_id) INTO 96 BUCKETS;
```

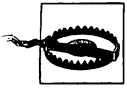
不过，将数据正确地插入到表的过程完全取决于用户自己！`CREATE TABLE` 语句中所规定的信息仅仅定义了元数据，而不影响实际填充表的命令。

下面介绍如何在使用 `INSERT ... TABLE` 语句时，正确地填充表。首先，我们需要设置一个属性来强制 Hive 为目标表的分桶初始化过程设置一个正确的 `reducer` 个数。然后我们再执行一个查询来填充分区。例如：

```
hive> SET hive.enforce.bucketing = true;

hive> FROM raw_logs
> INSERT OVERWRITE TABLE weblog
> PARTITION (dt='2009-02-25')
> SELECT user_id, url, source_ip WHERE dt='2009-02-25';
```

如果我们没有使用 `hive.enforce.bucketing` 属性，那么我们就需要自己设置和分桶个数相匹配的 `reducer` 个数，例如，使用 `set mapred.reduce.tasks=96`，然后在 `INSERT` 语句中，需要在 `SELECT` 语句后增加 `CLUSTER BY` 语句。



警告

对于所有表的元数据，指定分桶并不能保证表可以正确地填充。用户可以根据前面的示例来确保是否正确地填充了表。

分桶有几个优点。因为桶的数量是固定的，所以它没有数据波动。桶对于抽样再合适不过。如果两个表都是按照 `user_id` 进行分桶的话，那么 Hive 可以创建一个逻辑上正确的抽样。分桶同时有利于执行高效的 `map-side JOIN`，正如我们在第 6.4.9 节“`map-side JOIN`”中讨论过的一样。

9.7 为表增加列

Hive 允许在原始数据文件之上定义一个模式，而不像很多的数据库那样，要求必须以特定的格式转换和导入数据。这样的分离方式的好处是，当为数据文件增加新的字段时，可以容易地适应表定义的模式。

Hive 提供了 `SerDe` 抽象，其用于从输入中提取数据。`SerDe` 同样用于输出数据，尽管输出功能并非经常使用，因为 Hive 主要用于查询。一个 `SerDe` 通常是从左到右进行解析的。通过指定的分隔符将行分解成列。`SerDe` 通常是非常宽松的。例如，如果某行的字段个数比预期的要少，那么缺少的字段将返回 `null`。如果某行的字段个数比预期的要多，那么多出的字段将会被省略掉。增加新字段的命令只需要一条 `ALTER TABLE ADD COLUMN` 命令就可完成。因为日志格式通常是对已有字段增加更多信息，所以这样是非常有用的：

```
hive> CREATE TABLE weblogs (version LONG, url STRING)
> PARTITIONED BY (hit_date int)
> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

hive> ! cat log1.txt
1 /mystuff
1 /toys

hive> LOAD DATA LOCAL INPATH 'log1.txt' int weblogs partition(20110101);

hive> SELECT * FROM weblogs;
1 /mystuff 20110101
1 /toys 20110101
```

随着时间的推移，可能会为底层数据增加一个新字段。如下这个例子就是展示为数据新增 `user_id` 字段的过程。需要注意的是，一些旧的原始数据文件可能不包含这个字段：

```
hive> ! cat log2.txt
2 /cars bob
2 /stuff terry
```

```
hive> ALTER TABLE weblogs ADD COLUMNS (user_id string);

hive> LOAD DATA LOCAL INPATH 'log2.txt' int weblogs partition(20110102);

hive> SELECT * from weblogs
1 /mystuff      20110101 NULL
1 /toys         20110101 NULL
2 /cars         20110102 bob
2 /stuff        20110102 terry
```

需要注意的是，通过这种方式，无法在已有字段的开始或者中间增加新字段。

9.8 使用列存储表

Hive 通常使用行式存储，不过 Hive 也提供了一个列式 SerDe 来以混合列式格式存储信息。虽然这种格式是可以用于任意类型的数据的，不过对于某些数据集使用这种方式是最优的。

9.8.1 重复数据

假设有足够多的行，像 state 字段和 age 字段这样的列将会有很多重复的数据。这种类型的数据如果使用列式存储将会非常好，如表 9-1 所示。

表 9-1 有 3 个字段的样例表

state	uid	age
NY	Bob	40
NJ	Sara	32
NY	Peter	14
NY	Sandra	4

9.8.2 多列

表 9-2 具有非常多的字段。

表 9-2 有众多字段的样例表

state	uid	age	server	tz	many_more...
NY	Bob	40	web1	est	stuff
NJ	Sara	32	web1	est	stuff
NY	Peter	14	web3	pst	stuff
NY	Sandra	4	web45	pst	stuff

查询通常只会使用到一个字段或者很少的一组字段。基于列式存储将会使分析表数据执行得更快：

```
hive> SELECT distinct(state) from weblogs;  
NY  
NJ
```

用户可以参考第 15.3.2 节“RCFile”内容来看看如何使用这种格式。

9.9 （几乎）总是使用压缩

几乎在所有情况下，压缩都可以使磁盘上存储的数据量变小，这样可以通过降低 I/O 来提高查询执行速度。Hive 可以无缝地使用很多压缩类型。不使用压缩唯一令人信服的理由就是产生的数据用于外部系统，或者非压缩格式（例如文本格式）是最兼容的。

但是压缩和解压缩都会消耗 CPU 资源。MapReduce 任务往往是 I/O 密集型的，因此 CPU 开销通常不是问题。不过，对于工作流这样的 CPU 密集型的场景，例如一些机器学习算法，压缩实际上可能会从更多必要的操作中获取宝贵的 CPU 资源，从而降低性能。

第 11 章“其他文件格式和压缩方法”中有关于如何使用压缩更详细的介绍。

HiveQL 是一种声明式语言，用户会提交声明式的查询，而 Hive 会将其转换成 MapReduce job。大多数情况，用户不需要了解 Hive 内部是如何工作的，这样可以专注于手头上的事情。而内部复杂的查询解析、规划、优化和执行过程是由 Hive 开发团队多年的艰难工作来实现的，不过大部分时间用户可以直接无视这些内部逻辑。

不过，当用户对于 Hive 具有越来越多的经验后，学习下 Hive 背后的理论知识以及底层的一些实现细节，会让用户更加高效地使用 Hive。

本章将会分几种不同的议题来介绍 Hive 性能调优。一些调优涉及到调整配置参数的值（就像“调整旋钮”一样），而其他一些调优过程则包括启用或者禁用某些特定的特性。

10.1 使用 EXPLAIN

学习 Hive 是如何工作的（在读完本书之后）第一个步骤就是学习 EXPLAIN 功能，其可以帮助我们学习 Hive 是如何将查询转化成 MapReduce 任务的。

思考下面这个例子：

```
hive> DESCRIBE onecol;
number int

hive> SELECT * FROM onecol;
5
5
4

hive> SELECT SUM(number) FROM onecol;
14
```

现在，我们在前面例子中最后一个查询语句前加上 EXPLAIN 关键字，然后来查询下查询计划和其他一些信息。这个查询本身是不会执行的。

```
hive> EXPLAIN SELECT SUM(number) FROM onecol;
```

上面查询语句的输出需要一些解释和实践才能够理解。

首先，会打印出抽象语法树。它表明 Hive 是如何将查询解析成 token(符号)和 literal(字面值的)，是第一步将查询转化到最终结果的一部分。

```
ABSTRACT SYNTAX TREE:
(TOK_QUERY
 (TOK_FROM (TOK_TABREF (TOK_TABNAME onecol)))
 (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))
 (TOK_SELECT
 (TOK_SELEXPR
 (TOK_FUNCTION sum (TOK_TABLE_OR_COL number))))))
```

(为了适应页面的展示，这里对实际的输出信息进行了缩进处理。)

对于那些不熟悉解析器和分词器的用户来说，这个看起来令人难以应付。不过，即使用户是这个领域的新手，也是可以来研究下这个输出信息的。这样可以方便用户了解 Hive 是怎么处理 SQL 语句的。(作为第一步，用户可以忽略掉 TOK_这个前缀来查看输出信息。)

尽管我们的查询会将其输出写入到控制台，但 Hive 实际上会先将输出写入到一个临时文件中，正如下面输出信息中所表示的：

```
'(TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))'
```

接下来，我们可以看到列名 number，以及表名 onecol，还有 sum 函数。

一个 Hive 任务会包含有一个或多个 *stage(阶段)*，不同的 *stage* 间会存在着依赖关系。正如用户可以预料到的，越复杂的查询通常将会引入越多的 *stage*，而通常 *stage* 越多就需要越多的时间来完成任务。

一个 *stage* 可以是一个 MapReduce 任务，也可以是一个抽样阶段，或者一个合并阶段，还可以是一个 limit 阶段，以及 Hive 需要的其他某个任务的一个阶段。默认情况下，Hive 会一次只执行一个 *stage(阶段)*，不过我们稍后将在第 10.6 节“并行执行”中讨论如何并行执行。

某些阶段执行时间很短，像将文件转移到其他路径下这样的操作，其他一些阶段如果数据量小，那么执行也是很快的，即使有时它们需要一个 map 过程或 reduce 过程：

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-0 is a root stage
```

STAGE PLAN 部分比较冗长也比较复杂。Stage-1 包含了这个 job 的大部分处理过程，而且会触发一个 MapReduce job。TableScan 以这个表作为输入，然后会产生一个只有字段 number 的输出。Group By Operator 会应用到 sum(number)，然后会产生一个输出字段_col0(这是为临时结果字段按规则起的临时字段名)。这些都是发生在 job 的 map 处理阶段过程，也就是都是位于 Map Operator Tree 下面的：

```
STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Alias -> Map Operator Tree:
        onecol
          TableScan
            alias: onecol
            Select Operator
              expressions:
                expr: number
                type: int
            outputColumnNames: number
            Group By Operator
              aggregations:
                expr: sum(number)
            bucketGroup: false
            mode: hash
            outputColumnNames: _col0
            Reduce Output Operator
              sort order:
                tag: -1
              value expressions:
                expr: _col0
                type: bigint
```

在 reduce 过程这边，也就是 Reduce Operator Tree 下面，我们可以看到相同的 Group by Operator，但是这次其应用到的是对_col0 字段进行 sum 操作。最后，在 reducer 中我们看到了 File Output Operator，其说明了输出结果将是文本格式，是基于字符串的输出格式：HiveIgnoreKeyTextOutputFormat；

```
Reduce Operator Tree:
  Group By Operator
    aggregations:
      expr: sum(VALUE._col0)
    bucketGroup: false
    mode: mergepartial
    outputColumnNames: _col0
    Select Operator
      expressions:
        expr: _col0
        type: bigint
```

```

outputColumnNames: _col0
File Output Operator
  compressed: false
  GlobalTableId: 0
  table:
    input format: org.apache.hadoop.mapred.TextInputFormat
    output format:
      org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

```

因为这个 job 没有 LIMIT 语句，因此 Stage-0 阶段是一个没有任何操作的阶段：

```

Stage: Stage-0
  Fetch Operator
    limit: -1

```

理解 Hive 是如何对每个查询进行解析和计划的复杂的细节并非总是有用的。不过，这是分析复杂的或执行效率低的查询的一个不错的方式，特别是当我们需要尝试各种各样的调优方式时。我们可以在“逻辑”层查看到这些调整会产生什么样的影响，这样可以关联到性能的度量。

10.2 EXPLAIN EXTENDED

使用 EXPLAIN EXTENDED 语句可以产生更多的输出信息。为了方便起见，我们不会展示完整的输出，不过我们将会展示 Reduce Operator Tree 来演示输出的不同点：

```

Reduce Operator Tree:
  Group By Operator
    aggregations:
      expr: sum(VALUE._col0)
    bucketGroup: false
    mode: mergepartial
    outputColumnNames: _col0
  Select Operator
    expressions:
      expr: _col0
      type: bigint
    outputColumnNames: _col0
  File Output Operator
    compressed: false
    GlobalTableId: 0
    directory: file:/tmp/edward/hive_2012-[long number]/-ext-10001
    NumFilesPerFileSink: 1
    Stats Publishing Key Prefix:
      file:/tmp/edward/hive_2012-[long number]/-ext-10001/
    table:
      input format: org.apache.hadoop.mapred.TextInputFormat
      output format:

```



```

                                org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOu
tputFormat
                                properties:
                                    columns _col0
                                    columns.types bigint
                                    escape.delim \
                                    serialization.format 1
                                TotalFiles: 1
                                GatherStats: false
                                MultiFileSpray: false

```

强烈建议用户比较下这两个 Reduce Operator Tree 下的输出。

10.3 限制调整

LIMIT 语句是大家经常使用到的，经常使用 CLI 的用户都会使用到。不过，在很多情况下 LIMIT 语句还是需要执行整个查询语句，然后再返回部分结果的。因为这种情况通常是浪费的，所以应该尽可能地避免出现这种情况。Hive 有一个配置属性可以开启，当使用 LIMIT 语句时，其可以对源数据进行抽样：

```

<property>
  <name>hive.limit.optimize.enable</name>
  <value>true</value>
  <description>Whether to enable to optimization to
    try a smaller subset of data for simple LIMIT first.</description>
</property>

```

一旦属性 `hive.limit.optimize.enable` 的值设置为 `true`，那么还会有两个参数可以控制这个操作，也就是 `hive.limit.row.max.size` 和 `hive.limit.optimize.limit.file`：

```

<property>
  <name>hive.limit.row.max.size</name>
  <value>100000</value>
  <description>When trying a smaller subset of data for simple LIMIT,
    how much size we need to guarantee each row to have at least.
  </description>
</property>

<property>
  <name>hive.limit.optimize.limit.file</name>
  <value>10</value>
  <description>When trying a smaller subset of data for simple LIMIT,
    maximum number of files we can sample.</description>
</property>

```

这个功能的一个缺点就是，有可能输入中有用的数据永远不会被处理到，例如，像任

意的一个需要 reduce 步骤的查询，JOIN 和 GROUP BY 操作，以及聚合函数的大多数调用，等等，将会产生很不同的结果。也许这个差异在很多情况下是可以接受的，但是重要的是要理解。

10.4 JOIN 优化

在第 6.4.2 节“JOIN 优化”和第 6.4.9 节“map-side JOIN”中我们讨论过如何优化 job 性能。这里我们不会再次重新进行说明，不过需要提醒自己要清楚哪个表是最大的，并将最大的表放置在 JOIN 语句的最右边，或者直接使用/* streamtable(table_name) */ 语句指出。

如果所有表中有一个表足够得小，是可以完成载入到内存中的，那么这时 Hive 可以执行一个 map-side JOIN，这样可以减少 reduce 过程，有时甚至可以减少某些 map task 任务。有时候即使某些表不适合载入内存也可以使用 mapJOIN，因为减少 reduce 阶段可能比将不太大的表分发到每个 map task 中会带来更多的好处。

10.5 本地模式

大多数的 Hadoop Job 是需要 Hadoop 提供的完整的可扩展性来处理大数据集的。不过，有时 Hive 的输入数据量是非常小的。在这种情况下，为查询触发执行任务的时间消耗可能会比实际 job 的执行时间要多得多。对于大多数这种情况，Hive 可以通过本地模式在单台机器上（或某些时候在单个进程中）处理所有的任务。对于小数据集，执行时间可以明显被缩短。

用户可以按照如下这个例子中所演示的方式，在执行过程中临时启用本地模式：

```
hive> set oldjobtracker=${hiveconf:mapred.job.tracker};

hive> set mapred.job.tracker=local;

hive> set mapred.tmp.dir=/home/edward/tmp;

hive> SELECT * from people WHERE firstname=bob;
...

hive> set mapred.job.tracker=${oldjobtracker};
```

用户可以通过设置属性 hive.exec.mode.local.auto 的值为 true，来让 Hive 在适当的时候自动启动这个优化。通常用户可以将这个配置写在 \$HOME/.hiverc 文件中。

如果希望对所有的用户都使用这个配置，那么可以将这个配置项增加到 \$HIVE_HOME/conf/hive-site.xml 文件中：

```
<property>
  <name>hive.exec.mode.local.auto</name>
  <value>true</value>
  <description>
    Let hive determine whether to run in local mode automatically
  </description>
</property>
```

10.6 并行执行

Hive 会将一个查询转化成一個或者多个阶段。这样的阶段可以是 MapReduce 阶段、抽样阶段、合并阶段、limit 阶段，或者 Hive 执行过程中可能需要的其他阶段。默认情况下，Hive 一次只会执行一个阶段。不过，某个特定的 job 可能包含众多的阶段，而这些阶段可能并非完全互相依赖的，也就是说有些阶段是可以并行执行的，这样可能使得整个 job 的执行时间缩短。不过，如果有更多的阶段可以并行执行，那么 job 可能就越快完成。

通过设置参数 `hive.exec.parallel` 值为 `true`，就可以开启并发执行。不过，在共享集群中，需要注意下，如果 job 中并行执行的阶段增多，那么集群利用率就会增加：

```
<property>
  <name>hive.exec.parallel</name>
  <value>true</value>
  <description>Whether to execute jobs in parallel</description>
</property>
```

10.7 严格模式

Hive 提供了一个严格模式，可以防止用户执行那些可能产生意想不到的不好的影响的查询。

通过设置属性 `hive.mapred.mode` 值为 `strict` 可以禁止 3 种类型的查询。

其一，对于分区表，除非 WHEHRE 语句中含有分区字段过滤条件来限制数据范围，否则不允许执行。换句话说，就是用户不允许扫描所有分区。进行这个限制的原因是，通常分区表都拥有非常大的数据集，而且数据增加迅速。没有进行分区限制的查询可能会消耗令人不可接受的巨大资源来处理这个表：

```
hive> SELECT DISTINCT(planner_id) FROM fracture_ins WHERE planner_id=5;
FAILED: Error in semantic analysis: No Partition Predicate Found for
Alias "fracture_ins" Table "fracture_ins"
```

如下这个语句在 WHERE 语句中增加了一个分区过滤条件（也就是限制了表分区）：

```
hive> SELECT DISTINCT(planner_id) FROM fracture_ins
> WHERE planner_id=5 AND hit_date=20120101;
... normal results ...
```

其二，对于使用了 ORDER BY 语句的查询，要求必须使用 LIMIT 语句。因为 ORDER BY 为了执行排序过程会将所有的结果数据分发到同一个 reducer 中进行处理，强制要求用户增加这个 LIMIT 语句可以防止 reducer 额外执行很长一段时间：

```
hive> SELECT * FROM fracture_ins WHERE hit_date>2012 ORDER BY planner_id;
FAILED: Error in semantic analysis: line 1:56 In strict mode,
limit must be specified if ORDER BY is present planner_id
```

只需要增加 LIMIT 语句就可以解决这个问题：

```
hive> SELECT * FROM fracture_ins WHERE hit_date>2012 ORDER BY planner_id
> LIMIT 100000;
... normal results ...
```

其三，也就是最后一种情况，就是限制笛卡尔积的查询。对关系型数据库非常了解的用户可能期望在执行 JOIN 查询的时候不使用 ON 语句而是使用 WHERE 语句，这样关系型数据库的执行优化器就可以高效地将 WHERE 语句转化成那个 ON 语句。不幸的是，Hive 并不会执行这种优化，因此，如果表足够大，那么这个查询就会出现不可控的情况：

```
hive> SELECT * FROM fracture_act JOIN fracture_ads
> WHERE fracture_act.planner_id = fracture_ads.planner_id;
FAILED: Error in semantic analysis: In strict mode, cartesian product
is not allowed. If you really want to perform the operation,
+set hive.mapred.mode=nonstrict+
```

下面这个才是个正确的使用 JOIN 和 ON 语句的查询：

```
hive> SELECT * FROM fracture_act JOIN fracture_ads
> ON (fracture_act.planner_id = fracture_ads.planner_id);
... normal results ...
```

10.8 调整 mapper 和 reducer 个数

Hive 通过将查询划分成一个或者多个 MapReduce 任务达到并行的目的。每个任务都可能具有多个 mapper 和 reducer 任务，其中至少有一些是可以并行执行的。确定最佳的 mapper 个数和 reducer 个数取决于多个变量，例如输入的数据量大小以及对这些数据执行的操作类型等。

保持平衡性是有必要的。如果有太多的 mapper 或 reducer 任务，就会导致启动阶段、调度和运行 job 过程中产生过多的开销；而如果设置的数量太少，那么就on能没有充分

利用好集群内在的并行性。

当执行的 Hive 查询具有 reduce 过程时，CLI 控制台会打印出调优后的 reducer 个数。下面我们来看一个包含有 GROUP BY 语句的例子，因为这种查询总是需要 reduce 过程的。与此相反，很多其他查询会转换成只需要 map 阶段的任务：

```
hive> SELECT pixel_id, count FROM fracture_ins WHERE hit_date=20120119
  > GROUP BY pixel_id;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 3
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
...
```

Hive 是按照输入的数据量大小来确定 reducer 个数的。我们可以通过 `dfs -count` 命令来计算输入量大小，这个命令和 Linux 中的 `du -s` 命令类似；其可以计算指定目录下所有数据的总大小：

```
[edward@etl02 ~]$ hadoop dfs -count /user/media6/fracture/ins/* | tail -4
1 8 2614608737 hdfs://.../user/media6/fracture/ins/hit_date=20120118
1 7 2742992546 hdfs://.../user/media6/fracture/ins/hit_date=20120119
1 17 2656878252 hdfs://.../user/media6/fracture/ins/hit_date=20120120
1 2 362657644 hdfs://.../user/media6/fracture/ins/hit_date=20120121
```

(为了方便排版查看，我们对实际的数据进行了格式化并省略掉了一些详细信息。)

属性 `hive.exec.reducers.bytes.per.reducer` 的默认值是 1GB。如果将这个属性值调整为 750MB 的话，那么下面这个任务 Hive 就会使用 4 个 reducer：

```
hive> set hive.exec.reducers.bytes.per.reducer=750000000;

hive> SELECT pixel_id, count(1) FROM fracture_ins WHERE hit_date=20120119
  > GROUP BY pixel_id;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 4
...
```

默认值通常情况下是比较合适的。不过，有些情况下查询的 map 阶段会产生比实际输入数据量要多得多的数据。如果 map 阶段产生的数据量非常多，那么根据输入的数据量大小来确定的 reducer 个数就显得有些少了。同样地，map 阶段也可能会过滤掉输入数据集中的很大一部分的数据而这时可能需要少量的 reducer 就满足计算了。

一个快速的进行验证的方式就是将 reducer 个数设置为固定的值，而无需 Hive 来计算得到这个值。如果用户还记得的话，Hive 的默认 reducer 个数应该是 3。可以通过设置属性 `mapred.reduce.tasks` 的值为不同的值来确定是使用较多还是较少的 reducer 来缩短执行时间。需要记住，受外部因素影响，像这样的标杆值十分复杂，例如其他用户并发执行 job 的情况。Hadoop 需要消耗好几秒时间来启动和调度 map 和 reduce 任务 (task)。在进行性能测试的时候，要考虑到这些影响因子，特别是 job 比较小的时候。

当在共享集群上处理大任务时，为了控制资源利用情况，属性 `hive.exec.reducers.max` 显得非常重要。一个 Hadoop 集群可以提供的 map 和 reduce 资源个数 (也称为“插槽”)，是固定的。某个大 job 可能就会消耗完所有的插槽，从而导致其他 job 无法执行。通过设置属性 `hive.exec.reducers.max` 可以阻止某个查询消耗太多的 reducer 资源。有必要将这个属性配置到 `$HIVE_HOME/conf/hive-site.xml` 文件中。对这个属性值大小的一个建议的计算公式如下：

$(\text{集群总 Reduce 槽位个数} * 1.5) / (\text{执行中的查询的平均个数})$
1.5 倍数是一个经验系数，用于防止未充分利用集群的情况。

10.9 JVM 重用

JVM 重用是 Hadoop 调优参数的内容，其对 Hive 的性能具有非常大的影响，特别是对于很难避免小文件的场景或 task 特别多的场景，这类场景大多数执行时间都很短。

Hadoop 的默认配置通常是使用派生 JVM 来执行 map 和 reduce 任务的。这时 JVM 的启动过程可能会造成相当大的开销，尤其是执行的 job 包含有成百上千个 task 任务的情况。JVM 重用可以使得 JVM 实例在同一个 job 中重新使用 N 次。N 的值可以在 Hadoop 的 `mapred-site.xml` 文件 (位于 `$HADOOP_HOME/conf` 目录下) 中进行设置：

```
<property>
  <name>mapred.job.reuse.jvm.num.tasks</name>
  <value>10</value>
  <description>How many tasks to run per jvm. If set to -1, there is no
limit.
  </description>
</property>
```

这个功能的一个缺点是，开启 JVM 重用将会一直占用使用到的 task 插槽，以便进行重用，直到任务完成后才能释放。如果某个“不平衡的” job 中有某几个 reduce task 执行的时间要比其他 reduce task 消耗的时间多得多的话，那么保留的插槽就会一直空着却无法被其他的 job 使用，直到所有的 task 都结束了才会释放。

10.10 索引

索引可以用来加快含有 GROUP BY 语句的查询的计算速度。

Hive 从 v0.8.0 版本后增加了一个 bitmap 索引实现。Bitmap 索引一般在指定的列排重后的值比较小时进行使用。请参考第 8.1.1 节“Bitmap 索引”中的详细说明。

10.11 动态分区调整

正如在第 5.2.1 节“动态分区插入”中所讲解过的一样，动态分区 INSERT 语句可以通过简单的 SELECT 语句向分区表中创建很多新的分区。

这是一个非常强大的功能，不过如果分区的个数非常多，那么就会在系统中产生大量的输出控制流。对于 Hadoop 来说，这个种情况并不是常见的使用场景，因此，其通常会一次创建很多的文件，然后会向这些文件中写入大量的数据。

跳出这些框框，Hive 可以通过配置限制动态分区插入允许创建的分区数在 1000 个左右。虽然太多的分区对于表来说并不好，不过，通常还是将这个值设置的更大以便这些查询执行。

首先，通常在 hive-site.xml 配置文件中设置动态分区模式为严格模式（也就是属性值为 strict），这个配置在第 10.7 节“严格模式”中有介绍过。开启严格模式的时候，必须保证至少有一个分区是静态的，正如在第 5.2.1 节“动态分区插入”中所展示的。

```
<property>
  <name>hive.exec.dynamic.partition.mode</name>
  <value>strict</value>
  <description>In strict mode, the user must specify at least one
static partition in case the user accidentally overwrites all
partitions.</description>
</property>
```

然后，可以增加一些相关的属性信息，例如通过如下属性来限制查询可以创建的最大动态分区个数：

```
<property>
  <name>hive.exec.max.dynamic.partitions</name>
  <value>300000</value>
  <description>Maximum number of dynamic partitions allowed to be
created in total.</description>
</property>

<property>
  <name>hive.exec.max.dynamic.partitions.pernode</name>
  <value>10000</value>
```

```
<description>Maximum number of dynamic partitions allowed to be
created in each mapper/reducer node.</description>
</property>
```

还有一个配置是来控制 `DataNode` 上一次可以打开的文件的个数。这个参数必须设置在 `DataNode` 的 `$HADOOP_HOME/conf/hdfs-site.xml` 配置文件中。

在 Hadoop v0.20.2 版本中，这个属性的默认值是 256，太小了。这个值会影响到最大的线程数和资源数，因此，也并不推荐将这个属性值设置为一个极大值。同时需要注意的是，在 Hadoop v0.20.2 版本中，更改这个属性值需要重启 `DataNode` 才能够生效：

```
<property>
  <name>dfs.datanode.max.xcievers</name>
  <value>8192</value>
</property>
```

10.12 推测执行

推测执行是 Hadoop 中的一个功能，其可以触发执行一些重复的任务 (task)。尽管这样会因对重复的数据进行计算而导致消耗更多的计算资源，不过这个功能的目的是通过加快获取单个 task 的结果以及进行侦测将执行慢的 `TaskTracker` 加入到黑名单的方式来提高整体的任务执行效率。

Hadoop 的推测执行功能由 `$HADOOP_HOME/conf/mapred-site.xml` 文件中的如下 2 个配置项控制着：

```
<property>
  <name>mapred.map.tasks.speculative.execution</name>
  <value>true</value>
  <description>If true, then multiple instances of some map tasks
  may be executed in parallel.</description>
</property>

<property>
  <name>mapred.reduce.tasks.speculative.execution</name>
  <value>true</value>
  <description>If true, then multiple instances of some reduce tasks
  may be executed in parallel.</description>
</property>
```

不过，Hive 本身也提供了配置项来控制 `reduce-side` 的推测执行：

```
<property>
  <name>hive.mapred.reduce.tasks.speculative.execution</name>
  <value>true</value>
  <description>Whether speculative execution for
  reducers should be turned on. </description>
```



```
</property>
```

关于调优这些推测执行变量，还很难给一个具体的建议。如果用户对于运行时的偏差非常敏感的话，那么可以将这些功能关闭掉。如果用户因为输入数据量很大而需要执行长时间的 map 或者 reduce task 的话，那么启动推测执行造成的浪费是非常巨大的。

10.13 单个 MapReduce 中多个 GROUP BY

另一个特别的优化试图将查询中的多个 GROUP BY 操作组装到单个 MapReduce 任务中。如果想启动这个优化，那么需要一组常用的 GROUP BY 键：

```
<property>
  <name>hive.multigroupby.singlemr</name>
  <value>>false</value>
  <description>Whether to optimize multi group by query to generate single M/R
  job plan. If the multi group by query has common group by keys, it will be
  optimized to generate single M/R job.</description>
</property>
```

10.14 虚拟列

Hive 提供了 2 种虚拟列：一种用于将要进行划分的输入文件名，另一种用于文件中的块内偏移量。当 Hive 产生了非预期的或 null 的返回结果时，可以通过这些虚拟列诊断查询。通过查询这些“字段”，用户可以查看到哪个文件甚至哪行数据导致出现问题：

```
hive> set hive.exec.rowoffset=true;

hive> SELECT INPUT_FILE_NAME, BLOCK_OFFSET_INSIDE_FILE, line
  > FROM hive_text WHERE line LIKE '%hive%' LIMIT 2;
har://file/user/hive/warehouse/hive_text/folder=docs/
data.har/user/hive/warehouse/hive_text/folder=docs/README.txt 2243
  http://hive.apache.org/

har://file/user/hive/warehouse/hive_text/folder=docs/
data.har/user/hive/warehouse/hive_text/folder=docs/README.txt 3646
- Hive 0.8.0 ignores the hive-default.xml file, though we continue
```

(为了方便排版和查看，我们对输出进行了格式调整并且在输出行中间增加了空行。)

第 3 种虚拟列提供了文件的行偏移量。这个需要通过如下参数显式地启用：

```
<property>
  <name>hive.exec.rowoffset</name>
  <value>>true</value>
  <description>Whether to provide the row offset virtual column</description>
</property>
```

这样设置后就可以在类似于如下的查询中使用了：

```
hive> SELECT INPUT__FILE__NAME, BLOCK__OFFSET__INSIDE__FILE,  
  > ROW__OFFSET__INSIDE__BLOCK  
  > FROM hive_text WHERE line LIKE '%hive%' limit 2;  
file:/user/hive/warehouse/hive_text/folder=docs/README.txt 2243 0  
file:/user/hive/warehouse/hive_text/folder=docs/README.txt 3646 0
```

其他文件格式和压缩方法

Hive 的一个独特的功能就是：Hive 不会强制要求将数据转换成特定的格式才能使用。Hive 利用 Hadoop 的 InputFormat API 来从不同的数据源读取数据，例如文本格式、sequence 文件格式，甚至用户自定义格式。同样地，使用 OutputFormat API 也可以将数据写成不同的格式。

尽管 Hadoop 的文件系统支持对于非压缩数据的线性扩展存储，但是对数据进行压缩还是有很大好处的。压缩通常都会节约可观的磁盘空间，例如，基于文本的文件可以压缩 40%甚至更高比例。压缩同样可以增加吞吐量和性能。这看上去似乎并不合常理，因为压缩和解压缩会增加额外的 CPU 开销，不过，通过减少载入内存的数据量而提高 I/O 吞吐量会更加提高网络传输性能。

Hadoop 的 job 通常是 I/O 密集型而不是 CPU 密集型的。如果是这样的话，压缩可以提高性能。不过，如果用户的 job 是 CPU 密集型的话，那么使用压缩可能会降低执行性能。确定是否进行压缩的唯一方法就是尝试不同的选择，并测量对比执行结果。

11.1 确定安装编解码器

基于用户所使用的 Hadoop 版本，会提供不同的编解码器。Hive 中可以通过 set 命令查看 Hive 配置文件中或 Hadoop 配置文件中配置的值。

通过查看属性 io.compression.codec，可以看到编解码器之间是按照逗号进行分割的：

```
# hive -e "set io.compression.codecs"  
io.compression.codecs=org.apache.hadoop.io.compress.GzipCodec,  
org.apache.hadoop.io.compress.DefaultCodec,  
org.apache.hadoop.io.compress.BZip2Codec,  
org.apache.hadoop.io.compress.SnappyCodec
```

11.2 选择一种压缩编/解码器

使用压缩的优势是可以最小化所需要的磁盘存储空间，以及减小磁盘和网络 I/O 操作。不过，文件压缩过程和解压过程会增加 CPU 开销。因此，对于压缩密集型的 job 最好使用压缩，特别是有额外的 CPU 资源或磁盘存储空间比较稀缺的情况。

所有最新的那些 Hadoop 版本都已经内置支持 GZIP 和 BZip2 压缩方案了，包括加速对这些格式的压缩和解压缩的本地 Linux 库。绑定支持 Snappy 压缩是最近才增加的，不过，如果用户当前使用的 Hadoop 版本不支持该功能的话，那么自行增加相关的库即可¹。另外，还有一种常用的压缩方案，即 LZ0 压缩²。

那么，为什么我们需要不同的压缩方案呢？每一个压缩方案都在压缩/解压缩速度和压缩率间进行权衡。BZip2 压缩率最高，但是同时需要消耗最多的 CPU 开销。GZip 是压缩率和压缩/解压缩速度上的下一个选择。因此，如果磁盘空间利用率和 I/O 开销都需要考虑的话，那么这 2 种压缩方案都是有吸引力的。

LZO 和 Snappy 压缩率相比前面的 2 种要小但是压缩/解压缩速度要快，特别是解压缩过程。如果相对于磁盘空间和 I/O 开销，频繁读取数据所需的解压缩速度更重要的话，那么它们将是不错的选择。

另一个需要考虑的因素是压缩格式的文件是否是可分割的。MapReduce 需要将非常大的输入文件分割成多个划分（通常一个文件块对应一个划分，也就是 64MB 的倍数），其中每个划分会被分发到一个单独的 map 进程中。只有当 Hadoop 知道文件中记录的边界时才可以进行这样的分割。对于文本文件，每一行都是一条记录，但是 GZip 和 Snappy 将这些边界信息掩盖掉了。不过，BZip2 和 LZ0 提供了块（BLOCK）级别的压缩，也就是每个块中都含有完整的记录信息，因此 Hadoop 可以在块边界级别对这些文件进行划分。

虽然 GZip 和 Snappy 压缩的文件不可划分，但是并不能因此而排除它们。当用户创建文件的时候，可以将文件分割成期望的文件大小。通常输出文件的个数等于 reducer 的个数。也就是说如果用户使用了 N 个 reducer，那么通常就会得到 N 个输出文件。需要注意的是，如果有一个不可分割的文件特别的大，那么就会出现一个单独的 task 来读取整个文件，进行处理。

关于压缩我们还有更多的内容要说，不过我们推荐用户参考 Tom White (O' Reilly) 所著的《Hadoop 编程指南》一书，这里，我们会专注于在 Hive 中如何使用指定的格式。

从 Hive 的角度来看，对于文件格式，有 2 个方面的内容需要说明。一个方面是文件是怎样分隔成行（记录）的。文本文件使用 \n(换行符)作为默认的行分隔符。当用户没有

1. See <http://code.google.com/p/hadoop-snappy/>.

2. See <http://wiki.apache.org/hadoop/UsingLzoCompression>.

使用默认的文本文件格式时，用户需要告诉 Hive 使用的 InputFormat 和 OutputFormat 是什么。事实上，用户需要指定对于输入和输出格式实现的 Java 类的名称。InputFormat 中定义了如何读取划分，以及如何将划分分割成记录，而 OutputFormat 中定义了如何将这些划分写回到文件或控制台输出中。

第 2 个方面是记录是如何分割成字段（或列）的。Hive 使用 ^A 作文本文件中默认的分段分隔符。Hive 使用 SerDe（也就是序列化/反序列化的简写）作为对输入记录（反序列化）进行分割以及写记录（序列化）的“模板”。这时，用户只需要指定可以完成这 2 部分工作的一个 Java 类即可。

所有的这些信息用户在创建表的时候都可以在表定义语句中进行指定。创建完成后，用户可以像平时一样查询表，而无需关心底层格式。因此，如果用户是 Hive 的使用者，而不是 Java 工程师，就无需关心关于 Java 的信息。如果需要，用户所在团队的工程师可以帮忙指定需要的定义信息，而之后就可以像之前一样工作了。

11.3 开启中间压缩

对中间数据进行压缩可以减少 job 中 map 和 reduce task 间的数据传输量。对于中间数据压缩，选择一个低 CPU 开销的编/解码器要比选择一个压缩率高的编/解码器要重要得多。属性 `hive.exec.compress.intermediate` 的默认值是 `false`，如果要开启中间压缩，就需要将这个属性值修改为默认值为 `true`：

```
<property>
  <name>hive.exec.compress.intermediate</name>
  <value>true</value>
  <description> This controls whether intermediate files produced by Hive
between
  multiple map-reduce jobs are compressed. The compression codec and other
options
  are determined from hadoop config variables mapred.output.compress*
</description>
</property>
```



提示

对于其他 Hadoop job 来说控制中间数据压缩的属性是 `mapred.compress.map.output`。

Hadoop 压缩默认的编/解码器是 `DefaultCodec`。可以通过修改属性 `mapred.map.output.compression.codec` 的值来修改编/解码器。这是一个 Hadoop 配置项，可以在 `$HADOOP_HOME/conf/mapred-site.xml` 文件中或 `$HADOOP_HOME/conf/hive-site.xml` 文件中进行配置。`SnappyCodec` 是一个比较好的中间文件压缩编/解码器，因为其很好地结合了低 CPU 开销和好的压缩执行效率：

```
<property>
```

```
<name>mapred.map.output.compression.codec</name>
<value>org.apache.hadoop.io.compress.SnappyCodec</value>
<description> This controls whether intermediate files produced by Hive
between multiple map-reduce jobs are compressed. The compression codec
and other options are determined from hadoop config variables
mapred.output.compress* </description>
</property>
```

11.4 最终输出结果压缩

当 Hive 将输出写入到表中时，输出内容同样可以进行压缩。属性 `hive.exec.compress.output` 控制着这个功能。用户可能需要保持默认配置文件中的默认值 `false`，这样默认的输出就是非压缩的纯文本文件了。用户可以通过在查询语句或执行脚本中设置这个值为 `true`，来开启输出结果压缩功能：

```
<property>
  <name>hive.exec.compress.output</name>
  <value>>false</value>
  <description> This controls whether the final outputs of a query
(to a local/hdfs file or a Hive table) is compressed. The compression
codec and other options are determined from hadoop config variables
mapred.output.compress* </description>
</property>
```



提示

对于其他 Hadoop 任务，开启最终输出结果压缩功能的属性是 `mapred.output.compress`。

如果属性 `hive.exec.compress.output` 的值设置为 `true`，那么这时需要为其指定一个编解码器。对于输出文件，使用 GZip 进行压缩是个不错的主意，因为其通常可以大幅度降低文件的大小。但是，需要记住的是 GZip 压缩的文件对于后面的 MapReduce job 而言是不可分割的。

```
<property>
  <name>mapred.output.compression.codec</name>
  <value>org.apache.hadoop.io.compress.GzipCodec</value>
  <description>If the job outputs are compressed, how should they be
compressed?
  </description>
</property>
```

11.5 sequence file 存储格式

压缩文件确实能够节约存储空间，但是，在 Hadoop 中存储裸压缩文件的一个缺点就是，通常这些文件是不可分割的。可分割的文件可以划分成多个部分，由多个 mapper 并行

进行处理。大多数的压缩文件是不可分割的，也就是说只能从头读到尾。

Hadoop 所支持的 `sequence file` 存储格式可以将一个文件划分成多个块，然后采用一种可分割的方式对块进行压缩。

如果想在 Hive 中使用 `sequence file` 存储格式，那么需要在 `CREATE TABLE` 语句中通过 `STORED AS SEQUENCEFILE` 语句进行指定：

```
CREATE TABLE a_sequence_file_table STORED AS SEQUENCEFILE;
```

`Sequence file` 提供了 3 种压缩方式：`NONE`、`RECORD` 和 `BLOCK`，默认是 `RECORD` 级别（也就是记录级别）。不过，通常来说，`BLOCK` 级别（也就是块级别）压缩性能最好而且是可以分割的。和很多其他的压缩属性一样，这个属性也并非 `Hive` 特有的。用户可以在 Hadoop 的 `mapred-site.xml` 文件中进行定义，或者在 `Hive` 的 `hive-site.xml` 文件中进行定义，需要的时候，还可以在脚本中或查询语句前进行指定：

```
<property>
  <name>mapred.output.compression.type</name>
  <value>BLOCK</value>
  <description>If the job outputs are to compressed as SequenceFiles,
  how should they be compressed? Should be one of NONE, RECORD or BLOCK.
  </description>
</property>
```

11.6 使用压缩实践

我们已经介绍了 `Hive` 中可以使用的一些压缩相关的配置属性，这些属性的不同组合方式将会产生不同的输出。下面，让我们在一些例子中使用这些属性，然后展示下它们的输出将会是什么。需要注意的是，`CLI` 中通过 `set` 命令设置的属性在同一个会话中会一直生效的。因此，同一个会话中的不同例子间应该注意将设置复原，或者直接重开启一个 `Hive` 会话：

```
hive> SELECT * FROM a;
4      5
3      2
hive> DESCRIBE a;
a      int
b      int
```

首先，我们先开启中间数据压缩功能。这不会影响到最终输出结果。不过从 `job` 的计数器信息上看，可以发现这个 `job` 中间传送的数据量变小了，因为 `shuffle sort`(混洗排序) 数据被压缩了：

```
hive> set hive.exec.compress.intermediate=true;
hive> CREATE TABLE intermediate_comp_on
  > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
  > AS SELECT * FROM a;
```

```
Moving data to: file:/user/hive/warehouse/intermediate_comp_on
Table default.intermediate_comp_on stats: [num_partitions: 0, num_files: 1,
num_rows: 2, total_size: 8, raw_data_size: 6]
...
```

和预期的一样，中间数据压缩没有影响到最终的输出，最终的数据结果仍然是非压缩的：

```
hive> dfs -ls /user/hive/warehouse/intermediate_comp_on;
Found 1 items
/user/hive/warehouse/intermediate_comp_on/000000_0

hive> dfs -cat /user/hive/warehouse/intermediate_comp_on/000000_0;
4      5
3      2
```

我们同样可以为中间数据压缩配置其他的编/解码器而不使用默认的编/解码器。下面这个例子，我们选择使用 GZip（尽管通常 Snappy 是更好的选择）。为显示清晰，将下面语句中的第 1 行分成了 2 行：

```
hive> set mapred.map.output.compression.codec
      =org.apache.hadoop.io.compress.GZipCodec;
hive> set hive.exec.compress.intermediate=true;

hive> CREATE TABLE intermediate_comp_on_gz
      > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
      > AS SELECT * FROM a;
Moving data to: file:/user/hive/warehouse/intermediate_comp_on_gz
Table default.intermediate_comp_on_gz stats:
[num_partitions: 0, num_files: 1, num_rows: 2, total_size: 8, raw_data_size:6]

hive> dfs -cat /user/hive/warehouse/intermediate_comp_on_gz/000000_0;
4      5
3      2
```

下一步，我们可以开启输出结果压缩：

```
hive> set hive.exec.compress.output=true;

hive> CREATE TABLE final_comp_on
      > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
      > AS SELECT * FROM a;
Moving data to: file:/tmp/hive-edward/hive_2012-01-15_11-11-01_884_.../-ext-10001
Moving data to: file:/user/hive/warehouse/final_comp_on
Table default.final_comp_on stats:
[num_partitions: 0, num_files: 1, num_rows: 2, total_size: 16, raw_data_size:6]

hive> dfs -ls /user/hive/warehouse/final_comp_on;
Found 1 items
```



```
/user/hive/warehouse/final_comp_on/000000_0.deflate
```

输出信息中的表统计信息显示 total_size (总大小) 是 16B, 但是 raw_data_size (裸数据) 是 6B, 多出的存储空间是被 deflate 算法消耗了。同时我们可以看到, 输出的文件后缀名是 deflate。

不推荐使用 cat 命令来查看这个压缩文件, 因为用户只能看到二进制输出。不过, Hive 可以正常地查询这个数据:

```
hive> dfs -cat /user/hive/warehouse/final_comp_on/000000_0.deflate;
... UGLYBINARYHERE ...
```

```
hive> SELECT * FROM final_comp_on;
4          5
3          2
```

这种无缝的处理压缩文件的能力并非是 Hive 独有的, 实际上, 这里是使用了 Hadoop 的 TextInputFormat 进行的处理。尽管在这种情况下这个命名有些令人混淆, 但是 TextInputFormat 可以识别文件后缀名为 deflate 或 gz 的压缩文件, 并且可以很轻松地进行处理。Hive 无需关心底层的文件是否是压缩的, 以及是使用何种压缩方案进行压缩的。

下面我们改变下输出结果压缩所使用的编解码器, 然后看下结果 (这里语句的第 2 行为了显示清晰, 也分成 2 行了):

```
hive> set hive.exec.compress.output=true;
hive> set mapred.output.compression.codec
      =org.apache.hadoop.io.compress.GzipCodec;
hive> CREATE TABLE final_comp_on_gz
      > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
      > AS SELECT * FROM a;
Moving data to: file:/user/hive/warehouse/final_comp_on_gz
Table default.final_comp_on_gz stats:
[num_partitions: 0, num_files: 1, num_rows: 2, total_size: 28, raw_data_size:6]
```

```
hive> dfs -ls /user/hive/warehouse/final_comp_on_gz;
Found 1 items
/user/hive/warehouse/final_comp_on_gz/000000_0.gz
```

正如用户可以看到的, 输出文件夹下现在包含了零个或多个 gz 文件。Hive 提供了在 Hive shell 中执行像 zcat 这样的本地命令的快速方法。通过 ! 符号, Hive 可以执行外部的命令, 直到返回结果。

zcat 是一个命令行实用程序, 用来解压缩, 并进行输出显示:

```
hive> ! /bin/zcat /user/hive/warehouse/final_comp_on_gz/000000_0.gz;
4          5
3          2
```

```
hive> SELECT * FROM final_comp_on_gz;
OK
4      5
3      2
```

Time taken: 0.159 seconds

使用这种输出压缩能够完成那种很小，操作起来很快的文件的二进制压缩。不过，回想一下，输出文件的个数是和处理数据所需要的 mapper 个数或 reducers 个数有关的。在最坏的情况下，可能最终结果是文件夹中只产生了一个大的压缩文件，而且是不可分割的。这意味着后续步骤并不能并行地处理这个数据。这个问题的答案就是使用 **sequence file**：

```
hive> set mapred.output.compression.type=BLOCK;
hive> set hive.exec.compress.output=true;
hive> set mapred.output.compression.codec=org.apache.hadoop.io.compress.
GzipCodec;
```

```
hive> CREATE TABLE final_comp_on_gz_seq
> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
> STORED AS SEQUENCEFILE
> AS SELECT * FROM a;
```

Moving data to: file:/user/hive/warehouse/final_comp_on_gz_seq

Table default.final_comp_on_gz_seq stats:

```
[num_partitions: 0, num_files: 1, num_rows: 2, total_size: 199, raw_data_
size: 6]
```

```
hive> dfs -ls /user/hive/warehouse/final_comp_on_gz_seq;
Found 1 items
/user/hive/warehouse/final_comp_on_gz_seq/000000_0
```

Sequence file 是二进制格式的，但是，我们可以很容易地查询文件头。通过查看文件头可以确认结果是否是我们需要的（为方便显示，输出格式进行了调整）：

```
hive> dfs -cat /user/hive/warehouse/final_comp_on_gz_seq/000000_0;
SEQ[org.apache.hadoop.io.BytesWritable[]org.apache.hadoop.io.BytesWri
table[]
org.apache.hadoop.io.compress.GzipCodec[]
```

因为 **sequence file** 中嵌入的元数据信息以及 Hive 元数据信息，Hive 无需任何特别的设置就可以查询这张表。Hadoop 同样提供了 `dfs -text` 命令来从 **sequence file** 文件中去除掉文件头和压缩，然后显示裸数据：

```
hive> dfs -text /user/hive/warehouse/final_comp_on_gz_seq/000000_0;
4      5
3      2
hive> select * from final_comp_on_gz_seq;
OK
4      5
3      2
```

最后，我们来同时使用直接数据压缩和最终输出数据压缩，而且使用不同压缩编/解码器的 `sequence file`！这些设置通常应用在生产环境，这些环境中的数据量很大，而这些设置可以提高其性能：

```
hive> set mapred.map.output.compression.codec
      =org.apache.hadoop.io.compress.SnappyCodec;
hive> set hive.exec.compress.intermediate=true;
hive> set mapred.output.compression.type=BLOCK;
hive> set hive.exec.compress.output=true;
hive> set mapred.output.compression.codec
      =org.apache.hadoop.io.compress.GzipCodec;

hive> CREATE TABLE final_comp_on_gz_int_compress_snappy_seq
      > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
      > STORED AS SEQUENCEFILE AS SELECT * FROM a;
```

11.7 存档分区

Hadoop 中有一种存储格式名为 HAR，也就是 Hadoop Archive (Hadoop 归档文件) 的缩写。一个 HAR 文件就像在 HDFS 文件系统中的 `TAR` 文件一样是一个单独的文件。不过，其内部可以存放多个文件和文件夹。在一些使用场景下，较旧的文件夹和文件比较新的文件夹和文件被访问的概率要低很多。如果某个特定的分区下保存的文件有成千上万的话，那么就需要 HDFS 中的 NameNode 消耗非常大的代价来管理这些文件。通过将分区下的文件归档成一个巨大的，但是同时可以被 Hive 访问的文件，可以减轻 NameNode 的压力。不过其缺点是，HAR 文件查询效率不高，同时，HAR 文件并非是压缩的，因此也不会节约存储空间。

在下面的例子中，我们将使用 Hive 自带的文件作为表数据。

首先，创建一个分区表，然后将 Hive 包中自带的文件加载到表中：

```
hive> CREATE TABLE hive_text (line STRING) PARTITIONED BY (folder STRING);

hive> ! ls $HIVE_HOME;
LICENSE
README.txt
RELEASE_NOTES.txt

hive> ALTER TABLE hive_text ADD PARTITION (folder='docs');

hive> LOAD DATA INPATH '${env:HIVE_HOME}/README.txt'
      > INTO TABLE hive_text PARTITION (folder='docs');
Loading data to table default.hive_text partition (folder=docs)

hive> LOAD DATA INPATH '${env:HIVE_HOME}/RELEASE_NOTES.txt'
```

```

> INTO TABLE hive_text PARTITION (folder='docs');
Loading data to table default.hive_text partition (folder=docs)

hive> SELECT * FROM hive_text WHERE line LIKE '%hive%' LIMIT 2;
http://hive.apache.org/ docs
- Hive 0.8.0 ignores the hive-default.xml file, though we continue docs

```

某些 Hadoop 版本（例如 Hadoop v0.20.2 版本）要求包含 Hadoop 归档工具接口的 JAR 包要放置在 Hive 的 auxlib 路径下：

```

$ mkdir $HIVE_HOME/auxlib
$ cp $HADOOP_HOME/hadoop-0.20.2-tools.jar $HIVE_HOME/auxlib/

```

在归档之前，我们来看下这个表下面的底层目录结构。需要注意的是，表下面的数据是存储在分区目录下的，因为这是一个管理分区表：

```

hive> dfs -ls /user/hive/warehouse/hive_text/folder=docs;
Found 2 items
/user/hive/warehouse/hive_text/folder=docs/README.txt
/user/hive/warehouse/hive_text/folder=docs/RELEASE_NOTES.txt

```

ALTER TABLE ... ARCHIVE PARTITION 语句将表转化成了一个归档表：

```

hive> SET hive.archive.enabled=true;
hive> ALTER TABLE hive_text ARCHIVE PARTITION (folder='docs');
intermediate.archived is
file:/user/hive/warehouse/hive_text/folder=docs_INTERMEDIATE_ARCHIVED
intermediate.original is
file:/user/hive/warehouse/hive_text/folder=docs_INTERMEDIATE_ORIGINAL
Creating data.har for file:/user/hive/warehouse/hive_text/folder=docs
in file:/tmp/hive-edward/hive_..._3862901820512961909/-ext-10000/partlevel
Please wait... (this may take a while)
Moving file:/tmp/hive-edward/hive_..._3862901820512961909/-ext-10000/partlevel
to file:/user/hive/warehouse/hive_text/folder=docs_INTERMEDIATE_ARCHIVED
Moving file:/user/hive/warehouse/hive_text/folder=docs
to file:/user/hive/warehouse/hive_text/folder=docs_INTERMEDIATE_ORIGINAL
Moving file:/user/hive/warehouse/hive_text/folder=docs_INTERMEDIATE_ARCHIVED
to file:/user/hive/warehouse/hive_text/folder=docs

```

（对于上面的输出，为便于展示，我们对其格式进行了调整，并使用...替换掉输出中的时间戳。）

下面这张表中由之前的两个文件变成了现在的一个 Hadoop 归档文件（也就是 HAR 文件）：

```

hive> dfs -ls /user/hive/warehouse/hive_text/folder=docs;
Found 1 items
/user/hive/warehouse/hive_text/folder=docs/data.har

```

ALTER TABLE ... UNARCHIVE PARTITION 命令可以将 HAR 中的文件提取出来然后重新放置到 HDFS 中：

```
ALTER TABLE hive_text UNARCHIVE PARTITION (folder='docs');
```

11.8 压缩：包扎

Hive 可以读和写不同类型的压缩文件，确实可以获取很大的性能提升，因为其可以节约磁存储空间以及处理开销。这种灵活性同样也有助于和其他工具进行集成，因为 Hive 无需使用 Java 编写的自定义的“适配器”就可以查询很多本地文件类型。

Hive 不可能满足用户的所有需求,有时一个第三方的库可以填补这个空白。而其他情况下,用户或其他本身是 Java 开发工程师的人员可能需要开发一些用户自定义函数(也就是 UDF,参考第 13 章的介绍)、序列化/反序列化器(也就是 SerDe,参考第 15.4 节“记录格式:SerDe”)、输入或者输出文件格式(参考第 15 章中的介绍)或者其他一些增强功能。

本章将探讨 Hive 自身的源代码,其中包括 Hive v0.8.0 版本中新增的开发工具包插件(也就是 PDK)。

12.1 修改 Log4J 属性

Hive 可以通过 \$HIVE_HOME/conf 目录下的 2 个 Log4J 配置文件来配置日志。其中 hive-log4j.properties 文件用来控制 CLI 和其他本地执行组件的日志,而 hive-exec-log4j.properties 文件用来控制 MapReduce task 内的日志,这个文件并非必须在 Hive 安装目录下存在,因为 Hive JAR 包中已经包含了默认属性值。事实上,conf 目录下的文件是带有 .template 文件扩展名的,因此默认是不会被加载的。如果想使用这 2 个配置,只需要将 .template 扩展名去掉,然后按需要进行修改就可以了:

```
$ cp conf/hive-log4j.properties.template conf/hive-log4j.properties
$ ... edit file ...
```

也可以临时地改变日志配置而无需拷贝和修改 Log4J 文件。在 Hive Shell 启动时可以通过 hiveconf 参数指定 log4.properties 文件中的任意属性。例如,下面的语句指定输出日志为 DEBUG 级别,而且输出到控制台中:

```
$ bin/hive -hiveconf hive.root.logger=DEBUG,console
12/03/27 08:46:01 WARN conf.HiveConf: hive-site.xml not found on CLASSPATH
```

12.2 连接 Java 调试器到 Hive

当开启详细输出也无法帮助找到解决问题的办法时，可以通过附加一个 Java 调试器，对 Hive 代码进行单步调试，来找到问题所在。

远程调试是 Java 提供的一个功能，其可以通过命令行对 JVM 指定参数后进行启动。Hive Shell 脚本提供了一个开关和控制台帮助信息帮助用户方便地设置这些属性（为了便于页面展示，下面输出信息中有部分信息省略掉了）：

```
$ bin/hive --help --debug
Allows to debug Hive by connecting to it via JDI API
Usage: hive --debug[:comma-separated parameters list]

Parameters:

recursive=<y|n>          Should child JVMs also be started in debug mode. Default:y
port=<port_number>      Port on which main JVM listens for debug connection.
Default...
mainSuspend=<y|n>       Should main JVM wait with execution for the debugger to
con...
childSuspend=<y|n>      Should child JVMs wait with execution for the debugger
to c...
swapSuspend              Swaps suspend options between main and child JVMs
```

12.3 从源码编译 Hive

使用 Apache 的 Hive 发行版通常是个不错的选择，不过用户可能需要当前版本中没有提供的功能，或者使用的是一个非公开的自己定制的内部分支。

因此，用户将需要对 Hive 进行源码编译。编译 Hive 的最低要求是，需要一个较新版本的 Java JDK、Subversion 和 ANT。Hive 也包含了一些默认不会进行编译的组件，例如通过 Thrift 生成的一些类。如果期望对 Hive 完成重编译，那么还需要安装好 Thrift 编译器。

下面的系列命令分别是从小 svn 中下载一份 Hive 发行版代码，然后对源码进行编译，编译后会在 hive-trunk/build/dist 目录下生成结果：

```
$ svn co http://svn.apache.org/repos/asf/hive/trunk hive-trunk
$ cd hive-trunk
$ ant package

$ ls build/dist/
bin examples LICENSE README.txt          scripts
```

12.3.1 执行 Hive 测试用例

Hive 本身有一个独特的内置的测试框架。Hive 确实有传统的 JUnit 测试用例，不过主要的测试还是通过执行以 .q 结尾的文件，然后将执行结果和之前的执行结果文件¹进行对比来进行的。在 Hive 源码目录下有很多的目录。其中，有“正面”的测试，也就是结果应该是正确的测试；还有“反面”的测试，也就是执行结果应该是失败的测试。

正面的测试是一个符合语法规则的查询，而负面的测试是一个不符合语法规则的或者尝试做一些 HiveQL 不允许做的操作：

```
$ ls -lah ql/src/test/queries/
total 76K
drwxrwxr-x. 7 edward edward 4.0K May 28 2011 .
drwxrwxr-x. 8 edward edward 4.0K May 28 2011 ..
drwxrwxr-x. 3 edward edward 20K Feb 21 20:08 clientnegative
drwxrwxr-x. 3 edward edward 36K Mar 8 09:17 clientpositive
drwxrwxr-x. 3 edward edward 4.0K May 28 2011 negative
drwxrwxr-x. 3 edward edward 4.0K Mar 12 09:25 positive
```

可以看一下 ql/src/test/queries/clientpositive/cast1.q 这个文件。用户需要知道的是，在测试过程中会首先自动创建一张名为 src 的表。表 src 具有 2 个字段，也就是 INT 类型的字段 key 和 STRING 类型的字段 value。因为 Hive 目前还不支持没有 FROM 语句的 SELECT 查询，所以对于一些函数的测试有一个技巧，那就是通过“硬编码”函数的输入进行测试，而无需真正访问表中的数据。

正如用户在下面的例子中所看到的，在 SELECT 语句中，src 表中的数据永远没有使用到：

```
hive> CREATE TABLE dest1(c1 INT, c2 DOUBLE, c3 DOUBLE,
> c4 DOUBLE, c5 INT, c6 STRING, c7 INT) STORED AS TEXTFILE;

hive> EXPLAIN
> FROM src INSERT OVERWRITE TABLE dest1
> SELECT 3 + 2, 3.0 + 2, 3 + 2.0, 3.0 + 2.0,
> 3 + CAST(2.0 AS INT) + CAST(CAST(0 AS SMALLINT) AS INT),
> CAST(1 AS BOOLEAN), CAST(TRUE AS INT) WHERE src.key = 86;

hive> FROM src INSERT OVERWRITE TABLE dest1
> SELECT 3 + 2, 3.0 + 2, 3 + 2.0, 3.0 + 2.0,
> 3 + CAST(2.0 AS INT) + CAST(CAST(0 AS SMALLINT) AS INT),
> CAST(1 AS BOOLEAN), CAST(TRUE AS INT) WHERE src.key = 86;
```

1.也就是说，它们更像是功能或可用性测试。


```
hive> SELECT dest1.* FROM dest1;
```

上面这个脚本的输出可以在 `ql/src/test/results/clientpositive/cast1.q.out` 这个文件中找到。这个输出结果文件太大了，如果这里展示完整的结果的话，就太浪费纸张了。不过，增加这个文件也没有价值。

如下命令展示的是 Hive 客户端分别执行一个正面的测试用例和一个负面的测试用例：

```
ant test -Dtestcase=TestCliDriver -Dqfile=mapreduce1.q
ant test -Dtestcase=TestNegativeCliDriver -Dqfile=script_broken_pipe1.q
```

上面两个测试仅仅是解析了查询语句，它们实际上并没有在客户端执行。现在它们已经被启用了，而推荐使用 `clientpositive` 和 `clientnegative` 来区分了。

用户同样可以在一次 `ant` 调用过程中执行多个测试脚本，这样可以节约时间（注意下面 `-Dqfile=...` 后面的字符串是一个完整的字符串，展示在多行是为了便于排版）：

```
ant test -Dtestcase=TestCliDriver -Dqfile=avro_change_schema.q,avro_
joins.q,
avro_schema_error_message.q,avro_evolved_schemas.q,avro_sanity_test.q,
avro_schema_literal.q
```

12.3.2 执行 hook

前置 hook 和后置 hook 是允许用户进行 hook 编码，然后在 Hive 中进行编译并执行自定义代码的实用工具。Hive 的测试框架就使用了 hook 来执行不会产生输出的命令，因此结果只会在测试内部出现：

```
PREHOOK: query: CREATE TABLE dest1(c1 INT, c2 DOUBLE, c3 DOUBLE,
c4 DOUBLE, c5 INT, c6 STRING, c7 INT) STORED AS TEXTFILE
PREHOOK: type: CREATETABLE
POSTHOOK: query: CREATE TABLE dest1(c1 INT, c2 DOUBLE, c3 DOUBLE,
c4 DOUBLE, c5 INT, c6 STRING, c7 INT) STORED AS TEXTFILE
```

12.4 配置 Hive 和 Eclipse

Eclipse 是一个开源的 IDE（集成开发环境）。通过如下操作可以让 Hive 代码在 Eclipse 中使用：

```
$ ant clean package eclipse-files
$ cd metastore
$ ant model-jar
$ cd ../ql
$ ant gen-test
```

一旦编译后，用户就可以将工程导入到 Eclipse 中，然后按通常使用的方式使用就可

以了。

按常规方式，首先在 Eclipse 中创建一个工作空间 (workspace)，然后使用 File (文件) → Import (导入) 命令，再选择 General (常规) → Existing Projects into Workspace (导入已经存在的工程到工作空间中)，最后选择 Hive 源码所在的目录即可。

当出现向导中列举的系列工程名称时，应该可以看到其中有一个是名为 hive-trunk 的工程，用户只需要选择这个工程然后按 Finish (完成) 就可以了。

图 12-1 展示了如何在 Eclipse 中启动 Hive 命令行 CLI 驱动器。

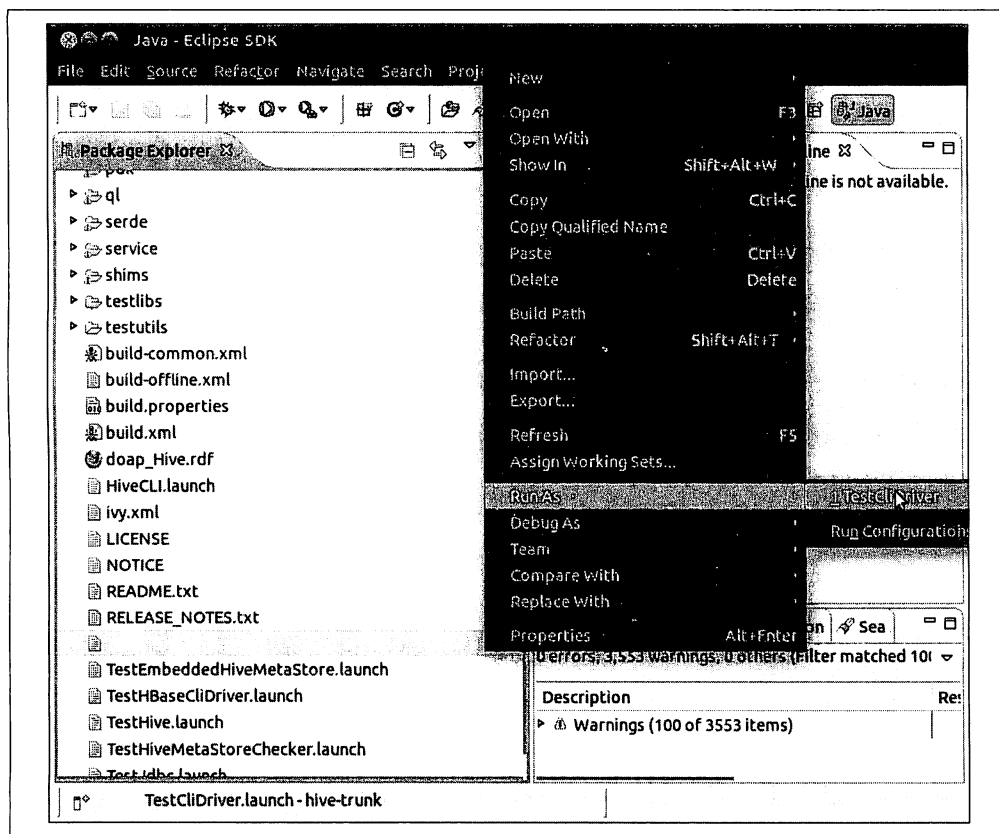


图 12-1 在 Eclipse 中启动 Hive 命令行 CLI 驱动器

12.5 Maven 工程中使用 Hive

用户可以在 Maven 编译中设置 Hive 为依赖包。Maven 资源库 <http://mvnrepository.com/artifact/org.apache.hive/hive-service> 中包含了最新的发行版。

这个页面里同样列出了 hive-service 所需要的依赖包。

下面是 Hive v0.9.0 版本的顶级依赖关系定义，这里没有包含与之相关的依赖树，因为实在太多了：

```
<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-service</artifactId>
  <version>0.9.0</version>
</dependency>
```

为我们后面将讲到的 `hive_test` 准备的 `pom.xml` 文件包含了 Hive v0.9.0 版本完整的依赖树关系。用户可以通过如下链接查看这个文件：https://github.com/edwardcapriolo/hive_test/blob/master/pom.xml。

12.6 Hive 中使用 `hive_test` 进行单元测试

还有一种使用 Hive 的方式就是利用 Thrift 通过 `HiveService` 来访问 Hive，然后写一些应用。不过，因为 Hive 需要众多的 JAR 依赖以及元数据模块，Thrift 服务通常难以在嵌入式的环境中使用。

`hive_test` 从 Maven 中获取到所有的 Hive 依赖，并且在本地建立起元数据和 Thrift 服务，然后提供测试类来让单元测试更加简单。同时，因为其是非常轻量级的，所以单元测试执行得很快。当然，这是相对于在 Hive 中通过 `test` 模块进行测试而言的，因为后者需要重编译整个工程，然后才能执行某个单元测试。

`hive_test` 是测试如 UDF、输入格式、`SerDe` 或其他作为 HiveQL 语言插件的模块代码的理想方式。不过这个无助于内部的 Hive 开发，因为所有的 Hive 组件都是从 Maven 中拖下来的，而且都是属于工程之外的。

在 Maven 工程中，创建一个 `pom.xml` 文件，然后按如下的方式将 `hive_test` 作为依赖项加入：

```
<dependency>
  <groupId>com.jointhegrid</groupId>
  <artifactId>hive_test</artifactId>
  <version>3.0.1-SNAPSHOT</version>
</dependency>
```

然后创建一个 `hive-site.xml` 文件：

```
$ cp $HIVE_HOME/conf/* src/test/resources/
$ vi src/test/resources/hive-site.xml
```

和常规的 `hive-site.xml` 文件不同的是，这个版本的配置文件不应该保存任何数据到一个永久的地方。这是因为单元测试并不应该创建或保存任何永久的状态。将

javax.jdo.option.ConnectionURL 这个属性设置为使用 Derby 作为数据库，这样就只会将数据库保存在主内存中。数据仓库目录 hive.metastore.warehouse.dir 设置为/tmp 下的某个目录，这样每次执行的单元测试都会被删除掉：

```
<configuration>

  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby:memory:metastore_db;create=true</value>
    <description>JDBC connect string for a JDBC metastore</description>
  </property>

  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>/tmp/warehouse</value>
    <description>location of default database for the warehouse</description>
  </property>

</configuration>
```

hive_test 提供了多个扩展了 JUnit 测试用例的类。HiveTestService 会先初始化好环境，清空数据仓库目录，然后会在进程中启动元数据服务和 HiveService。其通常是扩展测试的组件。不过，其他组件（例如 HiveTestEmbedded）同样是可使用的：

```
package com.jointhegrid.hive_test;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.Path;

/* Extending HiveTestService creates and initializes
the metastore and thrift service in an embedded mode */
public class ServiceHiveTest extends HiveTestService {

  public ServiceHiveTest() throws IOException {
    super();
  }

  public void testExecute() throws Exception {

    /* Use the Hadoop filesystem API to create a
data file */
    Path p = new Path(this.ROOT_DIR, "afile");
    FSDataOutputStream o = this.getFileSystem().create(p);
    BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(o));
    bw.write("1\n");
  }
}
```

```

        bw.write("2\n");
        bw.close();

        /* ServiceHive is a component that connections
        to an embedded or network HiveService based
        on the constructor used */
        ServiceHive sh = new ServiceHive();

        /* We can now interact through the HiveService
        and assert on results */
        sh.client.execute("create table atest (num int)");
        sh.client.execute("load data local inpath '"
            + p.toString() + "' into table atest");
        sh.client.execute("select count(1) as cnt from atest");
        String row = sh.client.fetchOne();
        assertEquals("2", row);
        sh.client.execute("drop table atest");
    }
}

```

12.7 新增的插件开发工具箱 (PDK)

Hive v0.8.0 版本中新加入了一个插件开发工具箱 (PDK)。PDK 的目的是允许开发者可以无需 Hive 源码，而仅仅需要二进制代码即可编译和测试插件。

PDK 相对较新，而且其本身因为有一些棘手的 bug，所以比较难用。如果不管怎样想尝试使用 PDK 的话，那么可以参考如下的 wiki 页面：<https://cwiki.apache.org/Hive/plugindeveloperkit.html>。不过要注意这个页面中也有一些错误，至少在写本书时还是存在一些错误的。

用户自定义函数 (UDF) 是一个允许用户扩展 HiveQL 的强大的功能。正如我们将看到的, 用户使用 Java 进行编码。一旦将用户自定义函数加入到用户会话中 (交互式的或者通过脚本执行的), 它们就将和内置的函数一样使用, 甚至可以提供联机帮助。Hive 具有多种类型的用户自定义函数, 每一种都会针对输入数据执行特定“一类”的转换过程。

在 ETL 处理中, 一个处理过程可能包含多个处理步骤。Hive 语言具有多种方式来将上一步骤的输入通过管道传递给下一个步骤, 然后在一个查询中产生众多输出。用户同样可以针对一些特定的处理过程编写自定义函数。如果没有这个功能, 那么一个处理过程可能就需要包含一个 MapReduce 步骤或者需要将数据转移到另一个系统中来实现这些改变。互联系统增加了复杂性, 并且增加了配置错误或其他错误的发生几率。当数据量是 GB 甚至 TB 级别时, 在不同系统中转移数据, 需要消耗大量的时间。与此相反, UDF 是在 Hive 查询产生的相同的 task 进程中执行的, 因此它们可以高效地执行, 而且其消除了和其他系统集成时所产生的复杂度。本章涵括创建和使用 UDF 的最佳实践。

13.1 发现和描述函数

在编写自定义 UDF 之前, 我们先来熟悉下 Hive 中自带的那些 UDF。需要注意的是在 Hive 中通常使用“UDF”来表示任意的函数, 包括用户自定义的或者内置的。

SHOW FUNCTIONS 命令可以列举出当前 Hive 会话中所加载的所有函数名称, 其中包括内置的和用户加载进来的函数, 加载方式稍后会进行介绍:

```
hive> SHOW FUNCTIONS;  
abs  
acos  
and  
array
```

```
array_contains
...
```

函数通常都有其自身的使用文档。使用 `DESCRIBE FUNCTION` 命令可以展示对应函数简短的介绍：

```
hive> DESCRIBE FUNCTION concat;
concat(str1, str2, ... strN) - returns the concatenation of str1, str2, ...
strN
```

函数也可能包含更多的详细文档，可以通过增加 `EXTENDED` 关键字进行查看：

```
hive> DESCRIBE FUNCTION EXTENDED concat;
concat(str1, str2, ... strN) - returns the concatenation of str1, str2, ...
strN
Returns NULL if any argument is NULL.
Example:
> SELECT concat('abc', 'def') FROM src LIMIT 1;
'abcdef'
```

13.2 调用函数

如果想使用函数，只需要在查询中通过调用函数名，并传入需要的参数即可。某些函数需要指定特定的参数个数和参数类型，而其他函数可以传入一组参数，参数类型可以是多样的。和关键字一样，函数名也是保留的字符串：

```
SELECT concat(column1,column2) AS x FROM table;
```

13.3 标准函数

用户自定义函数（英文缩写为 UDF）这个术语在狭义的概念上还表示以一行数据中的一列或多列数据作为参数然后返回结果是一个值的函数。大多数函数都是属于这类的。我们使用的例子中包含了很多的数学函数。例如 `round()`和 `floor()`，其可以将 `DOUBLE` 类型转换为 `BIGINT` 类型；还有 `abs()`，这个函数可以返回数值的绝对值。

其他一些例子中还包含有字符串操作函数。例如 `ucase()`，这个函数可以将字符串转换成全是大写字母的；`reverse()`函数，可以将字符串进行反转；而 `concat()`函数可以将输入的多个字符串拼接成一个字符串输出。

需要注意的是，这些 UDF 同样可以返回一个复杂的对象，例如 `array`、`map` 或者 `struct`。

13.4 聚合函数

另一种函数是聚合函数。所有的聚合函数、用户自定义函数和内置函数，都统称为用

户自定义聚合函数 (UDAF)。

聚合函数接受从零行到多行的零个到多个列，然后返回单一值。这样的例子包括数学函数 `sum()`，其返回所有输入求和后的值；`avg()`函数会计算所有输入的值的平均值；`min()`和 `max()`函数，可以分别返回输入值中的最小值和最大值：

```
hive> SELECT avg(price_close)
> FROM stocks
> WHERE exchange = 'NASDAQ' AND symbol = 'AAPL';
```

聚合方法通常和 `GROUP BY` 语句组合使用。下面这个例子我们在第 6.3 节“`GROUP BY` 语句”中使用过：

```
hive> SELECT year(ymd), avg(price_close) FROM stocks
> WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
> GROUP BY year(ymd);
1984      25.578625440597534
1985      20.193676221040867
1986      32.46102808021274
...
```

第 6 章中的表 6-3 中列举了可以在 HiveQL 中使用的所有内置聚合函数。

13.5 表生成函数

Hive 所支持的第 3 类函数就是表生成函数。和其他函数类别一样，所有的表生成函数，包括用户自定义的和内置的，都统称为用户自定义表生成函数 (UDTF)。

表生成函数接受零个或多个输入，然后产生多列或多行输出。例如，`array` 函数就是将一列输入转换成一个数组输出的。下面的查询语句中使用了 `array` 函数：

```
hive> SELECT array(1,2,3) FROM dual;
[1,2,3]
```

`explode()`函数以 `array` 类型数据作为输入，然后对数组中的数据进行迭代，返回多行结果，一行一个数组元素值。

```
hive> SELECT explode(array(1,2,3)) AS element FROM src;
1
2
3
```

不过，Hive 只允许表生成函数以特定的方式使用。例如，一个显著的限制就是，我们无法从表中产生其他的列。例 13-1 所示的这个查询可能是前面我们对于 `employees` 表进行的操作。我们可能需要列举出每名雇员的下属员工：

例 13-1 explode 函数的错误使用方式:

```
hive> SELECT name, explode(subordinates) FROM employees;
FAILED: Error in semantic analysis: UDTF's are not supported outside
the SELECT clause, nor nested in expressions
```

不过, Hive 提供了一个 LATERAL VIEW 功能来实现这种查询:

```
hive> SELECT name, sub
> FROM employees
> LATERAL VIEW explode(subordinates) subView AS sub;
John Doe      Mary Smith
John Doe      Todd Jones
Mary Smith    Bill King
```

需要注意的是, 对于职位不是经理的雇员(即没有下属员工的人)来说是没有输出行的, 例如 Bill King 和 Todd Jones 就不会有对应的输出行。因此, 输出中将会产生零到多行新纪录。

通过 LATERAL VIEW 可以方便地将 explode 这个 UDTF 得到的行转列的结果集合在一起提供服务。使用 LATERAL VIEW 需要指定视图别名和生成的新列的别名, 对于本例, 其分别是 subView 和 sub。

第 6 章中的表 6-4 中列举了所有的内置的表生成行数。

13.6 一个通过日期计算其星座的 UDF

下面我们开始编写自己的 UDF。假设我们有一张表, 表中的一个字段存储的是每个用户的生日。通过这个信息, 我们期望能够计算出每个人所属的星座。

下面是一个样本数据集, 我们将其放到用户根目录下一个名为 littlebigdata.txt 的文件中:

```
edward capriolo,edward@media6degrees.com,2-12-1981,209.191.139.200,M,10
bob,bob@test.net,10-10-2004,10.10.10.1,M,50
sara connor,sara@sky.net,4-5-1974,64.64.5.1,F,2
```

将样本数据载入到名为 littlebigdata 的表中:

```
hive > CREATE TABLE IF NOT EXISTS littlebigdata(
> name      STRING,
> email     STRING,
> bday      STRING,
> ip        STRING,
> gender    STRING,
> anum     INT)
> ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

hive> LOAD DATA LOCAL INPATH '${env:HOME}/littlebigdata.txt'
```

```
> INTO TABLE littlebigdata;
```

函数的输入将是一个日期，而函数输出将是表示该用户星座的字符串。

下面是我们期望的这个 UDF 的 Java 实现：

```
package org.apache.hadoop.hive.contrib.udf.example;

import java.util.Date;
import java.text.SimpleDateFormat;
import org.apache.hadoop.hive.ql.exec.UDF;

@Description(name = "zodiac",
    value = "_FUNC_(date) - from the input date string "+
        "or separate month and day arguments, returns the sign of the Zodiac.",
    extended = "Example:\n"
        + " > SELECT _FUNC_(date_string) FROM src;\n"
        + " > SELECT _FUNC_(month, day) FROM src;")

public class UDFZodiacSign extends UDF{

    private SimpleDateFormat df;

    public UDFZodiacSign(){
        df = new SimpleDateFormat("MM-dd-yyyy");
    }

    public String evaluate( Date bday ){
        return this.evaluate( bday.getMonth(), bday.getDay() );
    }

    public String evaluate(String bday){
        Date date = null;
        try {
            date = df.parse(bday);
        } catch (Exception ex) {
            return null;
        }
        return this.evaluate( date.getMonth()+1, date.getDay() );
    }

    public String evaluate( Integer month, Integer day ){
        if (month==1) {
            if (day < 20 ){
                return "Capricorn";
            } else {
                return "Aquarius";
            }
        }
    }
}
```

```

    if (month==2){
        if (day < 19 ){
            return "Aquarius";
        } else {
            return "Pisces";
        }
    }
    /* ...other months here */
    return null;
}
}

```

(译者注: 这段代码有 2 个很明显的错误, 其一, bday.getMonth 的返回值范围是 0~11, 其中值 0 表示一月份, 因此这里面正确的写法应该是 bday.getMonth+1, 其二 bday.getDay()返回的是一周中的第几天, 显然不对, 这里应该使用 bday.getDate()返回月中的天数。)

编写一个 UDF, 需要继承 UDF 类并实现 evaluate()函数。在查询执行过程中, 查询中对应的每个应用到这个函数的地方都会对这个类进行实例化。对于每行输入都会调用到 evaluate()函数。而 evaluate()处理后的值会返回给 Hive。同时用户是可以重载 evaluate 方法的。Hive 会像 Java 的方法重载一样, 自动选择匹配的方法。

代码中的 @Description(...)表示的是 Java 总的注解, 是可选的。注解中注明了关于这个函数的文档说明, 用户需要通过这个注解来阐明自定义的 UDF 的使用方法和例子。这样当用户通过 DESCRIBE FUNCTION ...命令查看该函数时, 注解中的 _FUNC_ 字符串将会被替换为用户为这个函数定义的“临时”函数名称, 定义方式下面会进行介绍。



提示

UDF 中 evaluate()函数的参数和返回值类型只能是 Hive 可以序列化的数据类型。例如, 如果用户处理的全是数值, 那么 UDF 的输出参数类型可以是基本数据类型 int、Integer 封装的对象或者是一个 IntWritable 对象, 也就是 Hadoop 对整型封装后的对象。用户不需要特别地关心将调用到哪个类型, 因为当类型不一致的时候, Hive 会自动将类型转换成匹配的类型。需要记住的是, null 在 Hive 中对于任何数据类型都是合法的, 但是对于 Java 基本数据类型, 不能是对象, 也不能是 null。

如果想在 Hive 中使用 UDF, 那么需要将 Java 代码进行编译, 然后将编译后的 UDF 二进制类文件打包成一个 JAR 文件。然后, 在 Hive 会话中, 将这个 JAR 文件加入到类路径下, 再通过 CREATE FUNCTION 语句定义好使用这个 Java 类的函数:

```
hive> ADD JAR /full/path/to/zodiac.jar;
```

```
hive> CREATE TEMPORARY FUNCTION zodiac
> AS 'org.apache.hadoop.hive.contrib.udf.example.UDFZodiacSign';
```

需要注意的是，JAR 文件路径是不需要用引号括起来的，同时，到目前为止这个路径需要是当前文件系统的全路径。Hive 不仅仅将这个 JAR 文件加入到 classpath 下，同时还将其加入到了分布式缓存中，这样整个集群的机器都是可以获得该 JAR 文件的。

现在这个判断星座的 UDF 可以像其他的函数一样使用了。需要注意下 CREATE FUNCTION 语句中的 TEMPORARY 这个关键字。当前会话中声明的函数只会在当前会话中有效。因此用户需要在每个会话中都增加 JAR 然后创建函数。不过，如果用户需要频繁地使用同一个 JAR 文件和函数的话，那么可以将相关语句增加到 \$HOME/.hiverc 文件中去：

```
hive> DESCRIBE FUNCTION zodiac;
zodiac(date) - from the input date string or separate month and day
arguments, returns the sign of the Zodiac.
```

```
hive> DESCRIBE FUNCTION EXTENDED zodiac;
zodiac(date) - from the input date string or separate month and day
arguments, returns the sign of the Zodiac.
```

Example:

```
> SELECT zodiac(date_string) FROM src;
> SELECT zodiac(month, day) FROM src;
```

```
hive> SELECT name, bday, zodiac(bday) FROM littlebigdata;
edward capriolo 2-12-1981 Aquarius
bob             10-10-2004  Libra
sara connor    4-5-1974   Aries
```

再次说明下，UDF 允许用户在 Hive 语言中执行自定义的转换过程。通过上面那个 UDF，Hive 现在可以通过用户生日计算得到相应的星座名称了，当然也可以做其他的聚合和转换过程。

当我们使用完自定义 UDF 后，我们可以通过如下命令删除此函数：

```
hive> DROP TEMPORARY FUNCTION IF EXISTS zodiac;
```

像通常一样，IF EXISTS 是可选的。如果增加此关键字，则即使函数不存在也不会报错。

13.7 UDF 与 GenericUDF

前面介绍的那个计算星座的 UDF 例子中，我们继承的是 UDF 类。Hive 还提供了一个对应的称为 GenericUDF 的类。GenericUDF 是更为复杂的抽象概念，但是其支持更好的 null 值处理同时可以处理一些标准的 UDF 无法支持的编程操作。GenericUDF 的一个例子就是 Hive 中的 CASE ... WHEN 语句，其会根据语句中输入的参数而产生复杂的

处理逻辑。下面我们将展示如何通过继承 `GenericUDF` 类来编写一个用户自定义函数，我们称之为 `nvl()`，这个函数传入的值如果是 `null`，那么就会返回一个默认值。

函数 `nvl()` 要求有 2 个参数。如果第 1 个参数是非 `null` 值，那么就会返回这个值；如果第 1 个参数是 `null`，那么将返回第 2 参数的值。`GenericUDF` 框架正适合处理这类问题。通过继承标准的 UDF 确实是一个解决方案，不过那样就需要针对如此多的输入类型重载众多的 `evaluate` 方法，这样显得非常麻烦。而 `GenericUDF` 将会以编程的方式检查输入的数据类型，然后做出合适的反馈。

我们以普通 `import` 语句的细目列表开始这个代码：

```
package org.apache.hadoop.hive.ql.udf.generic;

import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentLengthException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentTypeException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDFUtils;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
```

之后，我们使用 `@Description` 注解来为这个 UDF 写明使用文档：

```
@Description(name = "nvl",
value = "_FUNC_(value,default_value) - Returns default value if value"
    + " is null else returns value",
extended = "Example:\n"
    + " > SELECT _FUNC_(null,'bla') FROM src LIMIT 1;\n")
```

现在需要这个类继承 `GenericUDF`，然后开发这个类通常需要实现的方法。

其中 `initialize()` 方法会被输入的几个参数调用，并最终传入到一个 `ObjectInspector` 对象中。这个方法的目标是确定参数的返回类型。如果传入方法的类型是不合法的，这时用户同样可以向控制台抛出一个 `Exception` 异常信息。`returnOIRResolver` 是一个内置的类，其通过获取非 `null` 值的变量的类型并使用这个数据类型来确定返回值类型：

```
public class GenericUDFNvl extends GenericUDF {
    private GenericUDFUtils.ReturnObjectInspectorResolver returnOIRResolver;
    private ObjectInspector[] argumentOIs;

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments)
        throws UDFArgumentException {
        argumentOIs = arguments;
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException(
```

```

        "The operator 'NVL' accepts 2 arguments.");
    }
    returnOIRResolver = new GenericUDFUtils.ReturnObjectInspector
Resolver(true);
    if (!(returnOIRResolver.update(arguments[0]) && returnOIRResolver
        .update(arguments[1]))) {
        throw new UDFArgumentTypeException(2,
            "The 1st and 2nd args of function NLV should have the same type, "
            + "but they are different: \"" + arguments[0].getTypeName()
            + "\" and \"" + arguments[1].getTypeName() + "\"");
    }
    return returnOIRResolver.get();
}
...

```

方法 `evaluate` 的输入是一个 `DeferredObject` 对象数组，而 `initialize` 方法中创建的 `returnOIRResolver` 对象就用于从 `DeferredObjects` 对象中获取到值。在这种情况下，这个函数将会返回第 1 个非 null 值：

```

...
@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {
    Object retVal = returnOIRResolver.convertIfNecessary(arguments[0].get(),
        argumentOIs[0]);
    if (retVal == null) {
        retVal = returnOIRResolver.convertIfNecessary(arguments[1].get(),
            argumentOIs[1]);
    }
    return retVal;
}
...

```

最后一个要实现的方法就是 `getDisplayString()`，其用于 Hadoop task 内部，在使用到这个函数时来展示调试信息：

```

...
@Override
public String getDisplayString(String[] children) {
    StringBuilder sb = new StringBuilder();
    sb.append("if ");
    sb.append(children[0]);
    sb.append(" is null ");
    sb.append("returns");
    sb.append(children[1]);
    return sb.toString();
}
}

```

为了展示这个 UDF 的通用处理特性，下面的查询中对其调用了多次，每次都传入不同

类型的参数，正如下面例子所展示的：

```
hive> ADD JAR /path/to/jar.jar;

hive> CREATE TEMPORARY FUNCTION nvl
  > AS 'org.apache.hadoop.hive.ql.udf.generic.GenericUDFNvl';

hive> SELECT nvl( 1 , 2 ) AS COL1,
  >          nvl( NULL, 5 ) AS COL2,
  >          nvl( NULL, "STUFF" ) AS COL3
  > FROM src LIMIT 1;
1          5          STUFF
```

13.8 不变函数

到目前为止我们都是将代码打包成 JAR 文件，然后再使用 ADD JAR 和 CREATE TEMPORARY FUNCTION 命令来使用这些函数的。

用户同样可以将自己的函数永久地加入到 Hive 中，不过这就需要对 Hive 的 Java 文件进行简单的修改，然后重新编译 Hive。

对于 Hive 源代码，需要对 ql/src/java/org/apache/hadoop/hive/ql/exec/FunctionRegistry.java 这个 FunctionRegistry 类进行一行的代码修改。然后按照 Hive 源码分支的编译方式，对 Hive 源码重新编译即可。

尽管建议是重新部署整个新的编译后的版本，不过实际上只需要替换 hive-exec-*.jar 这个 JAR 文件即可，其中 * 表示的是版本号，需要替换成具体的值。

下面就是一个将 nvl() 函数增加到 Hive 内置函数列表中时对于 FunctionRegistry 类的修改方式：

```
...
registerUDF("parse_url", UDFParseUrl.class, false);
registerGenericUDF("nvl", GenericUDFNvl.class);
registerGenericUDF("split", GenericUDFSplit.class);
...
```

13.9 用户自定义聚合函数

用户同样可以定义聚合函数，不过，接口实现起来比较复杂。聚合函数会分多个阶段进行处理。基于 UDAF 执行的转换的不同，在不同阶段的返回值类型也可能是不同的。例如，sum() 这个 UDAF 可以接受基本数据类型中的整型输入，然后创建整型部分数据，最终产生一个整型结果；而 median() 这个聚合函数会接受整型输入，然后中间会产生一组整型部分数据，最后产生一个整型结果。

作为通用的用户自定义聚合函数的例子，可以查看 <http://svn.apache.org/repos/asf/hive/branches/branch-0.8/ql/src/java/org/apache/hadoop/hive/ql/udf/generic/GenericUDAFAverage.java> 这个 GenericUDAFAverage 类的源代码。



提示

聚合过程是在 map 或者 reduce 任务 (task) 中执行的，其是一个有内存限制的 Java 进程。因此，在聚合过程中存储大的结构化数据可能会产生内存溢出错误。对于 min() 这个 UDAF 而言，只需要在内存中保存单个元素来进行比较即可。而 collectset() 这个 UDAF 内部是使用 set 来排重存储数据，以减少内存使用量的。percentile_approx() 使用近似算法来获取近似结果以限制内存使用。在写 UDAF 的时候一定要注意内存使用的问题。通过配置参数 mapred.child.java.opts 可以调整执行过程的内存需求量，但是这种方式并非总是奏效。

```
<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx200m</value>
</property>
```

创建一个 COLLECT UDAF 来模拟 GROUP_CONCAT

MySQL 中有一个非常有用的函数名为 GROUP_CONCAT，其可以将一组中的所有元素按照用户指定的分隔符组装成一个字符串。下面这个例子展示了 MySQL 中是如何使用这个函数的：

```
mysql > CREATE TABLE people (
  name STRING,
  friendname STRING );

mysql > SELECT * FROM people;
bob      sara
bob      john
bob      ted
john     sara
ted      bob
ted      sara

mysql > SELECT name, GROUP_CONCAT(friendname SEPARATOR ',')
  FROM people
  GROUP BY name;
bob      sara,john,ted
john     sara
ted      bob,sara
```

我们无需增加新的语法就可以在 Hive 中实现同样的转换。首先，我们需要一个聚合

函数将所有的输入作为一个列表加入到集合中。Hive 中已经有了一个叫做 `collect_set` 的 UDAF 来将所有的输入加入到一个 `java.util.Set` 集合中。Set 类型的集合会在插入输入的时候自动进行排重，这对于 `GROUP CONCAT` 来说是不合适的。为了组装合适的集合，我们使用 `collect_set` 中的代码来将 Set 的实例替换成 `ArrayList` 实例。这样就不会对输入进行排重。这个聚合过程的结果就是产生一个包含所有值的数组。

我们需要记住的很重要的一点就是，用户的聚合计算应该是允许数据任意划分为多个部分进行计算而不会影响结果的。可以想象下写一个分而治之的算法，其中划分的数据完全不由用户控制而是由 Hive 进行控制的。比较正式的说明是，输入的行应该可以分成 2 个或多个子集，并可以分别对每个子集进行计算，同时允许将并行执行的各个子集的结果合并到其他并行执行的结果中，最终得到整个集合的结果。

下面的代码在 Github 中是可以查看到的。聚合过程的所有输入必须是基本数据类型。不像 `GenericUDF` 返回的是 `ObjectInspector` 对象，聚合过程返回的是 `GenericUDAFEvaluator` 的子类对象：

```
@Description(name = "collect", value = "_FUNC_(x) - Returns a list of objects. "+
"CAUTION will easily OOM on large data sets" )
public class GenericUDAFCollect extends AbstractGenericUDAFResolver {
    static final Log LOG = LogFactory.getLog(GenericUDAFCollect.class.
getName());

    public GenericUDAFCollect() {
    }

    @Override
    public GenericUDAFEvaluator getEvaluator(TypeInfo[] parameters)
        throws SemanticException {
        if (parameters.length != 1) {
            throw new UDFArgumentTypeException(parameters.length - 1,
                "Exactly one argument is expected.");
        }
        if (parameters[0].getCategory() != ObjectInspector.Category.
PRIMITIVE) {
            throw new UDFArgumentTypeException(0,
                "Only primitive type arguments are accepted but "
                + parameters[0].getTypeName() + " was passed as parameter 1.");
        }
        return new GenericUDAFMkListEvaluator();
    }
}
```

表 13-1 对基类中提供的部分方法进行了描述。

表 13-1 基类中提供的方法描述

方法名	描述
init	Hive 会调用此方法来初始实例化一个 UDAF evaluator 类
getNewAggregationBuffer	返回一个用于存储中间聚合结果的对象
iterate	将一行新的数据载入到聚合 buffer 中
terminatePartial	以一种可持久化的方法返回当前聚合的内容。这里所说的可持久化是指返回值只能使用 Java 基本数据类型和 array, 以及基本封装类型 (例如 Double), Hadoop 中 Writable 类、list 和 map 类型。不能使用用户自定义的类 (即使实现了 java.io.Serializable)
merge	将 terminatePartial 返回的中间部分聚合结果合并到当前聚合中
terminate	返回最终聚合结果给 Hive

在 init 方法中, 在判断评估器所处的模式之后, 可以设置返回结果类型的对象检查器。iterate()方法和 terminatePartial()方法会在 map 端使用到; 而 terminate()方法和 merge()方法会在 reduce 端使用到, 用于生成最终结果。在所有情况下, 合并过程都会产生大的列表:

```

public static class GenericUDAFMkListEvaluator extends GenericUDAFEvaluator {
    private PrimitiveObjectInspector inputOI;
    private StandardListObjectInspector loi;
    private StandardListObjectInspector internalMergeOI;

    @Override
    public ObjectInspector init(Mode m, ObjectInspector[] parameters)
        throws HiveException {
        super.init(m, parameters);
        if (m == Mode.PARTIAL1) {
            inputOI = (PrimitiveObjectInspector) parameters[0];
            return ObjectInspectorFactory
                .getStandardListObjectInspector(
                    (PrimitiveObjectInspector) ObjectInspectorUtils
                        .getStandardObjectInspector(inputOI));
        } else {
            if (!(parameters[0] instanceof StandardListObjectInspector)) {
                inputOI = (PrimitiveObjectInspector) ObjectInspectorUtils
                    .getStandardObjectInspector(parameters[0]);
                return (StandardListObjectInspector) ObjectInspectorFactory
                    .getStandardListObjectInspector(inputOI);
            } else {
                internalMergeOI = (StandardListObjectInspector) parameters[0];
                inputOI = (PrimitiveObjectInspector)
                    internalMergeOI.getListElementObjectInspector();
            }
        }
    }
}

```

```

        loi = (StandardListObjectInspector) ObjectInspectorUtils
            .getStandardObjectInspector(internalMergeOI);
        return loi;
    }
}
...

```

余下的方法和类定义会定义 `MkArrayAggregationBuffer` 以及修改这个 `buffer` 的顶级方法。



提示

用户可能已经注意到 Hive 尝试尽可能地避免通过 `new` 创建对象。Hadoop 和 Hive 依据这个规则创建尽可能少的临时对象，这样可以尽量减轻 JVM 的垃圾回收过程。在写 UDF 的时候，需要牢记这个原则，因为通常是可以引用重用对象的，而使用不变类型对象可能会导致出现 bug!

```

...
static class MkArrayAggregationBuffer implements AggregationBuffer {
    List<Object> container;
}

@Override
public void reset(AggregationBuffer agg) throws HiveException {
    ((MkArrayAggregationBuffer) agg).container =
        new ArrayList<Object>();
}

@Override
public AggregationBuffer getNewAggregationBuffer()
    throws HiveException {
    MkArrayAggregationBuffer ret = new MkArrayAggregationBuffer();
    reset(ret);
    return ret;
}

// Mapside
@Override
public void iterate(AggregationBuffer agg, Object[] parameters)
    throws HiveException {
    assert (parameters.length == 1);
    Object p = parameters[0];

    if (p != null) {
        MkArrayAggregationBuffer myagg = (MkArrayAggregationBuffer) agg;
        putIntoList(p, myagg);
    }
}
}

```

```

// Mapside
@Override
public Object terminatePartial(AggregationBuffer agg)
    throws HiveException {
    MkArrayAggregationBuffer myagg = (MkArrayAggregationBuffer) agg;
    ArrayList<Object> ret = new ArrayList<Object>(myagg.container.size());
    ret.addAll(myagg.container);
    return ret;
}

@Override
public void merge(AggregationBuffer agg, Object partial)
    throws HiveException {
    MkArrayAggregationBuffer myagg = (MkArrayAggregationBuffer) agg;
    ArrayList<Object> partialResult =
        (ArrayList<Object>) internalMergeOI.getList(partial);
    for(Object i : partialResult) {
        putIntoList(i, myagg);
    }
}

@Override
public Object terminate(AggregationBuffer agg) throws HiveException {
    MkArrayAggregationBuffer myagg = (MkArrayAggregationBuffer) agg;
    ArrayList<Object> ret = new ArrayList<Object>(myagg.container.size());
    ret.addAll(myagg.container);
    return ret;
}

private void putIntoList(Object p, MkArrayAggregationBuffer myagg) {
    Object pCopy =
        ObjectInspectorUtils.copyToStandardObject(p, this.inputOI);
    myagg.container.add(pCopy);
}
}

```

使用 collect 函数可以返回所有聚合后的值的数组列表:

```

hive> dfs -cat $HOME/afile.txt;
twelve 12
twelve 1
eleven 11
eleven 10

hive> CREATE TABLE collecttest (str STRING, countVal INT)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY '09' LINES TERMINATED BY '10';

hive> LOAD DATA LOCAL INPATH '${env:HOME}/afile.txt' INTO TABLE collecttest;

hive> SELECT collect(str) FROM collecttest;

```

```
[twelve,twelve,eleven,eleven]
```

函数 `concat_ws()` 的第 1 个参数是个分隔符，其他的参数可以是字符串或者字符串数组。返回值是按照指定分隔符将所有字符串拼接在一起后的字符串。例如，下面这个例子中我们使用逗号将一组字符串拼接成一个字符串：

```
hive> SELECT concat_ws( ',', collect(str)) FROM collecttest;
twelve,twelve,eleven,eleven
```

`GROUP_CONCAT` 函数可以按照如下语句通过组合使用 `GROUP BY`、`COLLECT` 和 `concat_ws()` 达到同样的效果：

```
hive> SELECT str, concat_ws( ',', collect(cast(countVal AS STRING)))
> FROM collecttest GROUP BY str;
eleven 11,10
twelve 12,1
```

13.10 用户自定义表生成函数

尽管 UDF 可用来返回 `array`(数组)和 `structure`(结构体)，但是它们无法返回多列或多行。用户自定义表生成函数(或简称 UDTF)通过提供一个可以返回多列甚至多行的程序接口来满足这个需求的。

13.10.1 可以产生多行数据的 UDTF

我们已经在好几个例子中使用了 `explode` 方法。`explode` 方法的输入是一个数组，而将数组中每个元素都作为一行进行输出。达到相同效果的另一种可选方式就是使用 UDTF，基于某个输入产生多行输出。这里我们将展示一个 UDTF，其效果类似于 `for` 循环。这个函数接受的是用户输入的起始数值和终止数值，然后输出 N 行数据：

```
hive> SELECT forx(1,5) AS i FROM collecttest;
1
2
3
4
5
```

这个类继承的是 `GenericUDTF` 接口。开始我们就声明了 3 个整型变量，也就是变量 `start`、变量 `end` 和变量 `inc`(代表递增量)。数组对象 `forwardObj` 用于存放要返回的结果行：

```
package com.jointhegrid.udf.collect;

import java.util.ArrayList;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDFUtils.*;
```

```

import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.*;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.*;
import org.apache.hadoop.io.IntWritable;

public class GenericUDTFFor extends GenericUDTF {

    IntWritable start;
    IntWritable end;
    IntWritable inc;

    Object[] forwardObj = null;
    ...

```

因为本函数的输入参数都是常数，所以在初始化的 `initialize` 方法中就可以确定各个变量的值了。对于本函数，非常量数据是无法到达 `evaluate` 方法进行处理。第 3 个参数，也就是递增量是可选的，默认值是 1：

```

...
@Override
public StructObjectInspector initialize(ObjectInspector[] args)
    throws UDFArgumentException {
    start=((WritableConstantIntObjectInspector) args[0])
        .getWritableConstantValue();
    end=((WritableConstantIntObjectInspector) args[1])
        .getWritableConstantValue();
    if (args.length == 3) {
        inc =((WritableConstantIntObjectInspector) args[2])
            .getWritableConstantValue();
    } else {
        inc = new IntWritable(1);
    }
    ...

```

本函数只会返回一行数据，而且这行数据的数据类型确定是整型的。我们需要提供一个列名，不过用户通常可以在后面重新命名列名：

```

...
this.forwardObj = new Object[1];
ArrayList<String> fieldNames = new ArrayList<String>();
ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();

fieldNames.add("col0");
fieldOIs.add(
    PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(
        PrimitiveCategory.INT));

return ObjectInspectorFactory.getStandardStructObjectInspector(
    fieldNames, fieldOIs);

```

```
}  
...
```

`process` 方法是实际进行处理的过程。需要注意的是，这个方法的返回类型是 `void`。这是因为 UDTF 可以向前获取零行或多行数据，而不像 UDF，其只有唯一返回值。这种情况下会在 `for` 循环中对 `forward` 方法进行多次调用，这样每迭代一次就可以获取一行数据：

```
...  
@Override  
public void process(Object[] args)  
    throws HiveException, UDFArgumentException {  
    for (int i = start.get(); i < end.get(); i = i + inc.get()) {  
        this.forwardObj[0] = new Integer(i);  
        forward(forwardObj);  
    }  
}  
  
@Override  
public void close() throws HiveException {  
}  
}
```

13.10.2 可以产生具有多个字段的单行数据的 UDTF

返回一行但是包含多列数据的 UDTF 的一个例子就是 `parse_url_tuple` 这个函数。这是个内置 Hive 函数，其有一个输入参数是一个 URL 链接，它还可以指定其他多个常数，来获取用户期望返回的特定部分：

```
hive> SELECT parse_url_tuple(weblogs.url, 'HOST', 'PATH')  
       > AS (host, path) FROM weblogs;  
google.com      /index.html  
hotmail.com     /a/links.html
```

这种类型的 UDTF 的好处是 URL 只需要被解析一次，然后就可以返回多个列。这显然是个性能优势。而替代方式是：如果使用 UDF 的话，那么就需要写多个 UDF，分别抽取出其 URL 的特定部分。使用 UDF 需要写更多的代码，同时因为 URL 需要多次进行解析，所以需要消耗更长的时间。可能需要类似于如下的使用方式：

```
SELECT PARSE_HOST(a.url) as host, PARSE_PORT(url) FROM weblogs;
```

13.10.3 可以模拟复杂数据类型的 UDTF

UDTF 可作为一种向 Hive 中增加更多复杂数据类型的技术。例如，某个复杂数据类型可以序列化成一个编码字符串，而 UDTF 可以在需要的时候对这个复杂数据类型进行反序列化。假设我们有一个名为 `Book` 的 Java 类。Hive 无法直接处理这样的数据类型，不过 `Book` 对象可以编码成字符串格式，并从字符串格式解码出来：

```
public class Book {
```

```

public Book () { }
public String isbn;
public String title;
public String [] authors;

/* note: this system will not work if your table is
   using '|' or ',' as the field delimiter! */
public void fromString(String parts){
    String [] part = part.split("\\|");
    isbn = Integer.parseInt( part[0] );
    title = part[1] ;
    authors = part[2].split(",");
}

public String toString(){
    return isbn+"\t"+title+"\t"+StringUtils.join(authors, ",");
}
}

```

假设 Book 对象的字符串格式是如下这种形式的，同时假设目前为止还无法使用 SerDe 划分来按照分隔符 “|” 和 “,” 进行分割：

```

hive> SELECT * FROM books;
5555555|Programming Hive|Edward,Dean,Jason

```

对于原始数据格式，可能是按照如下方式进行分割得到的：

```

hive> SELECT cast(split(book_info,"\\|")[0] AS INTEGER) AS isbn FROM books
> WHERE split(book_info,"\\|")[1] = "Programming Hive";
5555555

```

这个 HiveQL 语句可以正确地执行，不过对于最终用户来说有更加简单的处理方式。例如，写这种类型的查询可能需要参考相应的文档来确定使用哪个字段以及使用哪种数据类型，还要牢记类型转换的规则，以及其他一些事项。相比之下，使用 UDTF 可以使 HiveQL 更加简洁和方便阅读。下面这个例子，使用到了 parse_book() 这个 UDTF：

```

hive> FROM (
> parse_book(book_info) AS (isbn, title, authors) FROM Book ) a
> SELECT a.isbn
> WHERE a.title="Programming Hive"
> AND array_contains (authors, 'Edward');
5555555

```

函数 parse_book() 允许 Hive 返回不同数据类型的多个列，分别用来表示 book 的各个字段：

```

package com.jointhegrid.udf.collect;

import java.util.ArrayList;

```



```

import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspector
    .PrimitiveCategory;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive
    .PrimitiveObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.primitive
    .WritableConstantStringObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive
    .WritableStringObjectInspector;
import org.apache.hadoop.io.Text;

```

```

public class UDTFBook extends GenericUDTF{

    private Text sent;
    Object[] forwardObj = null;
    ...

```

这个函数将会返回包含有 3 个属性信息的数组，其中，ISBN 是个整数，title（书名）是个字符串，authors（作者）也是字符串。需要注意的是，所有的 UDF 都是可以返回嵌套数据类型的，例如，我们可以返回包含字符串数组的数组：

```

...
@Override
public StructObjectInspector initialize(ObjectInspector[] args)
    throws UDFArgumentException {

    ArrayList<String> fieldNames = new ArrayList<String>();
    ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();

    fieldNames.add("isbn");
    fieldOIs.add(PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(
        PrimitiveCategory.INT));

    fieldNames.add("title");
    fieldOIs.add(PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(
        PrimitiveCategory.STRING));

    fieldNames.add("authors");
    fieldOIs.add( ObjectInspectorFactory.getStandardListObjectInspector(
        PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(
            PrimitiveCategory.STRING)

```

```

    )
);

forwardObj= new Object[3];
return ObjectInspectorFactory.getStandardStructObjectInspector(
    fieldNames, fieldOIs);
}
...

```

process 方法只会返回一行。不过，对象数组中的每个元素都将绑定一个特定的变量：

```

...
@Override
public void process(Object[] os) throws HiveException {
    sent = new Text(((StringObjectInspector)args[0])
        .getPrimitiveJavaObject(os[0]));
    String parts = new String(this.sent.getBytes());
    String [] part = parts.split("\\|");
    forwardObj[0]=Integer.parseInt( part[0] );
    forwardObj[1]=part[1] ;
    forwardObj[2]=part[2].split(",");
    this.forward(forwardObj);
}

@Override
public void close() throws HiveException {
}
}

```

我们在 Book 这个 UDTF 后面使用了 AS，这样可以允许用户自定义结果字段的别名。这些别名可以在查询的其他部分使用到，而无需重复从 Book 中解析出信息：

```

client.execute(
    "create temporary function book as 'com.jointhegrid.udf.collect.UDTFBook'");
client.execute("create table booktest (str string) ");
client.execute(
    "load data local inpath '" + p.toString() + "' into table booktest");
client.execute("select book(str) AS (book, title, authors) from booktest");
[555 Programming Hive "Dean","Jason","Edward"]

```

13.11 在 UDF 中访问分布式缓存

UDF 是可以访问分布式缓存、本地文件系统、甚至分布式文件系统中的文件的。不过这样的访问应该小心使用，因为这种使用会显著地降低执行效率。

Hive 的一个常用的使用场景就是分析网页日志。比较大众的一个操作就是基于 IP 地址进行地理位置定位。Maxmind 提供了一个 GeoIP 数据库，并提供了 Java API 来访问这

个数据库。通过将这个 API 包装成一个 UDF，就可以在 Hive 查询中通过 IP 地址查询对应的地理位置信息。

GeoIP API 会使用一个较小的数据文件。其对于通过 UDF 访问一个分布式缓存文件来说是非常理想的。本例的完整代码可以通过如下链接获得：<https://github.com/edwardcapriolo/hive-geoip/>。

ADD FILE 命令用于通过 Hive 将数据文件加载到分布式缓存中。而 ADD JAR 命令可以将指定的 Java JAR 文件加载到分布式缓存和类路径中。最后，在执行查询前需要创建临时函数：

```
hive> ADD FILE GeoIP.dat;
hive> ADD JAR geo-ip-java.jar;
hive> ADD JAR hive-udf-geo-ip-jtg.jar;
hive> CREATE TEMPORARY FUNCTION geoip
    > AS 'com.joanthegrid.hive.udf.GenericUDFGeoIP';

hive> SELECT ip, geoip(source_ip, 'COUNTRY_NAME', './GeoIP.dat') FROM weblog;
209.191.139.200      United States
10.10.0.1           Unknown
```

(译者注：这个 SQL 中 source_ip 应该是 ip，正确的语句应该是 SELECT ip, geoip(ip, 'COUNTRY_NAME', './GeoIP.dat') FROM weblog。)

上面的例子中返回了 2 条记录，其中一条显示这个 IP 是来自美国的，而另一条显示对应的 IP 无对应的地理位置信息。

这个 geoip() 函数含有 3 个参数：第 1 个参数是 IP 地址，可以是字符串类型或者 long 类型；第 2 个参数是一个字符串，可选的值有 COUNTRY_NAME 或 DMA_CODE；最后 1 个参数是数据文件名，这个文件已经是加载到分布式缓存中了。

对 UDF 的第一次调用（也就是触发调用 Java 函数中 evaluate 方法）会实例化一个 LookupService 对象来使用分布式缓存中的那个数据文件。这个查询是需要保存在一个引用中的。因此，其只需在 map 或者 reduce task 的初始化阶段初始一次，就可在对应 task 的整个生命周期中使用。需要注意的是，LookupService 具有自己的内置缓存，也就是 LookupService.GEOIP_MEMORY_CACHE。因此，这个优化可以避免在查询 IP 时，频繁访问磁盘。

下面是 evaluate() 方法的源码：

```
@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {
    if (argumentOIs[0] instanceof LongObjectInspector) {
        this.ipLong = ((LongObjectInspector) argumentOIs[0]).get(arguments
[0].get());
    } else {
```

```

        this.ipString = ((StringObjectInspector)argumentOIs[0])
            .getPrimitiveJavaObject(arguments[0].get());
    }
    this.property = ((StringObjectInspector)argumentOIs[1])
        .getPrimitiveJavaObject(arguments[1].get());
    if (this.property != null) {
        this.property = this.property.toUpperCase();
    }
    if (ls ==null){
        if (argumentOIs.length == 3){
            this.database = ((StringObjectInspector)argumentOIs[1])
                .getPrimitiveJavaObject(arguments[2].get());
            File f = new File(database);
            if (!f.exists())
                throw new HiveException(database+" does not exist");
            try {
                ls = new LookupService ( f , LookupService.GEOIP_MEMORY_CACHE );
            } catch (IOException ex){
                throw new HiveException (ex);
            }
        }
    }
    ...

```

evaluate 方法中的 if 语句决定了将返回哪个方法的数据。在本例中，需要返回的内容是国家名：

```

...
if (COUNTRY_PROPERTIES.contains(this.property)) {
    Country country = ipString != null ?
        ls.getCountry(ipString) : ls.getCountry(ipLong);
    if (country == null) {
        return null;
    } else if (this.property.equals(COUNTRY_NAME)) {
        return country.getName();
    } else if (this.property.equals(COUNTRY_CODE)) {
        return country.getCode();
    }
    assert(false);
} else if (LOCATION_PROPERTIES.contains(this.property)) {
    ...
}
}

```

13.12 以函数的方式使用注解

本章中我们提及过 Description 标注，并介绍了其用于在运行时为 Hive 方法提供说明文

档。UDF 中还存在有其他的标注，正确使用这些标注可以使得函数更加简单，甚至有时对于某些 Hive 查询，可以提高执行效率：

```
public @interface UDFType {
    boolean deterministic() default true;
    boolean stateful() default false;
    boolean distinctLike() default false;
}
```

13.12.1 定数性 (deterministic) 标注

默认情况下，对于大多数的查询来说，都是满足定数性的，因为它们本身就具有定数性。当然 rand()函数是个例外。

如果一个 UDF 是非定数的，那么就不会包含在分区裁剪中。

下面是使用 rand()函数的，具有非定数性的查询的一个例子：

```
SELECT * FROM t WHERE rand() < 0.01;
```

如果 rand()是定数的，那么结果只会在计算阶段计算一次。因为包含有 rand()的查询是非定数的，因此对于每行数据，rand()的值都需要重新计算一次。

13.12.2 状态性 (stateful) 标注

几乎所有的 UDF 默认都是有状态性的，而 rand()函数是无状态性的，因为其每次调用都返回不同的值。stateful 标注适用于如下情况。

- ① 有状态性的 UDF 只能使用在 SELECT 语句后面，而不能使用到其他如 WHERE、ON、ORDER、GROUP 等语句后面。
- ② 当一个查询语句中存在有状态性的 UDF 时，那么隐含的信息就是，SELECT 将会和 TRANSFORM (例如，一个 DISTRIBUTE、CLUSTER、SORT 语句) 进行类似的处理，然后会在对应的 reducer 内部进行执行，以保证结果是预期的结果。
- ③ 如果状态性标记 stateful 设置值为 true，那么这个 UDF 同样应该作为非定数性的 (即使这时定数性标记 deterministic 的值是显式设置为 true 的)。

详细信息请参考如下链接：<https://issues.apache.org/jira/browse/HIVE-1994>。

13.12.3 唯一性

有些函数，即使其输入的列的值是非排重值，其结果也是类似于使用了 DISTINCT 进行了排重操作，这类场景可定义为具有唯一性。这样的例子有 min 和 max 函数，即使实际数据中有重复值，其最终结果也是唯一排重值。

13.13 宏命令

宏命令提供了在 HiveQL 中调用其他函数和操作符来定义函数的功能。对于特定的情况，使用宏命令要比使用 Java 编写 UDF 或使用 Hive 的 streaming 功能更加方便，因为宏命令无需额外编写代码或脚本。

可以使用 CREATE TEMPORARY MACRO 语法来定义一个宏命令。下面是一个使用宏命令创建 SIGMOID 函数的例子：

```
hive> CREATE TEMPORARY MACRO SIGMOID (x DOUBLE) 1.0 / (1.0 + EXP(-x));  
hive> SELECT SIGMOID(2) FROM src LIMIT 1;
```

(译者注：这个功能实际到 Apache Hive 0.10.0 版本还没有真正提供，对于这个问题请参考：<https://issues.apache.org/jira/browse/HIVE-2655>)

Streaming

Hive 是通过利用或扩展 Hadoop 的组件功能来运行的，常见的抽象有 InputFormat、OutputFormat、Mapper 和 Reducer，还包含一些自己的抽象接口，例如 SerializerDeserializer (SerDe)、用户自定义函数 (UDF) 和 StorageHandlers。

这些组件都是 Java 组件，不过 Hive 将这些复杂的底层实现隐藏起来了，而提供给用户通过 SQL 语句执行的方式，而不是使用 Java 代码。

Streaming 提供了另一种处理数据的方式。在 streaming job 中，Hadoop Streaming API 会为外部进程开启一个 I/O 管道。然后数据会被传给这个进程，其会从标准输入中读取数据，然后通过标准输出来写结果数据，最后返回到 Streaming API job。尽管 Hive 并没有直接使用 Hadoop 的 Streaming API，不过它们的工作方式是一样的。

这种管道计算模型对于 Unix 操作系统及其衍生系统，如 Linux 和 Mac OS X 的用户来说是非常熟悉的。



提示

Streaming 的执行效率通常会比对应的编写 UDF 或改写 InputFormat 对象的方式要低。管道中序列化然后反序列化数据通常是低效的，而且以通常的方式很难调试整个程序。不过，对于快速原型设计和支持非 Java 编写的已有代码来说是非常有用的。对于那些不想写 Java 代码的 Hive 用户来说，这也是个高效的方式。

Hive 中提供了多个语法来使用 streaming，包括：MAP()、REDUCE()和 TRANSFORM()。要注意的重要的一点是 MAP()实际上并非可以强制在 map 阶段执行 streaming，正如 reduce 并非可以强制在 reduce 阶段执行 streaming 一样。因为这个原因，对于相同的功能，通常建议使用常规的 TRANSFORM()语句，这样可以避免误导读者对查询语句产

生疑惑。

对于我们的 streaming 例子，我们将使用到一个表名为 a 的小表，其中有 2 个字段，分别是 col1 和 col2，它们都是 INT 类型的，表中有 2 行数据：

```
hive> CREATE TABLE a (col1 INT, col2 INT)
> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

hive> SELECT * FROM a;
4      5
3      2

hive> DESCRIBE a;
a      int
b      int
```

14.1 恒等变换

最基本的 streaming job 就是恒等运算。/bin/cat 这个 shell 命令可以将传递给它的数据直接输出，所以满足恒等运算。本例中，/bin/cat 这个 shell 假定已经安装到所有的 TaskTracker 节点了。实际上任意的 Linux 系统都会包含有这个脚本的！稍后，我们将展示当一些程序没有安装到集群中时，通过 Hive 如何将它们“加载”到集群中：

```
hive> SELECT TRANSFORM (a, b)
> USING '/bin/cat' AS newA, newB
> FROM default.a;
4      5
3      2
```

14.2 改变类型

TRANSFORM 返回的字段的数据类型默认是字符串类型的。不过可以通过如下语法将类型转换成其他数据类型：

```
hive> SELECT TRANSFORM (col1, col2)
> USING '/bin/cat' AS (newA INT , newB DOUBLE) FROM a;
4      5.0
3      2.0
```

14.3 投影变换

Streaming 中可以使用 cut 命令提取或者映射出特定的字段。换句话说，可以达到和 SELECT 语句相同的行为：


```
hive> SELECT TRANSFORM (a, b)
  > USING '/bin/cut -f1'
  > AS newA, newB FROM a;
4 |      NULL
3 |      NULL
```

可以注意到，上面的例子中，查询从外部处理过程中返回的只有一个字段，而实际期望的是 2 个字段，因此字段 newB 的值总是 NULL。默认情况下，TRANSFORM 需要 2 个字段，不过实际上可以为比其小的任意个数的字段：

```
hive> SELECT TRANSFORM (a, b)
  > USING '/bin/cut -f1'
  > AS newA FROM a;
4
3
```

14.4 操作转换

/bin/sed 程序（对于 Mac OS X 系统是/usr/bin/sed）是一个流编辑器。其可以接受输入数据流，然后按照用户的指定进行编辑，最后将编辑后的结果输出的输出数据流中。如下例子中将字符串“4”替换成了字符串“10”：

```
hive> SELECT TRANSFORM (a, b)
  > USING '/bin/sed s/4/10/'
  > AS newA, newB FROM a;
10 5
3   2
```

14.5 使用分布式内存

到目前为止所列举的 streaming 例子都是 UNIX 系统或其衍生系统自带的如 cat 和 sed 这样的系统脚本程序。当一个查询所需要的文件没有在每个 TaskTracker 上事先安装好时，用户需要使用分布式缓存将数据或者程序文件传输到集群中，然后在 job 完成后会清理掉这些数据和文件。（译者注：Hadoop 的分布式缓存可以对缓存内的文件安装 LRU 原则进行删除，因此并非是 job 一结束就立即删除掉文件的。）

这个功能很有用，因为在大规模集群上安装（有时需要卸载）大量的小组件会成为一件很有负担的事情。同时，缓存中会独立保存每个 job 的缓存文件，而不会相互干扰。

下面的例子是一个将摄氏温度转换成华氏温度的 bash shell 脚本：

```
while read LINE
do
  res=$(echo "scale=2; ((9/5) * $LINE) + 32" | bc)
  echo $res
```

```
done
```

可以在本地执行这个脚本来测试一下。这个脚本不会提示输入。输入 100，然后按回车键，这时这个进程会通过标准输出打印出 212.00；输入另一个数值，然后程序就会返回另一个相应的结果。用户可以持续地输入数值，也可以通过 Control + D 组合键退出程序终止输入。

```
#!/bin/bash
$ sh ctof.sh
100
212.00
0
32.00
^D
```

Hive 的 ADD FILE 功能可以将文件加入到分布式缓存中。而被增加的文件会被存储到每个 task 节点机器的当前工作目录下。这样可以使得 transform task 直接使用脚本而不用确定到哪里去找这些文件：

```
hive> ADD FILE ${env:HOME}/prog_hive/ctof.sh;
Added resource: /home/edward/prog_hive/ctof.sh

hive> SELECT TRANSFORM(col1) USING 'ctof.sh' AS convert FROM a;
39.20
37.40
```

14.6 由一行产生多行

到目前为止所列举的例子都是获取一行输入然后产生一行输出。同样，可以使用 Streaming 对每个输入行产生多行输出。这个功能可以产生类似于 EXPLODE() UDF 和 LATERAL VIEW 语法¹的输出。

给定一个输入文件 \$HOME/kv_data.txt，其内容如下：

```
k1=v1,k2=v2
k4=v4,k5=v5,k6=v6
k7=v7,k7=v7,k3=v7
```

我们希望这些数据能够以表格的形式展示。这样就可以让常见的 HiveQL 操作符处理这些行：

```
k1    v1
```

1.这个例子所使用到的源码和概念来自于 Larry Ogrodnik 在 2009 年 7 月 14 日在 Bizo 开发博客上发表的“Custom Map Scripts and Hive”博文。

```
k2    v2
k4    k4
```

创建如下这个 Perl 脚本，然后保存为\$HOME/split_kv.pl:

```
#!/usr/bin/perl
while (<STDIN>) {
    my $line = $_;
    chomp($line);
    my @kvs = split(/,/ , $line);
    foreach my $p (@kvs) {
        my @kv = split(/=/, $p);
        print $kv[0] . "\t" . $kv[1] . "\n";
    }
}
```

创建一个名为 kv_data 的表。这张表只定义了一个字段。行格式无需进行配置，因为 streaming 脚本会对输入字段进行分割:

```
hive> CREATE TABLE kv_data ( line STRING );

hive> LOAD DATA LOCAL INPATH '${env:HOME}/kv_data.txt' INTO TABLE kv_data;
```

对源表使用 transform 脚本。这样之前凌乱的，一行多个条目的格式就转化成了一个具有 2 个字段的键-值对数据:

```
hive> SELECT TRANSFORM (line)
> USING 'perl split_kv.pl'
> AS (key, value) FROM kv_data;
k1      v1
k2      v2
k4      v4
k5      v5
k6      v6
k7      v7
k7      v7
k3      v7
```

14.7 使用 streaming 进行聚合计算

同样可以使用 streaming 做类似于 Hive 的内置函数 SUM 一样的聚合运算。这是因为 streaming 处理过程可以对每行输入返回 0 或多行输出。

为了能在一个外部应用程序中完成聚合，脚本中在循环外面先定义了一个计数器，然后，循环内部不断从输入流中读取输入并进行计算，最终输出 sum 求和结果:

```
#!/usr/bin/perl
my $sum=0;
```

```

while (<STDIN>) {
    my $line = $_;
    chomp($line);
    $sum=${sum}+${line};
}
print $sum;

```

创建一个表并导入整数数据，每行一个整数，用于测试：

```

hive> CREATE TABLE sum (number INT);

hive> LOAD DATA LOCAL INPATH '${env:HOME}/data_to_sum.txt' INTO TABLE sum;
hive> SELECT * FROM sum;
5
5
4

```

将 `streaming` 程序增加到分布式缓存中，并在 `TRANSFORM` 查询中使用。这个处理过程将会返回一行结果，也就是对输入求和后的结果：

```

hive> ADD FILE ${env:HOME}/aggregate.pl;
Added resource: /home/edward/aggregate.pl

hive> SELECT TRANSFORM (number)
> USING 'perl aggregate.pl' AS total FROM sum;
14

```

不过可惜的是，不能够像 `UDAF SUM()`那样在单个查询中执行多个 `TRANSFORM` 过程。例如：

```

hive> SELECT sum(number) AS one, sum(number) AS two FROM sum;
14      14

```

同时，对于中间数据，如果不使用 `CLUSTER BY` 或者 `DISTRIBUTE BY` 的话，那么这个 `job` 可能会执行单个的非常长时间的 `map` 或者 `reduce task`。尽管并非所有的操作都可以并行执行，不过大多数操作还是可以的。下一节将讨论如何在需要的时候以并行的方式执行 `streaming`。

14.8 CLUSTER BY、DISTRIBUTE BY、SORT BY

Hive 提供了语法来控制数据是如何被分发和排序的。这些功能可以应用在大多数的查询中，不过在执行 `streaming` 处理时显得特别有用。例如，具有相同键的数据需要分发到同一个处理节点上，或者数据需要按照指定列或指定函数进行排序。Hive 提供了多种方式来控制这种行为。

第 1 种控制这种方法就是 `CLUSTER BY` 语句，其可以确保类似的数据可以分发

到同一个 reduce task 中，而且保证数据是有序的。

为了演示 CLUSTER BY 的用法，让我们来看一个特殊的例子：通过另外一种方式来实现第 1 章中所介绍的 Word Count 算法。现在，我们将使用到 TRANSFORM 功能和 2 个 Python 脚本，一个脚本用于将读取的文本的每行内容分割成单词；另一个脚本用于接受字频数据流以及单词的中间计数值（大多数是数字“1”），然后对每个单词的词频求和汇总。

下面是第 1 个 Python 脚本，其可以按照空格将每行内容分割成单词（按空格划分对于标点符号等可能不太合适）：

```
import sys

for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print "%s\t1" % (word.lower())
```

不必说明所有的 Python 语法，可以很容易看到，这个脚本从通用模块 sys 中引入了常见的函数，然后循环获取“标准输入流”中每行内容（也就是调用 stdin 函数），再按照空格对每行内容进行划分，生成一个单词集合 words，然后遍历整个集合并输出每个单词以及一个制表符（也就是\t），最后输出每个单词对应的词频¹。

在我们展示第 2 个 Python 脚本之前，让我们先讨论下传递给这个脚本的数据。在我们的 TRANSFORM Hive 查询中，我们将对第 1 个 Python 脚本的输出词组使用 CLUSTER BY。这样可以将所有相同的单词分配到同一组中，每行一对数据，每对的数据形式是单词\t次数：

```
word1 1
word1 1
word1 1
word2 1
word3 1
word3 1
...
```

因此，第 2 个 Python 脚本将会更复杂些，因为其需要缓存当前处理的单词，以及迄今为止这个单词出现的次数。当处理下一个单词时，这个脚本需要输出上一个单词的词频数，然后重置缓存。

下面就是第 2 个 Python 脚本：

1. 这是最简单的处理方式。我们也可以缓存单词的次数并直接输出最终词频数。这样处理会减少 I/O 开销，因此将会更快，但是也因此，其实现起来要更加复杂。

```

import sys

(last_key, last_count) = (None, 0)
for line in sys.stdin:
    (key, count) = line.strip().split("\t")
    if last_key and last_key != key:
        print "%s\t%d" % (last_key, last_count)
        (last_key, last_count) = (key, int(count))
    else:
        last_key = key
        last_count += int(count)

if last_key:
    print "%s\t%d" % (last_key, last_count)

```

我们将假设这 2 个 Python 脚本都在用户根目录下。

最后,我们来看下 2 个脚本结合使用的 Hive 查询语句。首先,我们通过 CREATE TABLE 语句创建一个输入表,表内容包含多行的文本数据。这个表我们在第 1 章使用过。任意文本文件都可以作为这张表的数据。其次,创建的表 word_count 用于保存计算后的输出结果。其有 2 个字段,一个是 word(表示单词),一个是 count(表示次数)。同时字段分隔符是\t。最后就是结合使用这 2 个脚本进行处理的 TRANSFORM 语句了:

```

hive> CREATE TABLE docs (line STRING);

hive> CREATE TABLE word_count (word STRING, count INT)
> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

hive> FROM (
> FROM docs
> SELECT TRANSFORM (line) USING '${env:HOME}/mapper.py'
> AS word, count
> CLUSTER BY word) wc
> INSERT OVERWRITE TABLE word_count
> SELECT TRANSFORM (wc.word, wc.count) USING '${env:HOME}/reducer.py'
> AS word, count;

```

USING 语句指定了 Python 脚本所在的绝对路径。

替代 CLUSTER BY 的最方便的方式就是使用 DISTRIBUTE BY 和 SORT BY。使用它们的常见场景就是,用户期望将数据按照某个字段划分,然后按照另一个字段排序。实际上,CLUSTER BY word 等价于 DISTRIBUTE BY word SORT BY word ASC。

如下 TRANSFORM 查询的字频输出结果是按照降序排序的:

```

FROM (
    FROM docs
    SELECT TRANSFORM (line) USING '/../mapper.py'

```

```

AS word, count
DISTRIBUTE BY word SORT BY word DESC) wc
INSERT OVERWRITE TABLE word_count
SELECT TRANSFORM (wc.word, wc.count) USING '/.../reducer.py'
AS word, count;

```

使用 `CLUSTER BY` 或者使用结合 `SORT BY` 的 `DISTRIBUTE BY` 是非常重要的。因为如果没有这些指示，Hive 可能无法合理地并行执行 job，所有的数据可能都会分发到同一个 reducer 上，这样就会导致整体 job 执行时间延长。

14.9 GenericMR Tools for Streaming to Java

通常情况下，使用 streaming 是为了将非 Java 的代码结合到 Hive 中。正如我们所看见的，streaming 几乎适用于所有的语言。使用 Java 编写 streaming 也是可以的，而且 Hive 中包含了 GenericMR API 来试图为 streaming 提供类似于 Hadoop 的 MapReduce API 的这样的接口：

```

FROM (
  FROM src
  MAP value, key
  USING 'java -cp hive-contrib-0.9.0.jar
        org.apache.hadoop.hive.contrib.mr.example.IdentityMapper'
  AS k, v
  CLUSTER BY k) map_output
REDUCE k, v
USING 'java -cp hive-contrib-0.9.0.jar
      org.apache.hadoop.hive.contrib.mr.example.WordCountReduce'
AS k, v;

```

为了清楚 `IdentityMapper` 是怎么编写的，我们可以看一下 GenericMR 所提供的接口。其中 `Mapper` 接口用于实现常见的 `Mapper` 实现方法。其提供了一个 `map` 方法，这里输入是字符串数组类型 `String[]` 的列值：

```

package org.apache.hadoop.hive.contrib.mr;

public interface Mapper {
    void map(String[] record, Output output) throws Exception;
}

```

`IdentityMapper` 不会对输入数据做任何改变而会将其传递给收集器 (collector)。在功能上这和本章前面部分所介绍的 `/bin/cat` 的功能是一致的：

```

package org.apache.hadoop.hive.contrib.mr.example;

import org.apache.hadoop.hive.contrib.mr.GenericMR;
import org.apache.hadoop.hive.contrib.mr.Mapper;

```

```

import org.apache.hadoop.hive.contrib.mr.Output;

public final class IdentityMapper {
    public static void main(final String[] args) throws Exception {
        new GenericMR().map(System.in, System.out, new Mapper() {
            @Override
            public void map(final String[] record, final Output output) throws
Exception {
                output.collect(record);
            }
        });
    }

    private IdentityMapper() {
    }
}

```

Reducer 接口提供了第一列字符串，而其他列可以通过记录迭代器获得。每次迭代都会返回一对字符串，其中第 0 个元素是重复的键而其下一次元素是值。output 对象表示输出结果：

```

package org.apache.hadoop.hive.contrib.mr;

import java.util.Iterator;

public interface Reducer {
    void reduce(String key, Iterator<String[]> records, Output output)
        throws Exception;
}

```

WordCountReduce 类中有一个累加器用于对记录迭代器中的每个元素进行计数。当所有的记录都被计数后，就会生成一个由键及其对应次数所组成的数组作为结果：

```

package org.apache.hadoop.hive.contrib.mr.example;

import java.util.Iterator;
import org.apache.hadoop.hive.contrib.mr.GenericMR;
import org.apache.hadoop.hive.contrib.mr.Output;
import org.apache.hadoop.hive.contrib.mr.Reducer;

public final class WordCountReduce {

    private WordCountReduce() {
    }

    public static void main(final String[] args) throws Exception {
        new GenericMR().reduce(System.in, System.out, new Reducer() {
            public void reduce(String key, Iterator<String[]> records, Output
output)
                throws Exception {

```



```

int count = 0;
while (records.hasNext()) {
    // note we use col[1] -- the key is provided again as col[0]
    count += Integer.parseInt(records.next()[1]);
}
output.collect(new String[] {key, String.valueOf(count)});
}
});
}
}

```

14.10 计算 cogroup

在 MapReduce 程序中经常会对多数据集进行 JOIN 连接处理，然后使用 TRANSFORM 进行处理。使用 UNION ALL 和 CLUSTER BY，我们可以实现 GROUP BY 操作的常见效果。（译者注：这里应该是 COGROUP BY 而不是 GROUP BY）



提示

Pig 提供了一个原生的 COGROUP BY 操作。

假设我们有多个不同源的日志文件，它们具有相同的 schema。我们期望将它们合并在一起，然后通过一个 reduce_script 脚本进行分析：

```

FROM (
  FROM (
    FROM order_log ol
    -- User Id, order Id, and timestamp:
    SELECT ol.userid AS uid, ol.orderid AS id, av.ts AS ts

    UNION ALL

    FROM clicks_log cl
    SELECT cl.userid AS uid, cl.id AS id, ac.ts AS ts
  ) union_msgs
  SELECT union_msgs.uid, union_msgs.id, union_msgs.ts
  CLUSTER BY union_msgs.uid, union_msgs.ts) map
INSERT OVERWRITE TABLE log_analysis
SELECT TRANSFORM(map.uid, map.id, map.ts) USING 'reduce_script'
AS (uid, id, ...);

```

自定义 Hive 文件和记录格式

Hive 可以通过多种方式自定义其功能。首先, Hive 具有变量和属性, 这个我们在第 2.7.2 节“变量和属性”中讨论过了。其次, 用户可以通过自定义 UDF 来扩展 Hive 功能, 在第 13 章我们有讨论过。最后, 用户可以自定义文件和记录格式, 这个是本章将进行讨论的内容。

15.1 文件和记录格式

Hive 中文件格式间具有明显的差异, 例如文件中记录的编码方式、记录格式以及记录中字节流的编码方式都是不同的。

本书到目前为止都是使用文本文件格式存储的, 也就是 CREATE TABLE 语句中默认的 STORED AS TEXTFILE (请参考第 3.3 节“文本文件数据编码”中的介绍)。在文本文件中每一行数据就是一条记录。大多数情况下这些记录使用默认的分隔符, 偶然有一些样例数据使用逗号或回车键作为字段分隔符。不过, 文本文件统一可以包含 JSON 或者 XML “文档”。

对于 Hive 而言, 文本文件格式选择和记录格式是对应的。我们将首先讨论文本格式, 然后再讨论不同的记录格式, 以及在 Hive 中如果使用这些格式。

15.2 阐明 CREATE TABLE 句式

贯穿本书, 我们已经介绍了创建表的例子。用户可能已经发现 CREATE TABLE 语句具有多种多样的语法。例如 STORED AS SEQUENCEFILE、ROW FORMAT DELIMITED、SERDE、INPUTFORMAT、OUTPUTFORMAT 这些语法。本章将涵括这个语法的大部分内容并给出样例。但是需要注意的是, 某些语法是其他语法的快捷用法, 也就是说,

可以使概念变得更易理解的语法。例如，语法 `STORED AS SEQUENCEFILE` 的替代方式是指定 `INPUTFORMAT` 为 `org.apache.hadoop.mapred.SequenceFileInputFormat`，并指定 `OUTPUTFORMAT` 为 `org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat`。

下面我们创建一些表然后使用 `DESCRIBE TABLE EXTENDED` 语句来查看下内部的实际变化情况。首先，我们将创建一个简单的表，然后对其进行描述（这里我们对输出进行了格式化，实际上 Hive 本身输出效果不是这样）（译者注：从 Apache Hive v0.10.0 开始 `EXTENDED` 输出也将是部分格式化的，和当前的输出效果还是有差异的）：

```
hive> create table text (x int) ;
hive> describe extended text;
OK
x      int
```

```
Detailed Table Information
Table(tableName:text, dbName:default, owner:edward, createTime:1337814583,
lastAccessTime:0, retention:0,
sd:StorageDescriptor(
  cols:[FieldSchema(name:x, type:int, comment:null)],
  location:file:/user/hive/warehouse/text,
  inputFormat:org.apache.hadoop.mapred.TextInputFormat,
  outputFormat:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat,
  compressed:false,
  numBuckets:-1,
  serdeInfo:SerDeInfo(
    name:null,
    serializationLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
    parameters:{serialization.format=1}
  ),
  bucketCols:[], sortCols:[], parameters:{}), partitionKeys:[],
parameters:{transient_lastDdlTime=1337814583},
viewOriginalText:null, viewExpandedText:null, tableType:MANAGED_
TABLE
)
```

现在我们再使用 `STORED AS SEQUENCEFILE` 语句创建一张表，用于对比：

```
hive> CREATE TABLE seq (x int) STORED AS SEQUENCEFILE;
hive> DESCRIBE EXTENDED seq;
OK
x      int
```

```
Detailed Table Information
Table(tableName:seq, dbName:default, owner:edward, createTime:1337814571,
lastAccessTime:0, retention:0,
sd:StorageDescriptor(
  cols:[FieldSchema(name:x, type:int, comment:null)],
  location:file:/user/hive/warehouse/seq,
```

```

inputFormat:org.apache.hadoop.mapred.SequenceFileInputFormat,
outputFormat:org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat,
compressed:false, numBuckets:-1,
serdeInfo:SerDeInfo(
  name:null,
  serializationLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
  parameters:{serialization.format=1}
),
bucketCols:[], sortCols:[], parameters:{}, partitionKeys:[],
parameters:{transient_lastDdlTime=1337814571},
viewOriginalText:null, viewExpandedText:null, tableType:MANAGED_TABLE
)
Time taken: 0.107 seconds

```

除非被 Hive 的强大功能所迷惑，否则很容易找出这 2 个表的差异。STORED AS SEQUENCEFILE 语句改变了 InputFormat 和 OutputFormat 的值：

```

inputFormat:org.apache.hadoop.mapred.TextInputFormat,
outputFormat:org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat,

inputFormat:org.apache.hadoop.mapred.SequenceFileInputFormat,
outputFormat:org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat,

```

当从表中读取数据时，Hive 会使用到 InputFormat；而当向表中写入数据时，其会使用 OutputFormat。



提示

InputFormat 会从文件中读取键-值对。默认情况下，Hive 会直接忽略掉键的内容而只是有值中的数据。这是因为键来自于 TextInputFormat，是表示块中的字节边界的一个长整型数值(其并非用户数据)。

本章后面部分将描述表元数据的其他方面信息。

15.3 文件格式

我们在第 3.3 节“文本文件数据编码”中讨论过使用的最简单的数据格式是文本文件格式，可以任意的分隔符进行分割。同时这也是默认的文件格式，等价于在创建表时通过 STORED AS TEXTFILE 语句指定使用文本存储格式。

文本文件格式便于和其他工具（如 Pig，Unix 文本工具中的 grep、sed 和 awk 等）共享数据。同时文本文件也便于查看和编辑。不过，相对于二进制文件，文本文件存储空间要大。正如在第 11 章中我们讨论过的，可以使用压缩，但是如果使用二进制文件存储格式的话，则既可以节约存储空间，也可以提高 I/O 性能。

15.3.1 SequenceFile

其中一种存储格式是 SequenceFile 文件存储格式，在定义表结构时可以通过 STORED AS SEQUENCEFILE 语句指定。

SequenceFile 文件是含有键-值对的二进制文件。当 Hive 将查询转换成 MapReduce job 时，对于指定的记录，其取决使用哪些合适的键-值对。

SequenceFile 是 Hadoop 本身就可以支持的一种标准文件格式，因为它也是在 Hive 和其他 Hadoop 相关的工具中共享文件的可以接受的选择。而对于 Hadoop 生态系统之外的其他工具而言，其并不适合使用。正如我们在第 11 章中讨论过的，SequenceFile 可以在块级别和记录级别进行压缩，这对于优化磁盘利用率和 I/O 来说非常有意义。同时仍然可以支持按照块级别的文件分割，以方便并行处理。

Hive 所支持的另一种高效二进制文件格式就是 RCFile。

15.3.2 RCfile

大多数的 Hadoop 和 Hive 存储都是行式存储的，在大多数场景下，这是比较高效的。这种高效归根于如下几个原因：大多数的表具有的字段个数都不大（一般就 1 到 20 个字段）；对文件按块进行压缩对于需要处理重复数据的情况比较高效，同时很多的处理和调试工具（例如 more、head、awk）都可以很好地应用于行式存储的数据。

并非所有的工具和数据存储都是采用行式存储的方式的，对于特定类型的数据和应用来说，采用列式存储有时会更好。例如，如果指定的表具有成百上千个字段，而大多数的查询只需要使用到其中的一小部分字段，这时扫描所有的行而过滤掉大部分的数据显然是个浪费。然而，如果数据是按照列而不是行进行存储的话，那么只要对其需要的列的数据进行扫描就可以了，这样可以提高性能。

对于列式存储而言，进行压缩通常会非常高效，特别是在这列的数据具有较低计数的时候（只有很少的排重值时）。同时，一些列式存储并不需要物理存储 null 值的列。

基于这些场景，Hive 中才设计了 RCFile。

尽管本书提供了非常有价值的信息资源，但是有时查找信息的最好方式还是查看源代码本身。关于 Hive 中的像 RCFile 这样的列式存储是如何工作的最好的描述可以在源代码中看到：

```
cd hive-trunk
find . -name "RCFile*"
vi ./ql/src/java/org/apache/hadoop/hive/ql/io/RCFile.java
* <p>
* RCFile stores columns of a table in a record columnar way. It first
* partitions rows horizontally into row splits. and then it vertically
* partitions each row split in a columnar way. RCFile first stores the
meta
```

```
* data of a row split, as the key part of a record, and all the data of
a row
* split as the value part.
* </p>
```

Hive 功能强大的一个方面体现在在不同的存储格式间转换数据非常地简单。存储信息存放在表的元数据信息中。当对表执行一个 SELECT 查询时，以及向其他表中执行 INSERT 操作时，Hive 就会使用这个表的元数据信息中提供的内容，然后自动执行转换过程。这样使用可以有多种选择，而不需要额外的程序来对不同的存储格式进行转换。

可以在创建表时使用 ColumnarSerDe、RCFileInputFormat 和 RCFileOutputFormat:

```
hive> select * from a;
OK
4      5
3      2
Time taken: 0.336 seconds
hive> create table columnTable (key int , value int)
> ROW FORMAT SERDE
> 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe'
> STORED AS
> INPUTFORMAT 'org.apache.hadoop.hive.ql.io.RCFileInputFormat'
> OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.RCFileOutputFormat';

hive> FROM a INSERT OVERWRITE TABLE columnTable SELECT a.col1, a.col2;
```

不可以使用通常可以打开 SequenceFile 的工具来打开 RCFile。不过，Hive 提供了一个 rfilecat 工具来展示 RCFile 文件内容:

```
$ bin/hadoop dfs -text /user/hive/warehouse/columntable/000000_0
text: java.io.IOException: WritableName can't load class:
org.apache.hadoop.hive.ql.io.RCFile$KeyBuffer
$ bin/hive --service rfilecat /user/hive/warehouse/columntable/000000_0
4      5
3      2
```

15.3.3 示例自定义输入格式: DualInputFormat

很多种数据库允许用户执行没有 FROM 语句的 SELECT 语句。这可用于执行简单的计算过程，例如 SELECT 1+2。如果 Hive 不允许执行这种类型的查询的话，那么其实可以通过从某个存在的标准中选取数据，然后将结果限制到一行，来达到同样的效果；或者用户可以创建一个只有一行数据的表。一些数据库提供了一个名为 dual 的表，其只有一行数据，用来满足这种使用场景。

默认情况下，标准的 Hive 表使用的输入格式是 TextInputFormat。TextInputFormat 会对输入的零个或多个划分进行计算。这个框架可以打开这些划分文件，然后使用一个

`RecordReader` 来读取划分中的数据。文本中每一行都会成为一条输入记录。为了创建一个适用于 `dual` 表的输入格式，我们需要创建一个输入格式，要求其只会返回一个只有一行数据的划分，而不管输入路径是什么¹。

在下面这个例子中，`DualInputFormat` 只会返回一个划分：

```
public class DualInputFormat implements InputFormat{
    public InputSplit[] getSplits(JobConf jc, int i) throws IOException {
        InputSplit [] splits = new DualInputSplit[1];
        splits[0]= new DualInputSplit();
        return splits;
    }
    public RecordReader<Text,Text> getRecordReader(InputSplit split,
JobConf jc,
        Reporter rprtr) throws IOException {
        return new DualRecordReader(jc, split);
    }
}
```

下面这个例子中划分只有一行数据。不需要进行序列化或反序列化操作：

```
public class DualInputSplit implements InputSplit {
    public long getLength() throws IOException {
        return 1;
    }
    public String[] getLocations() throws IOException {
        return new String [] { "localhost" };
    }
    public void write(DataOutput d) throws IOException {
    }
    public void readFields(DataInput di) throws IOException {
    }
}
```

`DualRecordReader` 中有一个布尔值变量 `hasNext`。当第 1 次调用 `next ()` 函数后，其值就设置成了 `false`。因此，这个记录读取器返回唯一一行，然后就结束了：

```
public class DualRecordReader implements RecordReader<Text,Text>{
    boolean hasNext=true;
    public DualRecordReader(JobConf jc, InputSplit s) {
    }
    public DualRecordReader(){
    }
    public long getPos() throws IOException {
        return 0;
    }
}
```

1.可以通过如下链接获取到 `DualInputFormat` 的源码：<https://github.com/edwardcapriolo/DualInputFormat>。

```

public void close() throws IOException {
}
public float getProgress() throws IOException {
    if (hasNext)
        return 0.0f;
    else
        return 1.0f;
}
public Text createKey() {
    return new Text("");
}
public Text createValue() {
    return new Text("");
}
public boolean next(Text k, Text v) throws IOException {
    if (hasNext){
        hasNext=false;
        return true;
    } else {
        return hasNext;
    }
}
}
}
}

```

下面我们创建一个使用 `DualInputFormat` 输入格式和默认的 `HiveIgnoreKeyTextOutPutFormat` 输出格式的表，然后从这个表中选取下数据，看其是否只访问一个空行。输入格式的 jar 包需要包含在 HADOOP 的 lib 目录下或者放置在 Hive 的 auxlib 目录下。

```

client.execute("add jar dual.jar");
client.execute("create table dual (fake string) "+
    "STORED AS INPUTFORMAT 'com.m6d.dualinputformat.DualInputFormat'"+
    "OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutput"+
    "Format'");
client.execute("select count(1) as cnt from dual");
String row = client.fetchOne();
assertEquals("1", row);
client.execute("select * from dual");
row = client.fetchOne();
assertEquals( "", row);

```

15.4 记录格式：SerDe

SerDe 是序列化/反序列化的简写形式。一个 SerDe 包含了将一条记录的非结构化字节（它是文件存储的一部分）转化成 Hive 可以使用的一条记录的过程。可以使用 Java 来实现 SerDe。Hive 本身自带了几个内置的 SerDe，还有其他一些第三方的 SerDe 可供选

择使用。

在内部，Hive 引擎使用定义的 InputFormat 来读取一行数据记录。这行记录之后会被传递给 SerDe.deserialize()方法进行处理。

简单的 SerDe 除非需要特定的属性时才会完整地实例化某个对象。

下面这个例子使用一个 RegexSerDe 来处理标准格式的 Apache Web 日志。这个 RegexSerDe 是作为 Hive 分支的标准功能来使用的：

```
CREATE TABLE serde_regex (
  host STRING,
  identity STRING,
  user STRING,
  time STRING,
  request STRING,
  status STRING,
  size STRING,
  referer STRING,
  agent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\|\\|[^\\|\\|]*\\|\\|)
  ([^ \\"]*|\"[^\"]*\\") (-|[0-9]*) (-|[0-9]*) (?: ([^ \\"]*|\"[^\"]*\\")
  ([^ \\"]*|\"[^\"]*\\\"))?",
  "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s"
)
STORED AS TEXTFILE;
```

现在我们可以载入数据，然后查询看看了：

```
hive> LOAD DATA LOCAL INPATH "../data/files/apache.access.log" INTO TABLE
serde_regex;
hive> LOAD DATA LOCAL INPATH "../data/files/apache.access.2.log" INTO
TABLE serde_regex;

hive> SELECT * FROM serde_regex ORDER BY time;
```

(为便于展示，这里有调整正则表达式的显示)

15.5 CSV 和 TSV SerDe

那么 CSV (逗号分隔值) 和 TSV (回车键分隔符) 文件呢? 当然，对于如数值数据的简单数据来说，就和我们前面看到的一样，用户只需要使用默认的文件格式并制定字段分隔符即可。不过，这种简单的处理方式既无法处理字符串中自身包含逗号或回车键的情况，也无法处理其他常见的转换，例如，是否对字符串都加引号或不加引号，或者每个文件的第一行是否需要显示“列名数据头”。

首先，如果有列名数据头的话，那么通常将这行数据移除也是可以的。这样有多种第三方 SerDe 来处理 CSV 或者 TSV 文件。对于 CSV 文件来说，可以考虑使用 CSVSerDe：

```
ADD JAR /path/to/csv-serde.jar;

CREATE TABLE stocks(ymd STRING, ...)
ROW FORMAT SERDE 'com.bizo.hive.serde.csv.CSVSerde'
STORED AS TEXTFILE
...;
```

尽管支持 TSV 应该是与 CSV 类似的，但是在写本书时还没有合适的第三方 TSV SerDe 可供使用。

15.6 ObjectInspector

在外壳的下面，Hive 使用一个名为 ObjectInspector 的检查器来将记录转换成 Hive 可以访问的对象。

15.7 Thing Big Hive Reflection ObjectInspector

Think Big Analytics 创建了一个名为 BeansStructObjectInspector 的基于 Java 反射机制的 ObjectInspector。使用 JavaBean 模型的反射机制，任何对象的“属性”信息都可以通过 get 方法获得，或者作为 public 成员变量在查询中进行引用。

下面是个介绍如何使用 BeansStructObjectInspector 的例子：

```
public class SampleDeserializer implements Deserializer {
    @Override
    public ObjectInspector getObjectInspector() throws SerDeException {
        return BeansStructObjectInspector.getBeansObjectInspector(YourObject.
class);
    }
}
```

15.8 XML UDF

XML 天生就是非结构化的，这使得 Hive 成为 XML 处理的一个强大的数据库平台。众多原因之一就是大型 XML 文档解析和处理所需要的复杂性和资源消耗使得 Hadoop 非常适合作为一个 XML 数据库平台，因为 Hadoop 可以并行地处理 XML 文档，Hive 也成为了解决 XML 相关数据的完美工具。此外，HiveQL 原生就支持访问 XML 的嵌套元素和值，然后进一步地，可以允许对嵌套的字段、值和属性进行连接操作。

XPath (XML 路径语言) 是一个由 W3C 创建的用于定位 XML 文档指定内容的全球性

的标准。使用 XPath 作为 XML 查询语言的话 Hive 可以从 XML 中提取数据并进入 Hive 系统中进行其他处理，这使得 Hive 对于 XML 处理显得极其有用。

XPath 将 XML 文档建模成一个节点树。并提供了访问如字符串、数值和布尔型等基本数据类型的功能。

尽管如 Oracle XML DB 和 MarkLogic 等商业解决方案都提供了原生的 XML 数据库解决方案，但作为开源软件的 Hive 利用 Hadoop 的 PB 级别的并行处理架构给更广泛高效的 XML 数据库带来了更多的生气。

15.9 XPath 相关的函数

Hive 中包含有一些从 Hive 0.6.0 发行版中引入的 XPath 相关的 UDF（见表 15-1）。

表 15-1 XPath 相关的 UDF

UDF 名称	描述
xpath	返回 Hive 中的一组字符串数组
xpath_string	返回一个字符串
xpath_boolean	返回一个布尔值
xpath_short	返回一个短整型数值
xpath_int	返回一个整型数值
xpath_long	返回一个长整型数值
xpath_float	返回一个浮点数数值
xpath_double, xpath_number	返回一个双精度浮点数数值

下面的例子展示的是对于常数字符串使用这些函数的情况：

```
hive> SELECT xpath('\<a><b id="foo">b1</b><b id="bar">b2</b></a>\', '\\//@id\')
> FROM src LIMIT 1;
[foo", "bar]
```

```
hive> SELECT xpath ('\<a><b class="bb">b1</b><b>b2</b><b>b3</b><c class="bb">c1</c>
<c>c2</c></a>\', \'a/*[@class="bb"]/text()\')
> FROM src LIMIT 1;
[b1", "c1]
```

（为排版好看，所以将例子中长长的 XML 字符串进行了转行。）

```
hive> SELECT xpath_double ('\<a><b>2</b><c>4</c></a>\', \'a/b + a/c\')
> FROM src LIMIT 1;
6.0
```

15.10 JSON SerDe

如果用户想使用 Hive 查询 JSON 格式的数据，那该怎么办呢？如果每行都只有一个 JSON “文件”的话，那么用户可以使用 TEXTFILE 作为输入输出文件格式，然后使用一个 JSON 的 SerDe 来将 JSON 文档作为一条记录进行解析。

有一个第三方的 JSON SerDe，其是在一个 Google “编程夏令营”项目中开启的，随后被其他的社区贡献者克隆和开启新的分支。Think Big Analytics 也创建了自己的分支并增加了增强功能，后面我们将进行讨论。

在下面例子中，这个 SerDe 用于从 JSON 数据中抽取出一些域，这些数据假设是来自某个信息系统的。并非要解析出 JSON 中所有的字段。而那些解析出来的字段都将作为表的字段：

```
CREATE EXTERNAL TABLE messages (  
    msg_id      BIGINT,  
    tstamp     STRING,  
    text       STRING,  
    user_id    BIGINT,  
    user_name  STRING  
)  
ROW FORMAT SERDE "org.apache.hadoop.hive.contrib.serde2.JsonSerde"  
WITH SERDEPROPERTIES (  
    "msg_id"="$$.id",  
    "tstamp"="$$.created_at",  
    "text"="$$.text",  
    "user_id"="$$.user.id",  
    "user_name"="$$.user.name"  
)  
LOCATION '/data/messages';
```

上面语句中的 WITH SERDEPROPERTIES 是 Hive 提供的一个功能，允许用户定义一些可以传递给 SerDe 的属性信息。在合适的情况下 SerDe 可以读取这些属性。Hive 本身并不关心这些属性表达什么含义。

在本例中，这些属性用于将 JSON 文档和表的字段对应起来。像 \$.user.id 这样的字符串表示获取每条记录，用 \$ 来表示，然后查找 user 的键。本例中也就是假定其是一个 JSON map，最后解析出 user 中 id 键所对应的值。这里 id 键所对应的值将对应于表中的 user_id 字段。

一旦定义好之后，那么用户就可以像通常一样执行查询，根本不用关心查询是如何从 JSON 中获取数据的！

15.11 Avro Hive SerDe

Avro 是一个序列化系统，其主要特点是它是一个进化的模式驱动的二进制数据存储模式。开始的时候，Avro 的目标和 Hive 的目标可能是冲突的，因为它们都期望提供模式或者元数据信息。不过 Hive 和 Hive 元数据存储具有可插拔的设计，因此可以支持 Avro 对模式的推断。

Hive 中的 Avro SerDe 是由 LinkedIn 创建的，其提供如下功能。

- ① 从 Avro 模式中推断出对应的 Hive 表的模式。
- ② 在一个表中而不是一个指定的模式中读取所有的 Avro 文件，这得益于 Avro 的可逆兼容性。
- ③ 支持结构嵌套模式。
- ④ 可以将所有的 Avro 数据类型转换成等价的 Hive 类型。大多数的类型可以精确映射，但是一些 Avro 类型在 Hive 中并不存在的话，那么将由 Hive 通过 Avro 自动进行转换。
- ⑤ 可以解析压缩的 Avro 文件。
- ⑥ 显式地通过 Union[T, null] 语句将 Avro 中可以表示为 null 的类型转化成 T，并在合适的情况下返回 null。
- ⑦ 可以将任意的 Hive 表写成 Avro 文件。

15.11.1 使用表属性信息定义 Avro Schema

如下语句通过指定 AvroSerDe、AvroContainerInputFormat 和 AvroContainerOutputFormat 创建了一个 Arvo 表。Arvo 具有其自身的模式定义语言。这个模式定义语言可以使用属性 avro.schema.literal 以字符串的形式存储在表的属性信息中。模式中指定了 3 个字段：int 类型的 number、string 类型的 firstname 和 string 类型的 lastname。

```
CREATE TABLE doctors
ROW FORMAT
SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS
INPUTFORMAT 'org.apache.hadoop.hive ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive ql.io.avro.AvroContainerOutputFormat'
TBLPROPERTIES ('avro.schema.literal'='{
  "namespace": "testing.hive.avro.serde",
  "name": "doctors",
  "type": "record",
  "fields": [
    {
      "name": "number",
```

```

        "type":"int",
        "doc":"Order of playing the role"
    },
    {
        "name":"first_name",
        "type":"string",
        "doc":"first name of actor playing role"
    },
    {
        "name":"last_name",
        "type":"string",
        "doc":"last name of actor playing role"
    }
    ]
}');

```

当使用 DESCRIBE 命令执行查看时，Hive 就会显示出这 3 个字段的名称和类型。从下面的输出信息中我们可以看到输出信息中的第 3 列显示字段是来自反序列化的。这表明是 SerDe 自己从字段中返回的信息，而不是存在于元数据存储中的静态数据：

```

hive> DESCRIBE doctors;
number          int          from deserializer
first_name      string       from deserializer
last_name       string       from deserializer

```

15.11.2 从指定 URL 中定义 Schema

同样可以以 URL 的方式提供模式。可以是 HDFS 中某个文件的路径或者是指向 HTTP 服务器的一个链接。通常可以在表属性中设置 avro.schema.url 的值，同时不设置 avro.schema.literal 的值。

模式可以是 HDFS 中的一个文件：

```
TBLPROPERTIES ('avro.schema.url'='hdfs://hadoop:9020/path/to.schema')
```

模式同样可以存储在 HTTP 服务器上：

```
TBLPROPERTIES ('avro.schema.url'='http://site.com/path/to.schema')
```

15.11.3 进化的模式

随着时间的推移可能会为数据集增加字段或者从数据集中减少一些字段。Avro 的设计中有考虑到这个情况。进化的模式是值随着时间而发生改变的。Avro 允许字段是 null。其同样允许如果数据文件中没有定义字段的话就返回指定的缺省值。

例如，如果 Avro 模式发生了改变，并增加了一个字段，那么在 default 字段中会提供一个值，如果没有找到这个新增列就返回这个值：

```

{
  "name": "extra_field",
  "type": "string",
  "doc": "an extra field not in the original file",
  "default": "fishfingers and custard"
}

```

15.12 二进制输出

这里有多种二进制输出。我们已经看过文件压缩、sequence 文件（压缩的或没有压缩的）以及相关的文件类型。

有时，很有必要读取和写入字节流。例如，用户可能有工具期望获取字节流，而无需任何类型的字段分隔符，而且用户既希望使用 Hive 来为这些工具生成合适的文件，又希望通过 Hive 来查询这些文件。用户可能也希望使用简洁的二进制格式存储数值而不是使用像“5034223,”这样的需要更多存储空间的字符串来进行存储。一个常用的例子就是查询 `tcpdump` 命令的输出来分析网络行为。

下面的表期望自己的文件是文本文件格式的，但是其可以将查询结果按二进制数据流方式进行写操作：

```

CREATE TABLE binary_table (num1 INT, num2 INT)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES ('serialization.last.column.takes.rest'='true')
STORED AS
INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveBinaryOutputFormat';

```

下面是个 `SELECT TRANSFORM` 查询的例子，其从表 `src` 中读取二进制数据流，然后将数据流传递给 shell 命令 `cat`，最后将结果覆盖，写入到表 `destination1` 中：

```

INSERT OVERWRITE TABLE destination1
SELECT TRANSFORM(*)
USING 'cat' AS mydata STRING
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES ('serialization.last.column.takes.rest'='true')
RECORDREADER 'org.apache.hadoop.hive.ql.exec.BinaryRecordReader'
FROM src;

```

Hive 的 Thrift 服务

Hive 具有一个可选的组件叫做 HiveServer 或者 HiveThrift，其允许通过指定端口访问 Hive。Thrift 是一个软件框架，其用于跨语言的服务开发。关于 Thrift，可以通过链接 <http://thrift.apache.org/> 获得更详细的介绍。Thrift 允许客户端使用包括 Java、C++、Ruby 和其他很多种语言，通过编程的方式远程访问 Hive。

访问 Hive 的最常用的方式就是通过 CLI 进行访问。不过，CLI 的设计使其不便于通过编程的方式进行访问。CLI 是胖客户端，其需要本地具有所有的 Hive 组件，包括配置，同时还需要一个 Hadoop 客户端及其配置。同时，其可作为 HDFS 客户端、MapReduce 客户端以及 JDBC 客户端（用于访问元数据库）进行使用。即使客户端安装是成功的，但是配置所有的网络访问还是会比较困难的，特别是对于不同网段或跨数据中心的情况。

16.1 启动 Thrift Server

如果想启用 HiveServer，可以在后台启动执行这个 Hive 服务：

```
$ cd $HIVE_HOME
$ bin/hive --service hiveserver &
Starting Hive Thrift Server
```

检查 HiveServer 是否启动成功的最快捷方法就是使用 netstat 命令查看 10,000 端口是否打开并监听连接：

```
$ netstat -nl | grep 10000
tcp 0 0 :::10000 :::* LISTEN
```

（为排版方便，上面例子中去除掉了一些空格。）正如前面所提到过的，HiveServer 使用 Thrift 提供服务。Thrift 提供了一个接口语言。通过这些接口，Thrift 编译器可以产生创建网络 RPC 的多种程序语言的客户端的代码。因为 Hive 是使用 Java 编写的，而

且 Java 的字节码是跨平台的，Thrift 服务的客户端包含在了 Hive 发行版中。使用这些客户端的方式就是在 IDE 中开启一个 Java 工程，然后载入这些库，或者通过 Maven 获取这些库。

16.2 配置 Groovy 使用 HiveServer

为了讲解这个例子，我们将先对使用到 Groovy 进行介绍。Groovy 是一个对于 Java 虚拟机来说敏捷的动态的程序语言。Groovy 是进行原型设计的理想选择，因为其很好地兼容 Java，并提供了一个所见即所得的（REPL）的编码过程：

```
$ curl -o http://dist.groovy.codehaus.org/distributions/groovy-binary-1.8.6.zip
$ unzip groovy-binary-1.8.6.zip
```

下载好压缩包后，通过修改 `groovy-starter.conf` 文件，将所有的 Hive JAR 文件加入到 Groovy 的 `classpath` 路径中。这将允许 Groovy 直接和 Hive 进行交互而无需人为地在每个会话中载入相关的 JAR 文件。

```
# load required libraries
load !{groovy.home}/lib/*.jar

# load user specific libraries
load !{user.home}/.groovy/lib/*.jar

# tools.jar for ant tasks
load ${tools.jar}

load /home/edward/hadoop/hadoop-0.20.2_local/*.jar
load /home/edward/hadoop/hadoop-0.20.2_local/lib/*.jar
load /home/edward/hive-0.9.0/lib/*.jar
```



提示

Groovy 有一个 `@grab` 标注，用于从 Maven 网站库中获取到 JAR 文件，但是到目前为止 Hive 的一些打包问题使得这种方式无法正常工作。

Groovy 在分支版本中提供了 `bin/groovysh` 这个 shell 脚本。这个脚本提供了一个 REPL 用于交互式编程。Groovy 代码和 Java 代码类似，尽管其确实有其他如终止功能的形式。大多数情况下，用户可以像写 Java 一样编写 Groovy 代码。

16.3 连接到 HiveServer

从这个 REPL 中，导入 Hive 和 Thrift 相关的类。这些类用于连接到 Hive，及创建一个

HiveClient 实例。HiveClient 含有用户和 Hive 交互所需要的常用方法：

```
$ $HOME/groovy/groovy-1.8.0/bin/groovysh
Groovy Shell (1.8.0, JVM: 1.6.0_23)
Type 'help' or '\h' for help.
groovy:000> import org.apache.hadoop.hive.service.*;
groovy:000> import org.apache.thrift.protocol.*;
groovy:000> import org.apache.thrift.transport.*;
groovy:000> transport = new TSocket("localhost" , 10000);
groovy:000> protocol = new TBinaryProtocol(transport);
groovy:000> client = new HiveClient(protocol);
groovy:000> transport.open();
groovy:000> client.execute("show tables");
```

16.4 获取集群状态信息

这个 `getClusterStatus` 方法会从 Hadoop 的 JobTracker 中获取信息。其可用于获取收集监控信息也可用于寻找空闲时间来提交任务：

```
groovy:000> client.getClusterStatus()
==> HiveClusterStatus(taskTrackers:50, mapTasks:52, reduceTasks:40,
maxMapTasks:480, maxReduceTasks:240, state:RUNNING)
```

16.5 结果集模式

在执行一个查询后，用户可以通过 `getSchema()` 方法获取到结果集的 `schema`。如果用户在执行一个查询前调用这个方法的话，那么其将返回一个 `null schema`：

```
groovy:000> client.getSchema()
==> Schema(fieldSchemas:null, properties:null)
groovy:000> client.execute("show tables");
==> null
groovy:000> client.getSchema()
==> Schema(fieldSchemas:[FieldSchema(name:tab_name, type:string,
comment:from deserializer)], properties:null)
```

16.6 获取结果

当执行查询后，用户可以通过 `fetchOne()` 方法获取到结果集。并不建议通过 Thrift 接口来获取数据量大的结果。不过，其确实提供了几种方法来通过单向指针获取数据。其中 `fetchOne()` 方法用于获取一行数据：

```
groovy:000> client.fetchOne()
==> cookjar_small
```

除了一次获取一行数据这种方式外,还可以通过 `fetchALL()`方法以字符数组的形式获取整个结果集:

```
groovy:000> client.fetchAll()
==> [macetest, missing_final, one, time_to_serve, two]
```

同时还提供了 `fetchN` 方法,其用于一次获取 N 行结果。

16.7 获取执行计划

执行过一个查询之后,就可以使用 `getQueryPlan()`方法来获取关于这个查询的状态信息。信息内容包括计数器的信息和 `job` 的状态信息:

```
groovy:000> client.execute("SELECT * FROM time_to_serve");
==> null
groovy:000> client.getQueryPlan()
==> QueryPlan(queries:[Query(queryId:hadoop_20120218180808_...-aedef367
ea2f3,
  queryType:null, queryAttributes:{queryString=SELECT * FROM
time_to_serve},
  queryCounters:null, stageGraph:Graph(nodeType:STAGE, roots:null,
adjacencyList:null), stageList:null, done:true, started:true)],
done:false, started:false)
```

(这里省略了一长串的数字。)

16.8 元数据存储方法

Hive 通过 Thrift 提供 Hive 元数据存储的服务。通常来说,用户应该不能够直接调用元数据存储方法来直接对元数据进行修改,而应该通过 HiveQL 语言让 Hive 来执行这样的操作。用户应该只能通过利用只读方法来获取表的元数据信息。例如,可以使用 `get_partition_names(String, String, short)`方法来确认查询可以使用的分区有哪些:

```
groovy:000> client.get_partition_names("default", "fracture_act", (short)0)
[ hit_date=20120218/mid=001839, hit_date=20120218/mid=001842,
hit_date=20120218/mid=001846 ]
```

重要的是,要记住,尽管元数据存储 API 的变化相对是比较稳定的,但是,包括它们的签名和目的,在不同的发行版中都可能变化。Hive 试图在 HiveQL 语言中保持兼容性,这样对于这些级别的变化就会掩盖掉了。

表检查器例子

以编程的方式来访问元数据存储的能力提供了部署过程中的监控和执行条件的能力。例如，可以通过编写一个检查器来确定是否所有的表都使用了压缩，或者所有以 `zz` 开头的表保存时间不能超过 10 天。如果有必要的话，这些小的“Hive 小程序”可以快速编写完成并可以远程执行。

查找非外部表的表

默认情况下，管理表会在数据仓库目录下存储表数据，这个目录通常默认为 `/user/hive/warehouse`。而通常情况下，外部表不会使用这个目录，但是也并非就不能使用这个目录存储外部表数据。通过增加一个规则，要求管理表必须存放数据到数据仓库目录下将会有利于保持环境的清晰。

在接下来的这个程序中，最外层的循环对 `get_all_databases()` 方法的返回结果列表进行迭代。而内层的循环对 `get_all_tables(database)` 方法的结果列表进行迭代。方法 `get_table(data base, table)` 的返回结果 `Table` 对象包含了这个表的所有元数据信息。我们可以检查表的存储路径和表的类型是否是 `MANAGED_TABLE`。外部表的表类型是 `EXTERNAL`。最终会返回一组“不符合规则的”表名，用列表 `bad` 表示：

```
public List<String> check(){
    List<String> bad = new ArrayList<String>();
    for (String database: client.get_all_databases() ){
        for (String table: client.get_all_tables(database) ){
            try {
                Table t = client.get_table(database,table);
                URI u = new URI(t.getSd().getLocation());
                if ( t.getTableType().equals("MANAGED_TABLE") &&
                    ! u.getPath().contains("/user/hive/warehouse") ){
                    System.out.println(t.getTableName()
                        + " is a non external table mounted inside /user/hive/
warehouse" );
                    bad.add(t.getTableName());
                }
            } catch (Exception ex){
                System.err.println("Had exception but will continue " +ex);
            }
        }
    }
    return bad;
}
```

16.9 管理 HiveServer

Hive CLI 会在本地创建如文件名为 `.hivehistory` 这样的部件，以及会在 `/tmp` 目录和

hadoop.tmp.dir 目录下创建一些条目。因为 HiveServer 成为了 Hadoop job 开启执行的地方，所以在部署的时候需要注意几个注意事项。

16.9.1 生产环境使用 HiveServer

除了在本机安装完整的 Hive 客户端，还可以通过 HiveServer 服务来提交 job。不过如果在生产环境下使用 HiveServer 的话，那么确实还需要解决一些额外的问题。通常客户端机器需要进行的形成执行计划和管理 task 的工作现在需要由服务端来完成了。如果用户同时执行多个客户端的话，那么就会对于单个 HiveServer 造成太大的压力。一种简单的解决办法就是使用 TCP 负载均衡或者通过代理的方式为一组后面的服务器进行均衡连接。

有多种方式来进行 TCP 负载均衡，用户需要咨询下网络管理员寻求最好的解决方案。我们建议通过 haproxy 工具来对众多的 ThriftServer 服务器均衡连接。

首先，需要列举出所有的物理 ThriftServer 服务器，并记录代理对应的虚拟服务器（见表 16-1 和表 16-2）。

表 16-1 物理服务器配置

短域名	主机名和端口
HiveService1	hiveserver1.example.pvt:10000
HiveService2	hiveserver2.example.pvt:10000

表 16-2 代理配置

主机名	IP 地址
hiveprimary.example.pvt	10.10.10.100

安装 ha-proxy 软件(也就是 HAP)。对于不同的操作系统和分发版本，安装步骤可能有所不同。下面的例子中展示的是 RHEL/CENTOS 分发版的安装命令：

```
$sudo yum install haproxy
```

按照前面的清单来完成配置文件：

```
$ more /etc/haproxy/haproxy.cfg
listen hiveprimary 10.10.10.100:10000
balance leastconn
mode tcp
server hivethrift1 hiveservice1.example.pvt:10000 check
server hivethrift2 hiveservice1.example.pvt:10000 check
```

通过系统的 init 脚本来启动 HAP。一旦确认可以正常执行后，那么就可以使用 chkconfig 命令将其加入到默认的系统启动过程中：

```
$ sudo /etc/init.d/haproxy start
$ sudo chkconfig haproxy on
```

16.9.2 清理

Hive 提供了配置变量 `hive.start.cleanup.scratchdir`，默认值是 `false`。将这个属性设置为 `true` 的话，那么就会在每次重启 `HiveServer` 服务时清理掉临时目录：

```
<property>
  <name>hive.start.cleanup.scratchdir</name>
  <value>true</value>
  <description>To clean up the Hive scratchdir while
  starting the Hive server</description>
</property>
```

16.10 Hive ThriftMetastore

典型情况下，Hive 会话会直接连接到一个 JDBC 数据库，这个数据库用作元数据存储数据库。Hive 提供了一个可选的组件名为 `ThriftMetastore`。在这种设置下，Hive 客户端会连接到 `ThriftMetastore`，而且会和 `JDBCMetastore` 进行通信。大多数的部署是不需要这个组件的。对于那些非 Java 客户端而又需要获取到元数据存储信息时才会使用这个组件。使用这种元数据库服务需要 2 个单独的配置。

16.10.1 ThriftMetastore 配置

`ThriftMetastore` 应该和实际使用 JDBC 的元数据存储进行通信，然后再通过如下方法启动 `metastore`：

```
$ cd ~
$ bin/hive --service metastore &
[1] 17096
Starting Hive Metastore Server
```

使用 `netstat` 命令来确认元数据服务启动并正在执行：

```
$ netstat -an | grep 9083
tcp 0 0 :::9083 :::* LISTEN
```

16.10.2 客户端配置

像 CLI 这样的客户端需要直接和元数据存储通信：

```
<property>
  <name>hive.metastore.local</name>
  <value>>false</value>
  <description>controls whether to connect to remove metastore server
  or open a new metastore server in Hive Client JVM</description>
```

```
</property>

<property>
  <name>hive.metastore.uris</name>
  <value>thrift://metastore_server:9083</value>
  <description>controls whether to connect to remove metastore server
  or open a new metastore server in Hive Client JVM</description>
</property>
```

这个变化对于用户来说是透明的。不过,对于 Hadoop 安全来说这里还是有细微的差别,而且需要以用户的身份信息来执行元数据存储相关的操作。

存储处理程序和 NoSQL

存储处理程序是一个结合 `InputFormat`、`OutputFormat`、`SerDe` 和 Hive 需要使用的特定的代码，来将外部实体作为标准的 Hive 表进行处理的整体。这样无论表是以文本文件的方式存储在 Hadoop 中，还是以列族的方式存储在如 *Apache HBase*、*Apache Cassandra* 和 *Amazon DynamoDB* 这样的 NoSQL 数据库中，用户都可以无缝地直接执行查询，解决问题。存储处理程序不仅限于 NoSQL 数据库，可以为多种类别的数据存储设计一个数据存储处理程序。



提示

对于特定的存储处理程序可能只需实现其中的某些功能。例如，一个给定的存储处理程序可能允许只读访问权限或实施其他一些限制。

存储处理程序提供了一个 ETL 的简化系统。例如，一个 Hive 查询可以从一个 `sequence file` 存储格式的表中选取数据，也可以进行输出。

17.1 Storage Handler Background

Hadoop 中有一个名为 `InputFormat` 的抽象接口类，其可以将来自不同源的数据格式化成为可以作为 job 输入的格式。`TextInputFormat` 就是 `InputFormat` 的一个具体实现。其会提供给 Hadoop 关于如何将给定文件路径下的文件分割成多个划分，然后分发给多个 task 进行处理的信息，而且还提供了一个 `RecordReader` 接口，该接口提供了用于从每个划分中读取数据的方法。

Hadoop 同是还提供了一个名为 `OutputFormat` 的抽象接口，其会获取到一个 job 的输出，然后将这个输出输入到一个实体中。`TextOutputFormat` 就是 `OutputFormat` 的一个具体实现。其可以持续地将结果输入到一个存储在 HDFS 或本地文件系统中的文

件中。

在 Hadoop 中输入和输出是物理文件的情况很正常，不过 InputFormat 和 OutputFormat 抽象接口可被用于从其他数据源（包括关系型数据库、NoSQL 存储（如 Cassandra）或 HBase，以及其他任何可通过 InputFormat 或者 OutputFormat 进行设计实现的存储）中读取和存放数据。

在“HiveQL”那一章中，我们展示了使用 Java 代码编写的 Word Count 的例子，然后展示了在 Hive 中等价的实现方式。Hive 中的抽象如表、类型、行格式以及其他元数据都用于 Hive 理解源数据。一旦 Hive 了解了源数据的描述信息，那么查询引擎就可以使用熟悉的 HiveQL 操作符来处理数据。

很多的 NoSQL 数据库都使用传统适配器实现了 Hive 连接器。

17.2 HiveStorageHandler

HiveStorageHandler 是 Hive 用于连接如 HBase、Cassandra 等类似的 NoSQL 存储的主要接口。检查下接口可以发现需要定义一个定制的 InputFormat、一个 OutputFormat 以及 SerDe。存储处理程序负责从底层存储子系统中读取或写入数据。这就转变成通过写 SELECT 查询读取数据系统中的数据，以及通过如 reports 这样的操作将数据写入到数据系统。

当在 NoSQL 数据库之上执行 Hive 查询时，NoSQL 系统的资源消耗、执行效率要比常规的基于 HDFS 的 Hive 和 MapReduce job 要慢。其中一部分原因源于服务器的 socket 连接资源消耗和对底层多个文件的合并过程，而从 HDFS 中典型的访问是完全顺序 I/O，顺序 I/O 在现代磁盘上是非常快的。

在一个系统整体架构中将 NoSQL 数据库和 Hadoop 结合使用的一个通用技术就是使用 NoSQL 数据库集群来进行实时处理工作，而利用 Hadoop 集群进行非实时面向批处理的工作。如果 NoSQL 系统是主数据存储，而其数据又需要通过 Hadoop 的批处理 job 进行查询的话，批量导出是一个将 NoSQL 数据导入到 HDFS 文件中的高效的方式。一旦 HDFS 中的文件通过导出生成后，就可以以最大效率执行批处理 Hadoop job。

17.3 HBase

如下例子展示了如何使用 HiveQL 创建一个指向 HBase 表的 Hive 表：

```
CREATE TABLE hbase_stocks(key INT, name STRING, price FLOAT)
  STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
  WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,stock:val")
  TBLPROPERTIES ("hbase.table.name" = "stocks");
```

如果想创建一个指向一个已经存在的 HBase 表的 Hive 表的话，那么就必须使用 CREATE EXTERNAL TABLE 这个 HiveQL 语句：

```
CREATE EXTERNAL TABLE hbase_stocks(key INT, name STRING, price FLOAT)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = "cf1:val")
TBLPROPERTIES ("hbase.table.name" = "stocks");
```

对于一个指定的 Hive 查询，不需要扫描整个 HBase 表，通过过滤下推裁剪将会得到返回给 Hive 的行数据。

可以进行谓词下推的类型如下。

- ① key < 20。
- ② key = 20。
- ③ key < 20，而且 key > 10。

其他的复杂类型的谓词都会忽略掉而不会启用下推功能。

下面是一个简单的例子，其先创建了一个简单的表，然后对这个表进行的查询语句将会使用到过滤条件下推的功能。需要注意的是下推的总是 HBase 的键，而非列组中的列值。

```
CREATE TABLE hbase_pushdown(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf:string");

SELECT * FROM hbase_pushdown WHERE key = 90;
```

如下这个查询不会查询下推，因为其过滤条件中包含有 OR 操作符。

```
SELECT * FROM hbase_pushdown
WHERE key <= '80' OR key >= '100';
```

和 HBase 结合使用的 Hive 支持 HBase 表和非 HBase 表的连接操作，也支持 HBase 表和非 HBase 表的连接操作。

默认情况下，下推优化是开启的，不过可以通过如下命令将此功能关闭：

```
set hive.optimize.ppd.storage=false;
```

当将 Hive 中的数据导入到 HBase 中时，需要注意，HBase 要求键是排重后唯一的，而 Hive 并无此要求。

下面是一些 Hive 和 HBase 列映射需要注意的问题。

- ① 没有访问 HBase 行时间戳的方式，只会返回最新版本的行。

② HBase 的键必须进行显式定义。

17.4 Cassandra

Cassandra 已经采用和 HBase 中类似的实现方式实现了 HiveStorageHandler 接口。这种实现方式最先是在 Brisk 项目中的 Datastax 中使用的。

这个模式非常简单，一个 Cassandra 列族和一个 Hive 表映射起来。同样地，Cassandra 列名和 Hive 列名直接映射起来。

17.4.1 静态列映射 (Static Column Mapping)

当用户指定了 Cassandra 中的哪些列期望和 Hive 列进行映射时使用静态列存储非常有用。下面的例子中，首先创建了一个 Hive 外部表，其和一个已存在的 Cassandra 键空间及列族相映射：

```
CREATE EXTERNAL TABLE Weblog (useragent string, ipaddress string, timestamp
string)
STORED BY 'org.apache.hadoop.hive.cassandra.CassandraStorageHandler'
WITH SERDEPROPERTIES (
    "cassandra.columns.mapping" = ":key,user_agent,ip_address,time_stamp")
TBLPROPERTIES (
    "cassandra.range.size" = "200",
    "cassandra.slice.predicate.size" = "150" );
```

17.4.2 为动态列转置列映射

某些 Cassandra 使用场景会用到动态列。这种使用场景是某个指定列组没有固定的、已命名的列，而是由列的行键代表某片数据。这常用于时间序列数据，其中列名代表时间，而列值代表那个时间的值。如果列名是已知的话，或者用户需要检索所有值时，这也是很有用的。

```
CREATE EXTERNAL TABLE Weblog (useragent string, ipaddress string, timestamp
string)
STORED BY 'org.apache.hadoop.hive.cassandra.CassandraStorageHandler'
WITH SERDEPROPERTIES (
    "cassandra.columns.mapping" = ":key,:column,:value");
```

17.4.3 Cassandra SerDe Properties

表 17-1 中所介绍的属性可以在 WITH SERDEPROPERTIES 语句中进行定义。

表 17-1 Cassandra SerDe 存储控制器属性

名称	描述
cassandra.cloumns.mapping	将 Hive 的列和 Cassandra 的列映射起来
cassandra.cf.name	Cassandra 中的列族名
cassandra.host	要连接的 Cassandra 节点 IP
cassandra.port	Cassandra RPC 端口，默认是 9160
cassandra.partitioner	分区器，默认是 RandomPartitioner

表 17-2 中所介绍的属性可以在 TBLPROPERTIES 语句中进行定义。

表 17-2 Cassandra 表属性

属性名	描述
cassandra.ks.name	Cassandra 键空间名称
cassandra.ks.repfactor	Cassandra 冗余因子，默认是 1
cassandra.ks.strategy	冗余策略，默认是 SimpleStrategy
cassandra.input.split.size	MapReduce 划分大小，默认是 64*1024
cassandra.range.size	MapReduce 批处理范围大小，默认是 1000
cassandra.slice.predicate.size	MapReduce 片推测大小，默认是 1000

17.5 DynamoDB

Amazon 的 Dynamo 是最先推出的 NoSQL 数据库之一。Dynamo 的设计影响了很多其他的数据库，包括 Cassandra 和 HBase。尽管其有很大的影响力，但是 Dynamo 目前还仅限制在 Amazon 内部进行使用。Amazon 发布了另一款受 Dynamo 影响的数据库，这个数据库被称为 DynamoDB。

DynamoDB 属于键-值数据库中的一种。在 DynamoDB 中，表就是一组元素 (item) 的集合，而且其中必需要有一个主键。一个元素 (item) 包含有一个键和任意数量的属性值。不同的元素 (item) 可以具有不同的属性集。

用户可以通过 Hive 查询 DynamoDB 中的表，而且用户可以迁移出数据或者导入数据到 S3 中。下面是一个关于股票的 Hive 表的例子，这个表底层其实是一张 DynamoDB 表：

```
CREATE EXTERNAL TABLE dynamo_stocks(
  key INT, symbol STRING,
  ymd STRING, price FLOAT)
STORED BY
'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES (
```

```
"dynamodb.table.name" = "Stocks",  
"dynamodb.column.mapping" =  
  "key:Key,symbol:Symbol,  
   ymd:YMD,price_close:Close");
```

关于 DynamoDB，可以通过如下链接获取更多信息：<http://aws.amazon.com/dynamodb/>。

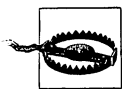
在了解 Hive 的安全机制之前，我们需要首先清楚 Hadoop 的安全机制以及 Hadoop 的历史。Hadoop 起源于 Apache Nutch 的子项目。在那个时期以及其整个早期原型时期，功能性需求比安全性需求优先级要高。分布式系统的安全问题要比正常情况下更加复杂，因为不同机器上的多个组件需要相互进行通信。

在如 v0.20.205 发行版之前的非安全 Hadoop 版本中，是通过调用本地的 `whoami` 脚本获取用户名的。用户可以自由地通过设置 `FSShell` 命令（文件系统命令）中的 `hadoop.job.ugi` 属性来改变这个参数值。Map 或者 Reduce 任务（task）都以相同的系统用户（通常用户名是 `hadoop` 或者 `mapred`）在 Task-Tracker 节点上执行任务。同样地，Hadoop 组件通常要监听数值比较大的端口。通常情况下，都是以非特权用户来执行的（例如，非 root 用户的其他用户）。

最近对于 Hadoop 的安全引入了多个变化，其中主要是对于 Kerberos 安全认证的支持，还包含其他一些问题的修复。Kerberos 允许客户端和服务端相互认证。客户端的每次请求中都会带有凭证（ticket）信息。在 TaskTracker 上执行的任务（task）都是由执行任务（job）的用户来执行的。用户无法通过设置 `hadoop.job.ugi` 属性的值来模拟其他人来执行任务。为了达到这个目的，所有的 Hadoop 组件从头到尾都要使用 Kerberos 安全认证。

Hive 在 Hadoop 引入 Kerberos 支持之前就已经存在了，而且 Hive 目前还没有完全和 Hadoop 的安全改变相融合。例如，Hive 元数据存储连接可能是直接连接到一个 JDBC 数据库或者是通过 Thrift 进行连接，这些都要使用用户的身份进行各种操作。像 `HiveService` 这样的基于 Thrift 的组件还是要冒充他人来执行。Hadoop 的文件用户权限模型（也就是对于一个文件分为用户、组和其他 3 层权限）和很多其他数据库中用户权限模型具有很大的差异，数据库中通常是使用对表行或者字段级别进行授权和权限回收操作来进行权限控制的。

本章试图强调在安全和非安全 Hadoop 版本中 Hive 的组件操作的不同点。关于 Hadoop 的安全特性的详细介绍，请参考 Tom White (O' Reilly) 所著的《Hadoop 编程指南》一书。



警告

Hadoop 中的安全支持仍然是相对比较新且处于进化中的。Hive 中有些部分还是无法和 Hadoop 的安全支持相融合的。本节所讨论的内容总结了当前 Hive 安全的状况，但这并非最终状况。

关于 Hive 安全的更多内容，可以参考 Hive 的安全 wiki 页面：<https://cwiki.apache.org/confluence/display/Hive/Security>。同时，和本书中其他章节不同的是，我们会要求用户查看更多相关的 JIRA 获取更多信息。

18.1 和 Hadoop 安全功能相结合

Hive v0.7.0 版本增加了和 Hadoop 安全功能的结合¹，这意味着，例如，当 Hive 提交任务到安全集群的 JobTracker 上时，其将使用合适的认证处理过程。用户权限可以被授予也可以被回收，这个下面我们将进行讨论。

不过在 Thrift 和其他组件中仍存在着几个已知的安全漏洞，这些在 Hive 的安全 wiki 页面上有列举出来。

18.2 使用 Hive 进行验证

如果文件和文件夹是多个用户共同拥有的话，那么文件的权限设置就变的非常重要。HDFS 中文件目录权限和 Unix 中的模式非常相似，都包含有 3 层：用户、组和其他。同时具有 3 种权限：可读、可写和可执行。Hive 中有一个配置变量 `hive.files.umask.value` 来定义对于新创建的文件设置的默认权限的 `umask` 值，也就是掩码字节数。

```
<property>
  <name>hive.files.umask.value</name>
  <value>0002</value>
  <description>The dfs.umask value for the hive created folders</description>
</property>
```

同时，当属性 `hive.metastore.authorization.storage.checks` 的值为 `true` 时，如果用户没有权限删除表底层的文件，Hive 就会阻止用户来删除这样的表。这个参数的默认值是

1. 请参考如下链接：<https://issues.apache.org/jira/browse/HIVE-1264>。

false，而其应该是设置为 true 的：

```
<property>
  <name>hive.metastore.authorization.storage.checks</name>
  <value>true</value>
  <description>Should the metastore do authorization checks against
    the underlying storage for operations like drop-partition (disallow
    the drop-partition if the user in question doesn't have permissions
    to delete the corresponding directory on the storage).</description>
</property>
```

当在安全模式下执行时，Hive 元数据存储要尽可能地将 `hive.metastore.execute.setugi` 设置为 true。

```
<property>
  <name>hive.metastore.execute.setugi</name>
  <value>>false</value>
  <description>In unsecure mode, setting this property to true will
    cause the metastore to execute DFS operations using the client's
    reported user and group permissions. Note that this property must
    be set on both the client and server sides. Further note that its
    best effort. If client sets it to true and server sets it to false,
    client setting will be ignored.</description>
</property>
```

可以在如下 JIRA 中获取更详细的信息：<https://issues.apache.org/jira/browse/HIVE-842>，“Hive 中安全任务架构”。

18.3 Hive 中的权限管理

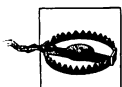
Hive v0.7.0 版本中同时增加了通过 HiveQL 进行授权设置的功能¹。默认情况下，授权模块是不开启的，需要将如下的属性设置为 true，才能开启授权：

```
<property>
  <name>hive.security.authorization.enabled</name>
  <value>true</value>
  <description>Enable or disable the hive client authorization</description>
</property>
<property>
  <name>hive.security.authorization.createtable.owner.grants</name>
  <value>ALL</value>
  <description>The privileges automatically granted to the owner whenever
    a table gets created.An example like "select,drop" will grant select
```

1.请参考 <https://issues.apache.org/jira/browse/HIVE-78>，“Hive 中的权限管理架构”以及如下链接中对于这个功能的描述：<https://cwiki.apache.org/Hive/languagemanual-auth.html>。


```
and drop privilege to the owner of the table</description>
</property>
```

默认情况下，hive.security.authorization.createtable.owner.grants 的值是 null，这使得用户无法访问自己的表。因此，我们也要给予表创建者对应的权限才能访问自己创建的表。



警告

当前用户是可以通过 set 命令来将相关的属性设置为 false 来关闭权限认证的。

18.3.1 用户、组和角色

可以对用户 (user)、组 (group) 或者角色 (role) 授予权限或者回收权限。我们将逐步演示对这些实体的授权过程。

```
hive> set hive.security.authorization.enabled=true;

hive> CREATE TABLE authorization_test (key int, value string);
Authorization failed:No privilege 'Create' found for outputs { database:default}.
Use show grant to get more details.
```

我们已经看到，我们使用的用户没有在 default 数据库下创建表的权限。我们可以对多个实体进行授权。第一个实体就是用户 (user)，Hive 中的用户就是用户的系统用户名。我们可以确定其名称，并按照如下语句将在 default 数据库下的创建表 (create) 权限授予这个用户：

```
hive> set system:user.name;
system:user.name=edward

hive> GRANT CREATE ON DATABASE default TO USER edward;

hive> CREATE TABLE authorization_test (key INT, value STRING);
```

我们可以通过 SHOW GRANT 命令查看授权结果情况：

```
hive> SHOW GRANT USER edward ON DATABASE default;

database           default
principalName      edward
principalType       USER
privilege           Create
grantTime           Mon Mar 19 09:18:10 EDT 2012
grantor edward
```

在用户很多同时表也很多的情况下，对于用户级别的权限授予将会给运维上带来高成本。一个更好的选择就是基于组 (group) 级别的权限授予。Hive 中组和用户的主 POSIX

组是等价的：

```
hive> CREATE TABLE authorization_test_group(a int,b int);

hive> SELECT * FROM authorization_test_group;
Authorization failed:No privilege 'Select' found for inputs
{ database:default, table:authorization_test_group, columnName:a}.
Use show grant to get more details.

hive> GRANT SELECT on table authorization_test_group to group edward;

hive> SELECT * FROM authorization_test_group;
OK
Time taken: 0.119 seconds
```

如果用户 (user) 和组 (group) 仍不够灵活的话，那么可以使用角色 (role)。用户可以放置在角色中同时可以为角色进行授权。角色是非常灵活的，因为和组不一样，组是由系统外部进行控制的，而角色是由 Hive 内部进行控制的：

```
hive> CREATE TABLE authentication_test_role (a int , b int);

hive> SELECT * FROM authentication_test_role;
Authorization failed:No privilege 'Select' found for inputs
{ database:default, table:authentication_test_role, columnName:a}.
Use show grant to get more details.

hive> CREATE ROLE users_who_can_select_authentication_test_role;

hive> GRANT ROLE users_who_can_select_authentication_test_role TO USER
edward;

hive> GRANT SELECT ON TABLE authentication_test_role
> TO ROLE users_who_can_select_authentication_test_role;

hive> SELECT * FROM authentication_test_role;
OK
Time taken: 0.103 seconds
```

18.3.2 Grant 和 Revoke 权限

表 18-1 列举出了可以配置的权限。

表 18-1 权限

名称	描述
ALL	赋予所有的权限
ALTER	有修改表结构的权限

名称	描述
CREATE	有创建表的权限
DROP	有删除表或表中的分区的权限
INDEX	创建表索引的权限（注意：当前并未实现）
LOCK	开启并发后，锁定和解锁定表的权限
SELECT	查询表或者分区中数据的权限
SHOW_DATABASE	查看所有数据库的权限
UPDATE	向表或者分区中插入或加载数据的权限

下面这个例子演示了如何使用 CREATE 权限：

```
hive> SET hive.security.authorization.enabled=true;

hive> CREATE DATABASE edsstuff;

hive> USE edsstuff;

hive> CREATE TABLE a (id INT);
Authorization failed:No privilege 'Create' found for outputs
{ database:edsstuff}. Use show grant to get more details.

hive> GRANT CREATE ON DATABASE edsstuff TO USER edward;

hive> CREATE TABLE a (id INT);

hive> CREATE EXTERNAL TABLE ab (id INT);
```

同样地，我们可以通过如下命名授予 ALTER 权限：

```
hive> ALTER TABLE a REPLACE COLUMNS (a int , b int);
Authorization failed:No privilege 'Alter' found for inputs
{ database:edsstuff, table:a}. Use show grant to get more details.

hive> GRANT ALTER ON TABLE a TO USER edward;

hive> ALTER TABLE a REPLACE COLUMNS (a int , b int);
```

需要注意的是，对于如下这种为分区表新增分区的操作是不需要 ALTER 权限的：

```
hive> ALTER TABLE a_part_table ADD PARTITION (b=5);
```

往表中加载数据的话需要使用 UPDATE 权限：

```
hive> LOAD DATA INPATH '${env:HIVE_HOME}/NOTICE'
> INTO TABLE a_part_table PARTITION (b=5);
```

```
Authorization failed:No privilege 'Update' found for outputs
{ database:edsstuff, table:a_part_table}. Use show grant to get more
details.
```

```
hive> GRANT UPDATE ON TABLE a_part_table TO USER edward;

hive> LOAD DATA INPATH '${env:HIVE_HOME}/NOTICE'
> INTO TABLE a_part_table PARTITION (b=5);
Loading data to table edsstuff.a_part_table partition (b=5)
```

删除表或者分区需要 DROP 权限:

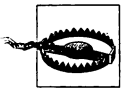
```
hive> ALTER TABLE a_part_table DROP PARTITION (b=5);
Authorization failed:No privilege 'Drop' found for inputs
{ database:edsstuff, table:a_part_table}. Use show grant to get more
details.
```

从表或者分区中查询数据的话需要 SELECT 权限:

```
hive> SELECT id FROM a_part_table;
Authorization failed:No privilege 'Select' found for inputs
{ database:edsstuff, table:a_part_table, columnName:id}. Use show
grant to get more details.
```

```
hive> GRANT SELECT ON TABLE a_part_table TO USER edward;
```

```
hive> SELECT id FROM a_part_table;
```



警告

目前 GRANT SELECT(COLUMN)这个操作语法是通过的, 不过实际不会执行。

用户同样可以通过如下命令授予全部的权限:

```
hive> GRANT ALL ON TABLE a_part_table TO USER edward;
```

18.4 分区级别的权限

Hive 中分区表非常常见。默认情况下, 是在表级别授予权限的。不过, 同样可以在分区级别进行权限授予。为达到这个目标, 只需要将表属性 PARTITION_LEVEL_PRIVILEGE 设置为 TRUE 即可:

```
hive> CREATE TABLE authorize_part (key INT, value STRING)
> PARTITIONED BY (ds STRING);

hive> ALTER TABLE authorization_part
> SET TBLPROPERTIES ("PARTITION_LEVEL_PRIVILEGE"="TRUE");
Authorization failed:No privilege 'Alter' found for inputs
```

```

(database:default, table:authorization_part}.
Use show grant to get more details.

hive> GRANT ALTER ON table authorization_part to user edward;

hive> ALTER TABLE authorization_part
  > SET TBLPROPERTIES ("PARTITION_LEVEL_PRIVILEGE"="TRUE");

hive> GRANT SELECT ON TABLE authorization_part TO USER edward;

hive> ALTER TABLE authorization_part ADD PARTITION (ds='3');

hive> ALTER TABLE authorization_part ADD PARTITION (ds='4');

hive> SELECT * FROM authorization_part WHERE ds='3';

hive> REVOKE SELECT ON TABLE authorization_part partition (ds='3') FROM
USER edward;

hive> SELECT * FROM authorization_part WHERE ds='3';
Authorization failed:No privilege 'Select' found for inputs
{ database:default, table:authorization_part, partitionName:ds=3,
columnName:key}.
Use show grant to get more details.

hive> SELECT * FROM authorization_part WHERE ds='4';
OK
Time taken: 0.146 seconds

```

18.5 自动授权

用户经常会期望创建表后不再执行烦人的授权命令，就可以具有相关的权限，而直接去执行后续的查询，等等。早期，用户可能需要具有 ALL 权限才可以，不过现在可以为默认情况指定更细节的权限。

属性 `hive.security.authorization.createtable.owner.grants` 中可以定义为创建表的用户自动授予对这张表的指定的权限。在下面的这个例子中，并不是授予 ALL 权限，而是为用户自动授予对其所创建的表的 SELECT 和 DROP 权限：

```

<property>
  <name>hive.security.authorization.createtable.owner.grants</name>
  <value>select,drop</value>
</property>

```

类似地，可以在创建表时自动授予指定用户指定的权限。属性 `hive.security.authorization.createtable.user.grants` 就控制着这个行为。下面这个例子展示了 Hive 管理员账号 `admin1`

和用户 `edward` 默认授予所有表的读权限，而 `user1` 只有创建表的权限。

```
<property>
  <name>hive.security.authorization.createtable.user.grants</name>
  <value>admin1,edward:select;user1:create</value>
</property>
```

对于组(`group`)和角色 (`role`) 同样具有类似的属性来控制自动授予的权限。对于组，该属性名为 `hive.security.authorization.createtable.group.grants`；对于角色，这个属性是 `hive.security.authorization.createtable.role.grants`。这些属性的值的形式和前面介绍的是相同的。

尽管 HiveQL 是一种 SQL 方言，但是 Hive 缺少通常在 update 和 insert 类型的查询中使用的对于列、行或者查询级别的锁支持。Hadoop 中的文件通常是一次写入的（尽管 Hadoop 确实支持有限的文件追加功能）。因为这个一次写入的天性和 MapReduce 的 streaming 类型，访问细粒度锁是不必要的。

不过，既然 Hadoop 和 Hive 是多用户系统，那么在一些情况下，锁和协调会是非常有用的。例如，如果一个用户期望锁定一个表，因为使用 INSERT OVERWRITE 这样的查询就可以修改表的内容，而同时第 2 个用户也尝试使用这个表解决某个查询问题，这样的查询可能会失败或者产生无效的结果。

Hive 可以被认为是一个胖客户端，因为在某种意义上每个 Hive CLI、Thrift server 或者 Web 接口实例都不是完全独立于其他实例的。因为这个独立性，所有锁必须由单独的系统进行协调。

19.1 Hive 结合 Zookeeper 支持锁功能

Hive 中包含了一个使用 Apache Zookeeper 进行锁定的锁功能。Zookeeper 实现了高度可靠的分布式协调功能。处理需要增加一些额外的设置和配置步骤。Zookeeper 对于 Hive 用户来说是透明的。

设置 Zookeeper，需要为其指定一个或者多个服务器来执行它的服务进程。典型的最小配置需要具有 3 个 Zookeeper 节点，这样就可以提供一个群体，并保持足够的冗余。

在下一个示例中，我们将使用 3 个节点：zk1.site.pvt、zk2.site.pvt 和 zk3.site.pvt。

下载并解压缩一个 Zookeeper 发行包。通过如下一系列命令，我们就可以将 Zookeeper 安装到/opt 目录下，安装过程需要 sudo 访问权限（对于 Zookeeper 之后的更新的版本，

如果有如下命令的话，差不多也是可以通过如下方式安装的)：

```
$ cd /opt
$ sudo curl -o http://www.ecoficial.com/am/zookeeper/stable/zookeeper-3.3.3.tar.gz
$ sudo tar -xf zookeeper-3.3.3.tar.gz
$ sudo ln -s zookeeper-3.3.3 zookeeper
```

创建一个目录用来存放 Zookeeper 的数据：

```
$ sudo mkdir /var/zookeeper
```

创建一个 Zookeeper 配置文件 /opt/zookeeper/conf/zoo.cfg。这个配置文件中的内容可以参考下面的信息，当然需要根据用户的安装情况适当进行修改：

```
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zk1.site.pvt:2888:3888
server.2=zk2.site.pvt:2888:3888
server.3=zk3.site.pvt:2888:3888
```

在每台服务器上，都需要创建一个 myid 文件，并确保文件内容和配置文件中配置的 ID 匹配。例如，对于 zk1.site.pvt 这个节点服务器上的 myid 文件，用户可以通过如下命令进行创建：

```
$ sudo echo 1 > /var/zookeeper/myid
```

最后，通过如下命令启动 Zookeeper：

```
$ sudo /opt/zookeeper/bin/zkServer.sh start
```



提示

用户需要使用 root 用户来启动这个进程，虽然对于大多数的进程都无需使用 root 用户启动。用户可以以通常的标准技术使用其他的用户来执行这个文件。

一旦 Zookeeper 节点间相互通信，可能就会在一台 Zookeeper 节点上创建数据，而从另一个节点上读取数据。例如，在某个节点上执行如下这个会话：

```
$ /opt/zookeeper/bin/zkCli.sh -server zk1.site.pvt:2181
[zk: zk1.site.pvt:2181(CONNECTED) 3] ls /
[zookeeper]
[zk: zk1.site.pvt:2181(CONNECTED) 4] create /zk_test my_data
Created /zk_test
```


然后，从其他某个节点上执行这个会话，或者在之前的那个节点上重新开一个控制台来执行这个会话：

```
$ /opt/zookeeper/bin/zkCli.sh -server zk1.site.pvt:2181
[zk: zk1.site.pvt:2181(CONNECTED) 0] ls /
[zookeeper, zk_test]
[zk: zk1.site.pvt:2181(CONNECTED) 1]
```

嗨！好吧，最艰难的部分已经过去了。现在我们需要配置 Hive，让其可以使用这些 Zookeeper 节点来启用并发支持。

在\$HIVE_HOME/hive-site.xml 这个配置文件中，需要增加如下属性配置：

```
<property>
  <name>hive.zookeeper.quorum</name>
  <value>zk1.site.pvt,zk1.site.pvt,zk1.site.pvt</value>
  <description>The list of zookeeper servers to talk to.
  This is only needed for read/write locks.</description>
</property>

<property>
  <name>hive.support.concurrency</name>
  <value>>true</value>
  <description>Whether Hive supports concurrency or not.
  A Zookeeper instance must be up and running for the default
  Hive lock manager to support read-write locks.</description>
</property>
```

配置好这些属性后，Hive 会对特定的查询自动启动获取锁，用户可以通过 SHOW LOCKS 命令查看当前的所有锁：

```
hive> SHOW LOCKS;
default@people_20111230 SHARED
default@places SHARED
default@places@hit_date=20111230 SHARED
...
```

假设这里的 places 是分区表，如下这些更加集中的查询也是支持的，其中的省略号可以换成一个适当的分区描述。

```
hive> SHOW LOCKS places EXTENDED;
default@places SHARED
...
hive> SHOW LOCKS places PARTITION (...);
default@places SHARED
...
hive> SHOW LOCKS places PARTITION (...) EXTENDED;
default@places SHARED
...
```

Hive 中提供了 2 种类型的锁，开启并发功能后，它们也就自动被启用了。某个表被读取的时候需要使用共享锁。多重并发共享锁也是允许使用的。

对于其他那些会以某种方式修改表的操作都是需要使用独占锁的。其不仅会冻结其他表修改操作，同时也会阻止其他进程进行查询。

当表是分区表时，对一个分区获取独占锁时会导致需要对表本身获取共享锁来防止发生不相容的变更，例如当表的一个分区正在被修改的时候尝试删除这个表。当然，对表使用独占锁会全局影响所有的分区。

19.2 显式锁和独占锁

用户同样可以显式地管理锁。例如，假设某个 Hive 会话对表 `people` 创建了一个显式锁：

```
hive> LOCK TABLE people EXCLUSIVE;
```

下面是另一个 Hive 会话，其尝试查询这个被锁定的表：

```
hive> SELECT COUNT(*) FROM people;
conflicting lock present for default@people mode SHARED
FAILED: Error in acquiring locks: locks on the underlying objects
cannot be acquired. retry after some time
```

通过 `UNLOCK TABLE` 语句可以对表进行解锁，解锁后其他会话的查询就可以正常工作了：

```
hive> UNLOCK TABLE people;
```

Hive 和 Oozie 整合

Apache Oozie 是一个工作流引擎服务器,其用于运行 Hadoop Map/Reduce 和 Pig 任务工作流,官网地址是:<http://incubator.apache.org/oozie/>。Hive 本身也具有一个内置的工作流系统。Hive 可以将一个查询转化成一個或者多个执行过程,例如 map/reduce 执行过程或者一个 move 转移文件的执行过程。如果其中某个过程失败了,Hive 就会清理这个进程并报告错误信息。如果其中某个执行过程成功了,Hive 就会执行下一个子过程直到整个 job 执行成功。同时,可以在一个 HQL 文件中放置多个语句,然后 Hive 会按照次序执行这些查询,直到文件中所有的语句都执行成功为止。

Hive 中的工作流控制系统对于处理单个任务或者处理按照次序执行的多个任务的效果是非常好的。不过,一些工作流要比之复杂得多。例如,用户可能需要这样一个处理过程,其第 1 步是一个传统的 MapReduce 任务,而第 2 步需要使用到第 1 步的输出作为输入,然后使用 Hive 进行处理,而最后 1 步是使用 distcp 命令将第 2 步的输出结果传递到一个远程集群中。这些形式的工作流对于 Oozie 工作流来说都是可以进行处理。

Oozie 工作流任务是一系列“动作”的有向无环图 (DAG)。一些工作流是根据需要运行触发的,但大多数情况下,我们有必要基于一定的时间段和(或)数据可用性和(或)外部事件来运行它们。Oozie 协调系统让用户可以基于这些参数来定义工作流执行计划。Oozie 一个重要的特点就是工作流的状态是和发起任务的客户端分离开来的。这种分离式的(发起,然后不再管理)任务启动方式是非常有用的。通常一个 Hive 任务是和提交这个任务的控制台联系在一起的。如果控制台中断了,那么其任务也就完成了一半了。

20.1 Oozie 提供的多种动作 (Action)

Oozie 提供了多个内置的动作。如下列举了其中的一些,并进行了简要的描述。

1. MapReduce

用户提供 Mapper 类和 Reducer 类，并设置一些附属变量。

2. Shell

包含参数的 Shell 命令也可作为动作来执行。

3. Java 动作

含有 main 方法的 Java 类，可以指定参数后执行。

4. Pig

可以执行 Pig 脚本。

5. Hive

可以执行 Hive 的 HQL 查询。

6. DistCp

可以执行 DistCp 命令，将数据拷贝到另一个 HDFS 集群，或从另一个 HDFS 集群拷贝数据到当前集群。

Hive Thrift Service Action

Hive 内置的动作可以很好地工作，但是其有一些缺点。它们会将 Hive 作为胖客户端来使用。Hive 分支中的大部分，包括 JAR 文件和配置文件，都需要拷贝到工作流目录下。当 Oozie 启动一个动作时，其就会随机地选择一个 TaskTracker 节点进行启动。如果用户设置元数据存储只能从指定的主机进行连接的话，那么就可能存在连接元数据存储的问题。因为如果任务失败了，Hive 会遗留下如 *hive-history* 或一些存放在 */tmp* 目录下的东西，所以需要确保清理所有 TaskTracker 上面的这些信息。

通过使用 Hive Thrift Service（参见第 16 章内容），（基本）可以解决胖客户端问题。其中 HiveServiceBAction（表示“Hive 服务 B 计划行动方案”）使得可以通过 Hive Thrift Service 提交任务（job）。这样有一个好处，即可以将所有的 Hive 操作汇集到一个预先定义的执行 Hive 服务的节点集合上。

```
$ cd ~
$ git clone git://github.com/edwardcapriolo/hive_test.git
$ cd hive_test
$ mvn wagon:download-single
$ mvn exec:exec
$ mvn install

$ cd ~
```

```
$ git clone git://github.com/edwardcapriolo/m6d_oozie.git
$ mvn install
```

20.2 一个只包含两个查询过程的工作流示例

通过配置特定的目录结构可以创建一个工作流。这里的目录结构里含有我们所需要的 JAR 文件，一个是 *job.properties* 文件，另一个是 *workflow.xml* 文件。这个目录必须创建在 HDFS 中，但最好先在本地创建好文件夹，然后将其拷贝到 HDFS 中：

```
$ mkdir myapp
$ mkdir myapp/lib
$ cp $HIVE_HOME/lib/*.jar myapp/lib/
$ cp m6d_oozie-1.0.0.jar myapp/lib/
$ cp hive_test-4.0.0.jar myapp/lib/
```

文件 *job.properties* 中配置有文件系统 *NameNode* 地址和 *Jobtracker* 地址。同时，还可以在这里设置一些 Hadoop Job 配置文件中的属性信息：

如下是个 *job.properties* 文件内容的例子：

```
nameNode=hdfs://rs01.hadoop.pvt:34310
jobTracker=rjt.hadoop.pvt:34311
queueName=default
oozie.libpath=/user/root/oozie/test/lib
oozie.wf.application.path=${nameNode}/user/root/oozie/test/main
```

文件 *workflow.xml* 对动作进行了定义：

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="java-main-wf">
  <start to="create-node"/>
  <!--The create-node actual defines a table if it does not
  already exist-->
  <action name="create-node">
    <java>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.job.queue.name</name>
          <value>${queueName}</value>
        </property>
      </configuration>
      <main-class>com.m6d.oozie.HiveServiceBAction</main-class>
      <arg>rhiveservice.hadoop.pvt</arg>
      <arg>10000</arg>
      <arg>CREATE TABLE IF NOT EXISTS zz_zz_abc (a int, b int)</arg>
    </java>
    <!-- on success proceded to query_node action -->
```

```

        <ok to="query_node"/>
        <!-- on fail end the job unsuccessfully-->
        <error to="fail"/>
    </action>

    <!-- populate the contents of the table with an
    insert overwrite query -->
    <action name="query_node">
        <java>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.job.queue.name</name>
                    <value>${queueName}</value>
                </property>
            </configuration>
            <main-class>com.m6d.oozie.HiveServiceBAction</main-class>
            <arg>rhiveservice.hadoop.pvt</arg>
            <arg>10000</arg>
            <arg>INSERT OVERWRITE TABLE zz_zz_abc SELECT dma_code,site_id
            FROM BCO WHERE dt=20120426 AND offer=4159 LIMIT 10</arg>
        </java>
        <ok to="end"/>
        <error to="fail"/>
    </action>

    <kill name="fail">
        <message>Java failed, error message
        [${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>
    <end name="end"/>
</workflow-app>

```

20.3 Oozie 网页控制台

使用 Oozie 网页控制台可以获得操作执行的细节，也因此便于进行任务错误定位。Oozie 会在每个 map 任务 (task) 中启动每个动作并获取其所有的输入和输出。Oozie 可以很好地展示这些信息并提供链接，方便到 Hadoop 的 *JobTracker 网页控制台* 查看这些任务 (job) 状态信息。

图 20-1 是一张 Oozie 网页控制台的屏幕截图。

20.4 workflow 中的变量

基于完全静态查询的工作流是很有用的，但是并不太实用。Oozie 中的大多数使用场景

是执行今天或本周的一系列文件的处理过程。

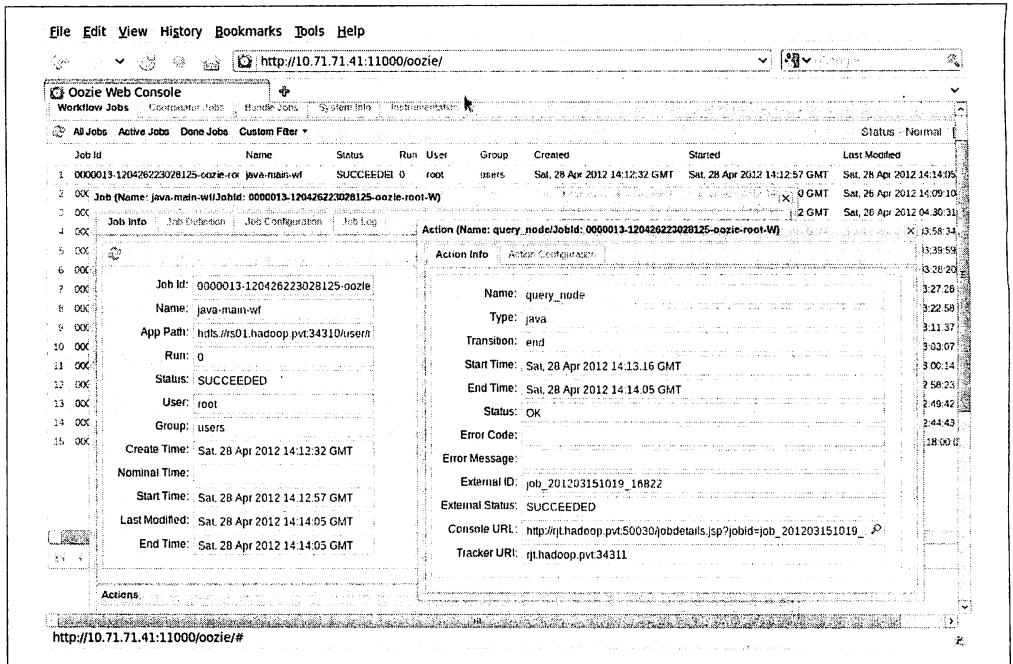


图 20-1 Oozie 网页控制台屏幕截图

在前面的工作流中，你可能注意到了 KILL 这个标记以及其中插入的变量：

```
<kill name="fail">
  <message>Java failed, error message
    [${wf:errorMessage(wf:lastErrorNode())}]</message>
</kill>
```

Oozie 提供了一个 ETL 来访问这些变量。*job.properties* 文件中定义的键-值对可以通过这种方式进行引用。

20.5 获取输出

Oozie 中还提供了一个可以放置到动作中的 `<captureOutput/>` 标记。通过这个标记可以捕获输出，并将其连同错误信息通过邮件发出或者将输出发送给其他处理过程。Oozie 在每个动作中都设置了一个 Java 属性，它可以用于输出那些可以写入的文件名。如下代码展示了如何获取这个属性：

```
private static final String
OOZIE_ACTION_OUTPUT_PROPERTIES = "oozie.action.output.properties";
```

```

public static void main(String args[]) throws Exception {
    String oozieProp = System.getProperty(OOZIE_ACTION_OUTPUT_PROPERTIES);
}

```

用户的应用程序可以向该位置输出数据。

20.6 获取输出到变量

我们已经讨论了如何获取输出以及如何获取 Oozie 变量，将这两者结合在一起使用，可以满足日常的工作流使用需求了。

看一下我们前面的那个例子，可以看到，我们是从硬编码的 FROM BCO WHERE dt=20120426 这天获取数据的。如果我们期望每天都执行这个工作流的话，那么我们就需要将硬编码的 dt=20120426 内容使用一个日期变量进行替换：

```

<action name="create_table">
  <java>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
    <configuration>
      <property>
        <name>mapred.job.queue.name</name>
        <value>${queueName}</value>
      </property>
    </configuration>
    <main-class>test.RunShellProp</main-class>
    <arg>/bin/date</arg>
    <arg>+x=%Y%m%d</arg>
    <capture-output />
  </java>
  <ok to="run_query"/>
  <error to="fail"/>
</action>

```

这将产生如下类似的输出：

```

$ date +x=%Y%m%d
x=20120522

```

那么就可以在这个处理过程后面使用这个输出了：

```

<arg>You said ${wf:actionData('create_table')['x']}</arg>

```

使用 Oozie 还可以做很多其他的工作，包括将 Hive 任务 (job) 和使用其他工具 (如 Pig、Java MapReduce 等) 实现的任务 (job) 的整合使用。

Hive 和亚马逊网络服务系统(AWS)

——Mark Grover

Amazon 提供的作为 AmazonWeb 服务 (AWS) 一部分的就是弹性 MapReduce (EMR)。使用 EMR 可以按需组建一个由节点组成的集群。这些集群用于 Hadoop 和 Hive 的安装和配置。(用户也可以配置这个集群, 以使用 Pig 或者其他工具。)用户可以执行 Hive 查询语句, 然后在完成所有任务后终止这个集群, 整个过程只需为使用这个集群的时间付费。本节将描述如何使用弹性 MapReduce, 介绍一些基本的实践, 包括使用 EMR 和其他一些替代方案的属性和配置。

在阅读本章时, 用户可以通过如下链接获取在线 AWS 文档说明: <http://aws.amazon.com/elasticmapreduce/>。本章将不会涵括所有的在 Amazon EMR 中使用 Hive 的细节。本章只会提供一个概述并详细探讨一些实践操作。

21.1 为什么要弹性 MapReduce

较小的团队和初创的公司经常没有足够的资源来建立自己的集群, 而自建的集群需要在初始阶段进行固定的投资消耗, 需要有能力和搭建、服务和切换, 包括维护 Hadoop 和 Hive 安装。

从另一方面来说, 弹性 MapReduce 性价比高, 而且其由 Amazon 来安装和维护。这对于那些没有能力或者不期望投资自己的集群的团队来说是非常有益的, 甚至对于大的团队也可以在不影响自己的生产集群的情况下对新工具和新创意进行测试。

21.2 实例

一个 Amazon 集群由一个或者多个实例组成。各种实例大小不同, 使用不同的 RAM,

提供不同的计算能力、存储空间、平台和 I/O 处理能力。很难说针对用户的使用场景使用多少的集群可以达到最佳效果。使用 EMR，可以在开始的时候使用较小的实例，并使用如 Ganglia 这样的工具监控性能，然后对多个不同大小的实例进行试验，找到平衡成本和性能的最佳值。

21.3 开始前的注意事项

在使用 Amazon EMR 之前，用户需要先设置一个 Amazon Web Services(AWS)账号。在“*Amazon EMR*”开始指南中提供了如何注册一个 AWS 账号的操作说明。

用户还需要创建一个 Amazon S3 数据桶用于存储用户输入数据并获取用户 Hive 处理过程的输出结果。

当用户设置好 AWS 账号后，需要确保其所有的 Amazon EC2 实例、键对、安全组和 EMR workflows 都位于同一个区域以避免跨区域数据传输消耗。将 Amazon S3 数据桶和 EMR workflows 设置在同一个区域将获得更高的性能。

尽管 Amazon EMR 支持多个版本的 Hadoop 和 Hive，不过只有一些配套的 Hadoop 和 Hive 版本才支持 Amazon EMR。查看 Amazon EMR 文档以获取支持的 Hadoop 和 Hive 组合版本。

21.4 管理自有 EMR Hive 集群

Amazon 提供了多种方式来创建、终止和修改 Hive 集群。目前用户可以通过如下 3 种方式来管理用户的 EMR Hive 集群。

1. EMR AWS 管理控制台（基于 Web 的前端控制台）

这是搭建集群而无需安装过程的最简单的方式。不过，当规模增大时，最好使用其他的方式进行搭建。

2. EMR 命令行交互界面

这种方式允许用户使用简单的基于 Ruby 的名为 elastic-mapreduce 的 CLI 来管理集群。Amazon EMR 在线文档描述了如何安装和使用 CLI。

3. EMR API

这种方式允许用户使用一个名为 EMR API 的特定语言的 SDK 来管理 EMR 集群。在 Amazon EMR 文档中有介绍下载和使用这种 SDK 的详细介绍。SDK 目前支持 Android、iOS、Java、PHP、Python、Ruby、Windows 和 .NET。SDK 的一个缺点是，有时特定的

SDK 包含的实现可能没有最新版本的 AWS API 提供的新。

通常会使用多种方式来管理 Hive 集群。

下面是一个使用 Ruby elastic-mapreduce CLI 来创建一个包含有配置好的 Hive 的单节点 Amazon EMR 集群的例子。其不仅提供了执行任务和退出的功能，还为集群安装了一个交互界面。这样的集群是学习 Hive 的理想工具。

```
elastic-mapreduce --create --alive --name "Test Hive" --hive-interactive
```

如果用户想使用 Pig 的话，那么在这里增加 `--pig-interface` 选项即可。

下一步就是按照 Amazon EMR 文档中描述的方式登录到这个集群中了。

21.5 EMR Hive 上的 Thrift Server 服务

通常情况下，Hive Thrift Server（请参考第 16 章的内容）是监听来自端口 10 000 的连接。不过，在 Amazon Hive 安装中，这个端口值取决于所使用的 Hive 的版本。这样做的目的是可以允许用户安装并并发支持多个版本的 Hive。因此，Hive 0.5.x 版本使用的是 10 000 端口。Hive 0.7.x 使用的是 10 001 端口，而 Hive v0.7.1 使用的是 10 002 端口。当 Amazon EMR 中新增新版本的 Hive 时这个端口值应该是要改变的。

21.6 EMR 上的实例组

每个 Amazon 集群都具有一个或者多个节点。每个节点都可以放入到如下 3 种实例组中的一组中。

1. 管理者实例组

这个实例组只含有一个节点，称之为管理者节点。这个管理者节点和 Hadoop 的 master 节点的功能是相同的。其上面运行着 namenode 和 jobtracker 守护进程，不过这上面还安装有 Hive。另外，其上还安装有一个 MySQL 服务器，其被配置为 EMR Hive 的元数据存储。（Apache Hive 中默认是使用 Derby 作为元数据存储的，这里不会使用这个。）在管理者节点上还运行着一个实例管理器。其用于在其他 2 种实例组中执行和管理其他实例。需要注意的是这个实例管理器同样会使用到管理者节点上的 MySQL 服务器。如果这个 MySQL 服务器不可用的话，那么这个实例控制器就无法执行和管理实例。

2. 核心实例组

核心实例组中的节点和 Hadoop slave 节点的具有相同的功能，会同时启动 datanode 和 tasktracker 守护进程。这些节点用于 MapReduce 任务计算，同时也用于 HDFS 存储。

一旦集群启动，这个实例组内的节点个数只能增不能减。需要特别注意的是，一旦集群终止了，这些节点上存储的数据就会丢失。

3. 任务 (task) 实例组

这是一个可选的实例组。在本组中的节点同样具有 Hadoop 中 `salve` 节点的功能。不过，它们只执行 `tasktracker` 进程。因此，这些节点用于 MapReduce 任务 (task)，但无法用于存储 HDFS 数据块。集群启动后，任务 (task) 实例组内的节点个数可能有时增加，有时减少。

当用户在高峰期的那几个小时增加集群处理能力，然后再调整到正常情况时，使用任务 (task) 实例组是非常便利的。同样对于那些既希望可以低成本使用现实现卖实例，而又期望当集群中节点被移除后不会有丢失数据的风险的情况，同样是适用这种解决办法的。

如果用户使用的是单点集群的话，那么这个节点将同时是管理者节点和核心节点。

21.7 配置 EMR 集群

在使用 EMR 集群时，用户经常需要部署自己的配置文件。一般要配置的文件有 `hive-site.xml`、`.hiverc`、`hadoop-env.sh`。Amazon 提供了一种方式来重载这些配置文件。

21.7.1 部署 `hive-site.xml` 文件

为了重载 `hive-site.xml` 文件，首先需要上传用户自定义 `hive-site.xml` 文件到 S3 中。我们假设其已经上传到 S3 中如下路径：`s3n://example.hive.oreilly.com/tables/hive_site.xml`。



提示

建议使用最新 `s3n` “模式”来访问 S3，其比之前的 `s3` 模式具有更高的性能。

如果用户是通过 `elastic-mapreduce Ruby` 客户端来启动集群的话，那么可以使用类似如下的命令使用用户自定义的 `hive-site.xml` 来分割集群：

```
elastic-mapreduce --create --alive --name "Test Hive" --hive-interactive \  
--hive-site=s3n://example.hive.oreilly.com/conf/hive_site.xml
```

如果用户是使用 SDK 来分割集群的话，那么可以使用合适的方法来重载 `hive-site.xml` 文件。在引导程序之后，用户需要 2 个配置步骤。其一是安装 Hive，其二是部署 `hive-site.xml` 文件。第 1 个步骤安装 Hive 需要同时调用 `--install-hive` 和 `--hive-versions`，后者的值是使用逗号分割符的期望安装到集群中的 Hive 版本列表。

第 2 个步骤是安装 Hive site 配置文件，使用的命令是`--install-hive-site`，其后跟着一个参数如`--hive-site=s3n://example.hive.oreilly.com/tables/hive_site.xml`，来指定要使用的 `hive-site.xml` 文件所在的位置。

21.7.2 部署.hiverc 脚本

对于.hiverc，用户同样需要先将其上传到 S3 中。然后用户可以使用配置过程或者在引导程序来将这个文件部署到集群中。需要注意的是.hiverc 可以部署到用户根目录下，或者部署在 Hive 安装目录下的 bin 目录下。

1. 使用配置步骤部署.hiverc 脚本

在写本篇时，Amazon 提供的名为 `hive-script` 的 Ruby 脚本还没有提供重载.hiverc 文件的功能，这个脚本可以在如下地址查看到：`s3n://us-east-1.elasticmapreduce/libs/hive/hive-script`。

因此，无法像安装 `hive-site.xml` 文件那么轻松地安装.hiverc 文件。不过，如果用户不介意修改 Ruby 代码的话，通过简单修改 Amazon 提供的 `hive-script` 脚本是可以启用安装.hiverc 的。进行这样的修改之后，需要将这个文件上传到 S3，然后使用这个版本而非 Amazon 提供的版本即可。修改脚本后可以将.hiverc 安装到用户的根目录下或者 Hive 安装目录下的 bin 目录下。

2. 使用辅助工具脚本部署.hiverc 脚本

或者，用户可以创建一个自定义的引导程序脚本来将.hiverc 文件传送到管理者节点的用户根目录下或 Hive 的安装目录下的 bin 目录下。在这样的脚本中，用户应该首先使用 S3 访问码配置集群上的 `s3cmd` 脚本，用户也需要使用这个 S3 访问码才能从 S3 下载.hiverc 文件。然后，使用类似如下这样的简单命令来从 S3 中下载文件，并将其部署到用户根目录下：

```
s3cmd get s3n://example.hive.oreilly.com/conf/.hiverc ~/.hiverc
```

然后在集群创建过程中使用一个引导程序动作来调用这个脚本，这和设置其他的引导程序动作是一样的。

21.7.3 建立一个内存密集型配置

如果用户执行的是一个内存密集型的任务 (Job)，Amazon 提供了一些预定义的引导程序动作可用于调优 Hadoop 配置参数。例如，在划分集群的时候可以使用 `memory-intensive` 引导程序动作，可以在 `elastic-mapreduce -create` 命令后使用如下的标记 (为显示美观，进行了换行)：

```
--bootstrap-action
  s3n://elasticmapreduce/bootstrap-actions/configurations/latest/memor
y-intensive
```

21.8 EMR 上的持久层和元数据存储

EMR 集群本身开始时就会在集群的管理者节点上安装一个 MySQL 服务器。默认情况下，EMR Hive 使用这个 MySQL 服务器作为元数据存储。不过，当这个集群被用户终止后所有存在节点上的数据也将会被清空！这通常是不可接受的。因为用户需要将表模式等信息保持在一个稳固的元数据存储上。

用户选取使用如下多种方法中的一种来绕过这个限制。

1. 使用 EMR 集群外的稳固的元数据存储

在第 2.5.3 节“使用 JDBC 连接元数据存储”中我们已经介绍过了如何在 Hive 中安装使用外部元数据存储。用户可以选用基于 MySQL 的 Amazon RDS（关系型数据服务），或者其他内部数据库服务器来作为元数据存储。当用户期望多个 EMR 集群使用同一个元数据存储或者同一个 EMR 集群上执行多个版本的 Hive 时，这是可以采取的最好的选择。

2. 使用初始化脚本

如果用户不希望使用外部数据库服务器来作为元数据存储的话，那么用户同样可以结合用户初始化脚本来使用管理者节点上的元数据存储。用户可以将类似如下的创建表语句保存在名为 `startup.q` 的文件中：

```
CREATE EXTERNAL TABLE IF NOT EXISTS emr_table(id INT, value STRING)
PARTITIONED BY (dt STRING)
LOCATION 's3n://example.hive.oreilly.com/tables/emr_table';
```

很有必要在创建表语句中使用 `IF NOT EXISTS` 语句，来确保不会删除管理者节点元数据存储中由之前初始化脚本创建好的同名的表。

到这里，我们已经在元数据存储中创建好表结构信息了，但是我们还没有加入分区信息。在 `startup.q` 文件中创建表语句后加入如下一行命令即可：

```
ALTER TABLE emr_table RECOVER PARTITIONS;
```

这行命令可以在元数据存储中补充所有的分区元数据信息内容。除了使用用户自定义的初始化脚本外，还可以使用 `.hiverc` 达到同样的效果，这个文件在 Hive CLI 启动时会

自动执行一遍。(我们将在第 21.14 节“EMR 和 EC2 以及 Apache Hive 的比较”中进行讨论。)

使用 `hiveirc` 的好处是其可以被自动调用。而缺点是，每次执行，Hive CLI 都会被调用一次，这就为后续的调用带来了不必要的时间消耗。

使用用户自定义初始化脚本的好处是，用户可以在工作流的执行周期中进行更有效的控制。不过，用户需要自己控制这个调用。无论如何，使用一个保存 Hive 语句的文件来用于初始化有一个附带的好处是，用户可以通过版本控制来跟踪 DDL 变更情况。



提示

当表和分区变得很多时，元数据信息将变得更多，这样这个系统中的初始化脚本执行的时间就会越来越长。因此如果表或分区非常多，不建议使用这种方法。

3. 在 S3 上进行 MySQL dump 操作

还有一种方式（尽管麻烦），就是在集群终止之前对元数据存储进行备份，然后在下一个工作流开始时恢复到元数据存储中。S3 在集群未使用时执行这个备份是最合适不过了。

需要注意的是，这个元数据存储并不能在 EMR 集群上支持的多个版本 Hive 共用。假设用户划分出一个集群，其中安装有 Hive 0.5.0 和 v0.7.1 2 个版本的话。当用户使用 Hive v0.5 创建了一张表后，那么是无法使用 Hive v0.7.1 来访问这张表的。如果用户期望多个 Hive 版本共用同一个元数据存储，用户必需要使用一个外部稳固的元数据存储。

21.9 EMR 集群上的 HDFS 和 S3

在 EMR 集群中 HDFS 和 S3 都有其独特的角色。当集群终止后其上所有的数据都会被清除掉。因为 HDFS 是由核心实例组中的临时存储节点组成的，所以当集群终止后 HDFS 上存储的数据会全部丢失。

从另一方面来说，S3 提供了一个和 EMR 集群相关联的持久性存储。不过，集群中的输入数据必须是存储在 S3 上的数据，而 Hive 处理过程产生的最终结果同时也必须持久化到 S3 中。不管怎样，S3 是 HDFS 的一个昂贵的替代存储方案。不过，处理过程中的中间数据应该存储在 HDFS 中，而只有需要持久化的最终结果才存储到 S3 中。

需要注意的一点是，如果使用 S3 作为输入数据源的话，存在一个缺点就是，无法使用 Hadoop 的本地化数据处理优化，其带来的性能表现可能是非常显著的。如果对于用户的分析来说这个功能是非常必要的，那么建议最好在处理前从 S3 中将些“热”数据导入到 HDFS 中，然后再进行处理。通过这个初始过程，就可以在后续的处理中使用

Hadoop 的本地处理优化。

21.10 在 S3 上部署资源、配置和辅助程序脚本

用户需要将所有的初始化脚本、配置脚本（例如 `hive-site.xml` 和 `.hiverc` 文件）、资源文件（例如需要载入到分布式缓存中的文件、UDF 或者 `streaming` 使用到的 JAR 文件等）等上传到 S3 中。因为 EMR Hive 和 Hadoop 安装原生地就可以处理 S3 路径，所以在随后的 Hadoop 处理任务中可以直接使用这些文件。

例如，用户可以将如下几行命令增加到 `.hiverc` 文件中，其可以正常执行：

```
ADD FILE s3n://example.hive.oreilly.com/files/my_file.txt;  
ADD JAR s3n://example.hive.oreilly.com/jars/udfs.jar;  
CREATE TEMPORARY FUNCTION my_count AS 'com.oreilly.hive.example.MyCount';
```

21.11 S3 上的日志

Amazon EMR 会将日志写入到 `log-uri` 字段所指向的 S3 路径下。其中包含有集群引导程序动作所产生的日志和在不同的集群节点上执行的守护进程所产生的日志。`log-uri` 这个配置项可以在 `elastic-mapreduce` Ruby 客户端安装目录下的 `credentials.json` 文件中进行配置；或者也可以在使用 `elastic-mapreduce` 划分集群时，使用 `--log-uri` 标记显式地进行指定。如果这个内容没有设置的话，那么就无法在 S3 中获得那些日志。

如果用户设置的工作流一旦认为遇到错误就终止掉的话，那么在集群终止后会丢失所有存放在集群上的日志信息。如果用户指定了 `log-uri` 配置的值，那么即使集群终止掉了，还是可以在 S3 上指定的路径下找到这些日志的。这些日志可以帮助用户确认产生失败的原因，帮助解决问题。

不过，如果用户将日志保存在 S3 中要记住尽量频繁地将不需要的日志清除掉以减少产生不必要的存储成本！

21.12 现买现卖

现买现卖业务允许用户随着需求的变化获得更低的价格来使用 Amazon 实例。Amazon 的在线文档对其有着非常详细的描述。

假设根据实际使用情况，用户希望所具有的 3 个实例组是现买现卖的。在这种情况下，在工作流的任意一个阶段都是可能导致集群终止的，这样就会导致丢失中间临时数据。如果重新进行这个计算很“便宜”的话，那么这可能并非是个严重的问题。一种解决办法就是将中间数据持久化到 S3 中，那么任务就可以从这些镜像中重新执

行了。

另一种方式就是只将任务实例组的节点作为现买现卖节点。如果这些现买现卖节点因为无效或者因为现买现卖价格增长而剔除出集群的话，原来的 workflow 在管理者和核心节点上还可以继续执行，而且不会有数据丢失。现买现卖节点重新加入到集群中后，MapReduce 任务可以侦测到它们，可以加快整个 workflow。

使用 elastic-mapreduce Ruby 客户端，现买现卖实例可以通过 --bid-price 选项指定一个权重来进行排序。下面这个例子展示了如何创建一个具有单个管理者、2 个核心节点和 2 个现买现卖节点（位于任务实例组）的集群，并为其指定权重为 10 美分：

```
elastic-mapreduce --create --alive --hive-interactive \  
--name "Test Spot Instances" \  
--instance-group master --instance-type m1.large \  
--instance-count 1 --instance-group core \  
--instance-type m1.small --instance-count 2 --instance-group task \  
--instance-type m1.small --instance-count 2 --bid-price 0.10
```

如果用户使用 Java SDK 划分一个微集群的话，那么可以使用下面的 GroupConfig 实例配置管理者、核心和任务实例组：

```
InstanceGroupConfig masterConfig = new InstanceGroupConfig()  
.withInstanceCount(1)  
.withInstanceRole("MASTER")  
.withInstanceType("m1.large");  
InstanceGroupConfig coreConfig = new InstanceGroupConfig()  
.withInstanceCount(2)  
.withInstanceRole("CORE")  
.withInstanceType("m1.small");  
InstanceGroupConfig taskConfig = new InstanceGroupConfig()  
.withInstanceCount(2)  
.withInstanceRole("TASK")  
.withInstanceType("m1.small")  
.withMarket("SPOT")  
.withBidPrice("0.05");
```



提示

如果某个 map 或者 reduce 任务 (task) 失败了，Hadoop 就需要重新启动它们。如果同一个 task 失败了 4 次 (可配置的，可以通过设置 MapReduce 属性 `mapred.map.max.attempts` 修改 map 任务的最多尝试次数；通过修改 `mapred.reduce.max.attempts` 可以修改 reduce 任务的最多尝试次数)，那么整个任务 (job) 就会失败。如果依赖于太多的现买现卖实例的话，那么用户的任务 (job) 可能会因为 TaskTracker 被移除出集群导致无法推测执行或导致整个任务 (job) 失败。

21.13 安全组

Hadoop 的 *JobTracker* 和 *NameNode* 用户接口可以在 EMRmaster 节点上分别在端口 9 100 和 9 101 上被访问到。用户可以通过 ssh 隧道或者使用动态 SOCKS 代理来访问它们。

为了在用户客户端机器上（在 Amazon 网络之外）通过浏览器访问到这些信息，用户需要通过 AWS 网页控制台对 Elastic MapReduce 管理者安全组进行修改，增加一个新的用户自定义 TCP 规则将客户端机器的 IP 在 9 100 和 9 101 端口上进行绑定即可。

21.14 EMR 和 EC2 以及 Apache Hive 的比较

对于 EMR 的一个弹性的替代方式就是引入多个 Amazon EC2 节点，并将 Hadoop 和 Hive 安装在一个定制的 Amazon 机器映像 (AMI) 中。这种方式使用户可以更好地对 Hadoop 和 Hive 的版本和配置进行控制。例如，用户可以在 EMR 发布某个工具最新版本之前尝试使用其新版本。

这种方式的一个缺点是，EMR 发布的定制版可能在 Apache Hive 发行版中并没有包含。例如，目前 Apache Hive 还无法完全支持 S3 文件系统（请参考 JIAR HIVE-2318）。对于 Amazon S3 查询还存在一个优化，用于减少初始化时间。这个功能只包含在 EMR Hive 中。通过增加如下配置到 `hive-site.xml` 文件中，可以开启此优化：

```
<property>
  <name>hive.optimize.s3.query</name>
  <value>true</value>
  <description> Improves Hive query performance for Amazon S3 queries
  by reducing their start up time </description>
</property>
```

同样地，也可在 Hive CLI 中执行如下命令启用此功能：

```
set hive.optimize.s3.query=true;
```

还有一个例子展示的是，如果在 HDFS 或 S3 中的目录结构是正确的话，那么可以通过命令自动补全元数据存储中的分区信息。当外部处理程序生成 Hive 表的分区目录时，使用这种方式补全元数据是非常方便的。通过如下命令即可达到这个效果，其中 `emr_table` 是表名：

```
ALTER TABLE emr_table RECOVER PARTITIONS;
```

下面是一个创建表的语句，供参考：

```
CREATE EXTERNAL TABLE emr_table(id INT, value STRING)
PARTITIONED BY (dt STRING)
LOCATION 's3n://example.hive.oreilly.com/tables/emr_table';
```

21.15 包装

Amazon EMR 提供了一个弹性的、可扩展的、安装配置简单的方式来搭建一个 Hadoop 和 Hive 集群，启动机器就可以执行查询。对于存储在 S3 上的数据同样是可以操作的。尽管大多数的配置已经为用户设置好了，用户还是有足够的灵活的方式进行一些自定义的配置。

22.1 介绍

在 Hadoop 中使用 Hive 进行数据处理，除了可以提供一种类 SQL 的语言供使用外，还提供了其他多个不错的功能。Hive 可以存储元数据，这意味着用户不需要记住数据的模式 (schema) 信息，同时，也意味着用户无需关注数据实际存储在哪里，以及以什么样的存储格式进行存储的。这就使得数据生产者、数据消费者和数据管理者之间相分离。数据生产者可以往数据中新增一行，而不会破坏数据消费者们的数据只读应用。数据管理者可以重置数据来修改存储格式，而无需修改数据生产者或者数据消费者的应用。

大部分的资深 Hadoop 用户不会仅使用一种工具来进行数据生产和数据消费。通常，用户都会以如下工具中的一种来开始使用 Hadoop: Hive、Pig、MapReduce 或者其他工具。当用户对于 Hadoop 的使用越来越深入后，他们将会发现他们所选择的工具并非是处理新任务的最优选择。对于刚开始使用 Hive 进行分析型查询的用户来说，他们可能会发现在进行 ETL 处理或构建数据模型时使用 Pig 会更好。而对于从 Pig 入手的用户来说，他们可能会发现在处理分析型查询的时候使用 Hive 会更好。

尽管像 Pig 和 MapReduce 这样的工具是不需要元数据的，但是如果提供了元数据信息的话也是有好处的。共享同一个元数据库可以使用户在多种工具间更容易地共享数据。使用 MapReduce 或者 Pig 加载和归一化数据，然后使用 Hive 来进行分析，这样的工作流是非常常见的。当所有的工具共享使用同一个元数据存储时，每种工具的用户都可以立即访问到由其他工具产生的数据，而无需其他加载或者转换步骤。

HCatalog 的存在就是为了满足这些需要的。其可以使 Hive 的元数据存储为基于 Hadoop 的其他工具所共用。其为 MapReduce 和 Pig 提供了连接器，这样用户就可以使用那些工具，从 Hive 数据仓库中读取和写入数据。其还提供了一个命令行工具，便于没有使用

Hive 的用户通过 Hive DDL 语句操作元数据存储。其还提供了一个消息通知服务，这样对于 Oozie 这样的工作流工具，在数据仓库提供新数据时，可以通知到这些工作流工具。

HCatalog 是相对独立于 Hive 的一个独立的 Apache 项目。其是 Apache 孵化器的一部分，大多数的 Apache 项目都是从这个孵化器中开始的。其有助于为其内部的项目构建社区并学习如何开发 Apache 开源软件。在写本书时，最新的版本是 HCatalog 0.4.0-incubating。

这个版本适用于 Hive 0.9、Hadoop 1.0 和 Pig 0.9.2。

(译者注：HCatalog 目前已经合并到 Apache Hive 中了，在 Apache Hive 0.10.0 正式引入了 Hcatalog。)

22.2 MapReduce

22.2.1 读数据

MapReduce 使用 Java 类 `InputFormat` 来读取输入数据。绝大多数情况下，这些类会直接从 HDFS 中读取数据。`InputFormat` 的一些实现类同样提供了从 HBase、Cassandra 和其他数据源读取数据的功能。`InputFormat` 的任务是双重的。首先，其决定了数据是如何划分成数据片然后为 MapReduce 的 map 任务 (task) 所并行处理的。其次，其提供了一个 `RecordReader`，MapReduce 使用这个类来从输入数据源中读取记录，然后将其转换成键和值来供 map 任务 (task) 进行处理。

HCatalog 提供了一个 `HCatInputFormat` 类来供 MapReduce 用户从 Hive 的数据仓库中读取数据。其允许用户只读取需要的表分区和字段。同时其还以一种方便的列表格式来展示记录，这样就不需要用户来进行划分了。



提示

`HCatInputFormat` 实现了 Hadoop 0.20 API，也就是 `org.apache.hadoop.mapreduce`，而不是 Hadoop 0.18 的 `org.apache.hadoop.mapred` API。这是因为其需要 MapReduce (0.20) API 中新增的某些功能。这意味着 MapReduce 用户需要使用这些接口来和 HCatalog 进行交互。不过，Hive 所需要的用于从磁盘读取数据的 `InputFormat` 也可以是旧的 `mapred` 下的接口实现的。因此如果用户当前使用的数据的格式是使用 MapReduce `InputFormat` 的话，那么就可以使用 `HCatalog.InputFormat` 这个类在新接口 `mapreduce` API 中是一个类，而在旧接口 `mapred` API 中是一个接口，不过前面都是将其作为类进行引用的。

在初始化 `HCatInputFormat` 时，首先要做的事情就是指定要读取的表。通过创建一个 `InputJobInfo` 类，然后指定数据库、表和分区过滤条件就可以达到这个目地。

```

/**
 * Initializes a new InputJobInfo
 * for reading data from a table.
 * @param databaseName the db name
 * @param tableName the table name
 * @param filter the partition filter
 */
public static InputJobInfo create(String databaseName,
    String tableName,
    String filter) {
    ...
}

```

databaseName 表示表所在的 Hive 数据库（或模式 schema）。如果这个值为 null，那么就会使用默认的名为 default 的数据库。tableName 表示将要读取的表的表名。它必须是非 null 的，而且必须是 Hive 中实际存在的一张表。filter 表示用户期望读取哪些分区下的数据。如果这个值为 null 的话，那么将读取整个表。这些需要特别小心，因为读取一张大表的所有分区将会导致扫描大量的数据。

过滤器的格式类似于类 SQL 的 where 语句部分。这里应该只允许指定分区字段。例如，假设要读取的分区表的分区字段名为 datestamp，那么过滤器可能就类似于 datestamp = "2012-05-26" 这样的格式。过滤器可以包含有 =、>、>=、<、<=、and 和 or 操作符。

对于 Hive v0.9.0 和之前的版本，在 ORM 映射层存在一个 bug，会导致对于 >、>=、< 或 <= 这样的过滤条件执行失败。



提示

可以通过 <https://issues.apache.org/jira/browse/HIVE-2084> 获取 HIVE-2084. D2397.1.patch 并将这个 patch 打到用户当前的 Hive 版本上，然后重新编译，就可以解决这个问题。这确实会存在一些风险，不过这取决于用户是如何部署 Hive 的。在这个 JIRA 下可以查看到一些讨论信息。

对于 InputJobInfo 实例和包含 MapReduce 任务(job)的 Job 实例，可通过 HCatInputFormat 的 setInput 将其传递给 HCatInputFormat。

```

Job job = new Job(conf, "Example");
InputJobInfo inputInfo = InputJobInfo.create(dbName, inputTableName, filter);
HCatInputFormat.setInput(job, inputInfo);

```

map 任务(task)需要指定值的类型是 HCatRecord 的。键的类型并不重要，因为 HCatalog 并不会将键提供给 map 任务(task)。例如，一个通过 HCatalog 读取数据的 map 任务(task)可能是如下形式的：

```

public static class Map extends

```

```

    Mapper<WritableComparable, HCatRecord, Text, Text> {

        @Override
        protected void map(
            WritableComparable key,
            HCatRecord value,
            org.apache.hadoop.mapreduce.Mapper<WritableComparable,
                HCatRecord, Text, HCatRecord>.Context context) {
            ...
        }
    }
}

```

HCatRecord 是 HCatalog 提供的一个用于和记录交互的类。其提供了简单的 get 和 set 方法，可以通过位置或者名称来获取记录。如果是通过列名获取字段内容的话，那么就须提供表的模式 (schema) 信息，因为每个独立的 HCatRecord 都不会保存一份这个表模式 (schema) 的引用信息。可以通过 HCatInputFormat.getOutputSchema() 方法获取到表模式信息。因为 Java 不允许按照返回值类型进行方法重载，对于不同的数据类型都需要提供其相应的 get 和 set 方法。这些方法使用的是类型对象而非标量类型（也就是说使用 java.lang.Integer 这样的 Java 对象，而不是像 int 这样的标量）。这就允许将 null 作为一个值来表示。同时还提供了对 Java 对象的 get 和 set 方法实现。

```

// get the first column, as an Object and cast it to a Long
Long cnt = record.get(0);

// get the column named "cnt" as a Long
Long cnt = record.get("cnt", schema);

// set the column named "user" to the string "fred"
record.setString("user", schema, "fred");

```

通常程序不需要读取一个输入的所有字段内容。在这种情况下，尽早尽快地去除掉不需要的字段很有意义。这对于像 RCFile 这样的列式存储更是有益的，越早过滤掉不需要的字段就意味着从磁盘读取更少的数据。这可以通过传递一个描述所需字段的模式来完成。这个过程必须在 job 配置时间内完成。如下这个例子将配置用户的 job 只读取 2 个字段，字段名分别是 user 和 url:

```

HCatSchema baseSchema = HCatBaseInputFormat.getOutputSchema(context);
List<HCatFieldSchema> fields = new List<HCatFieldSchema>(2);
fields.add(baseSchema.get("user"));
fields.add(baseSchema.get("url"));
HCatBaseInputFormat.setOutputSchema(job, new HCatSchema(fields));

```

22.2.2 写数据

和读数据类似，写数据时，需要指定数据库名和要写入的表的表名。如果要写入的表

是分区表而只想写入其中一个分区时，那么还要指定要写入的这个分区的分区名：

```
/**
 * Initializes a new OutputJobInfo instance for writing data from a table.
 * @param dbName the db name
 * @param tableName the table name
 * @param partitionValues The partition values to publish to, can be null
or empty Map
 */
public static OutputJobInfo create(String dbName,
                                   String tableName,
                                   Map<String, String> partitionValues) {
    ...
}
```

dbName 表示表所在的指定的 Hive 数据库（或模式）的名称。如果这个值为 null 的话，那么就会使用默认的 default 数据库。而 tableName 表示的是要写入的目标表名。表名必须是非 null 的，而且应该是 Hive 中实际存在的一个表。partitionValues 表示用户期望创建的分区名称。如果需要写入到特定的某个分区的话，那么这个 map 必须明确地指定这个分区。例如，如果这个表具有二级分区的话，那么在 map 中必须要指定这两级分区字段。对于非分区表，这个字段可以省略为 null。当以这种方式显式指定分区后，那么分区字段就没必要保存在数据文件中了。如果数据文件中包含了分区字段的内容，那么 HCatalog 在将数据写入到 Hive 前会将这些字段过滤丢弃掉，因为在 Hive 中是不会在数据中保存分区字段内容的。

同时向多个分区写入数据是允许的，这也就是所谓的动态分区，因为记录是在执行时动态划分到不同分区的。如果要使用动态分区，那么分区字段的值必须存在于数据中。例如，如果某张表是按照字段“timestamp”进行划分的，那么这个字段必须存在于 reducer 端的数据收集器中。这是因为 HCatalog 将读取分区字段来确定将数据写入到哪个分区去。在写数据的过程中，之前的分区字段中那些列的值将会被丢弃掉。

一旦创建好一个 OutputJobInfo 对象，然后就可以通过静态方法 setOutput 将其传递给 HCatOutputFormat：

```
OutputJobInfo outputInfo = OutputJobInfo.create(dbName, outputTableName,
null));
HCatOutputFormat.setOutput(job, outputInfo);
```

当 HCatOutputFormat 写数据时，输出的键的数据类型并不重要，而值的类型必须是 HCatRecord。可以在 reducer 阶段写数据，或者对于只有 map 过程的任务来说在 map 阶段写数据。

将上面的内容放到同一个例子中。下面的代码将会从表名为 rawevents 的表中读取时间戳为 20120531 的分区，然后对每个用户计算对应的事件个数，并将结果最终写入到表

cntd 中:

```
public class MRExample extends Configured implements Tool {

    public static class Map extends
        Mapper<WritableComparable, HCatRecord, Text, LongWritable> {

        protected void map(WritableComparable key,
            HCatRecord value,
            Mapper<WritableComparable, HCatRecord,
                Text, LongWritable>.Context context)
            throws IOException, InterruptedException {
            // Get our schema from the Job object.
            HCatSchema schema = HCatBaseInputFormat.getOutputSchema(context);
            // Read the user field
            String user = value.get("user", schema);
            context.write(new Text(user), new LongWritable(1));
        }
    }

    public static class Reduce extends Reducer<Text, LongWritable,
        WritableComparable, HCatRecord> {

        protected void reduce(Text key, Iterable<LongWritable> values,
            Reducer<Text, LongWritable,
                WritableComparable, HCatRecord>.Context context)
            throws IOException, InterruptedException {

            List<HCatFieldSchema> columns = new ArrayList<HCatFieldSchema>(2);
            columns.add(new HCatFieldSchema("user", HCatFieldSchema.Type.STRING, ""));
            columns.add(new HCatFieldSchema("cnt", HCatFieldSchema.Type.BIGINT, ""));
            HCatSchema schema = new HCatSchema(columns);

            long sum = 0;
            Iterator<IntWritable> iter = values.iterator();
            while (iter.hasNext()) sum += iter.next().getLong();
            HCatRecord output = new DefaultHCatRecord(2);
            record.set("user", schema, key.toString());
            record.setLong("cnt", schema, sum);
            context.write(null, record);
        }
    }

    public int run(String[] args) throws Exception {
        Job job = new Job(conf, "Example");
        // Read the "rawevents" table, partition "20120531", in the default
        // database
        HCatInputFormat.setInput(job, InputJobInfo.create(null, "rawevents",
            "datestamp='20120531'"));
    }
}
```

```

job.setInputFormatClass(HCatInputFormat.class);
job.setJarByClass(MRExample.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);
job.setOutputKeyClass(WritableComparable.class);
job.setOutputValueClass(DefaultHCatRecord.class);
// Write into "cntd" table, partition "20120531", in the default database
HCatOutputFormat.setOutput(job
    OutputJobInfo.create(null, "cntd", "ds=20120531"));
job.setOutputFormatClass(HCatOutputFormat.class);
return (job.waitForCompletion(true) ? 0 : 1);
}

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

```

22.3 命令行

因为 HCatalog 使用的是 Hive 的元数据存储，所以 Hive 用户是不需要使用其他额外的工具来访问元数据的。对于 Hive 用户，像以前一样使用 Hive 命令行工具就可以了。不过，对于那些不是 Hive 用户的 HCatalog 用户来说，提供了一个被称为 hcat 的命令行工具。这个工具和 Hive 的命令行工具有些类似。两者最大的不同是其只接受不会产生 MapReduce 任务 (job) 的命令。这意味着其可以支持大部分的 DDL (数据定义语言，用于定义数据的操作，如创建表这样的操作) 语句：

```
$ /usr/bin/hcat -e "create table rawevents (user string, url string);"
```

命令行支持表 22-1 中所示的这些操作。

表 22-1 hcat 命令行选项

选项	描述	例子
-e	通过命令行执行 DDL 语句	hcat -e "show tables;"
-f	执行包含有 DDL 语句的脚本文件	hcat -f setup.sql
-g	为创建的表指定组	hcat -g mygroup ...
-p	为创建的表指定目录权限	hcat -p rwxr-xr-x ...
-D	将键-值对以 Java 系统属性的形式传递给 HCatalog	hcat -D log.level=INFO
-h	提示用法帮助信息	hcat -h 或者 hcat

(译者注：原书中这个表描述有误，本表是修正后的内容。)

HCatalog 命令行不支持如下这些 SQL 操作：

- SELECT
- CREATE TABLE AS SELECT
- INSERT
- LOAD
- ALTER INDEX REBUILD
- ALTER TABLE CONCATENATE
- ALTER TABLE ARCHIVE
- ANALYZE TABLE
- EXPORT TABLE
- IMPORT TABLE

22.4 安全模型

HCatalog 并没有使用 Hive 的授权模型。不过，HCatalog 的用户认证功能和 Hive 是完全相同的。Hive 试图实现传统的数据库授权模型。不过，这在 Hadoop 生态系统中有一些限制。

因为是可以直接访问文件系统来获取底层数据的，所以 Hive 的权限控制是有限的。通过将 Hive 所对应的所有文件和文件夹的权限设置为执行 Hive 任务的用户所有这样的方式可以解决这个问题。通过这种方式可以防止其他用户读取或者写入数据，除了通过 Hive 进行操作。不过，这样有一个缺点，就是 Hive 中所有的 UDF 都需要以超级用户来执行，因为在 Hive 进程中会执行这些 UDF。因此，它们都需要数据仓库中对所有文件的读和写权限。

避免这个问题的唯一方式简单地说就是要声明 UDF 为特权操作，而且只允许有访问权限的那些人创建 UDF，尽管目前尚无机制强制这么做。在 Hive 中这可能是可以接受的，但是在 Pig 和 MapReduce 这些用户产生代码非常多的应用中显然就很难接受了。

为了解决这个问题，HCatalog 使用和存储层相同的权限进行权限控制。对于存储在 HDFS 中的数据，这意味着 HCatalog 将使用包含有数据的文件夹和文件的属性中的用户权限来判断是否有权限进行访问。如果有权限，那么其将被赋予相同的元数据访问权限。例如，如果某个用户具有表分区路径的父目录的写权限的话，那么其也就具有的这个表的写权限。

这样做的优点是这确实是安全的，很难通过修改抽象层来推翻系统权限控制。缺点是，HDFS 所支持的安全模型要比传统数据库弱得多。特别是，如字段级别的字段功能通过

这种模式是无法提供的。同时，用户只能通过将其加入到文件系统中文件对应的组中的方式来获取访问表的权限。

22.5 架构

正如前面所解释的，HCatalog 对于 Pig 和 MapReduce 使用它们的标准输入和输出机制。HCatLoader 和 HCatStorer 是相当简单的，因为它们使用的分别就是 HCatInputFormat 和 HCatOutputFormat。这 2 个 MapReduce 类很大一部分工作就是将 MapReduce 和 Hive 元数据存储结合起来。

图 22-1 展示的是 HCatalog 的架构图。

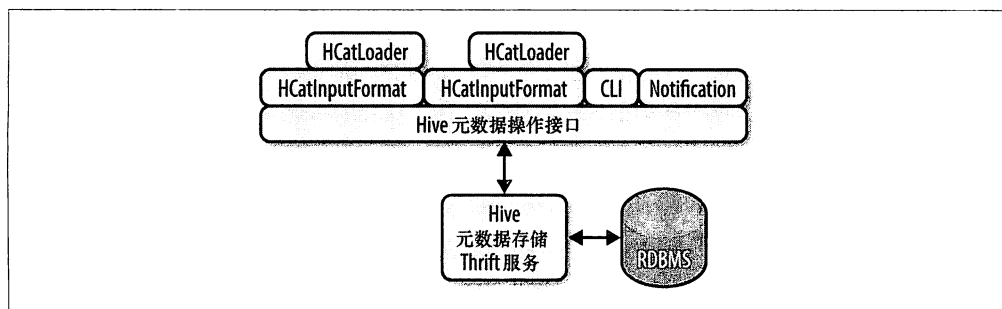


图 22-1 HCatalog 架构图

HCatInputFormat 通过和 Hive 的元数据存储进行通信来获取要读取的表和分区的信息。这包括获取表的模式，也包括获取每个分区的模式。对于每个分区，还要确定对应的用于读取该分区数据的实际的 InputFormat 和 SerDe。这些会被收集在一起，然后和所有的分区的数据划分一起，返回一个 InputSplits 对象列表。

同样地，对于每个底层的 InputFormat 都会有对应的 RecordReader 用于对划分进行解码。然后 HCatRecordReader 会通过和分区对应的 SerDe 将来自底层的 RecordReader 的值转化成 HCatRecord。这个过程中包含有为每个分区补充对应缺少的列。也就是说，当表模式中包含有分区模式中不存在的字段时，那么就会向 HCatRecord 中增加缺少的列且值为 null。同时，如果用户明确指定需要其中部分字段的话，那么这时就会去除掉不需要的字段。

HCatOutputFormat 也会和 Hive 元数据存储进行通信，以确认写入的文件格式和模式。尽管 HCatalog 当前只支持表指定的存储格式，不过无需为每个分区都打开不同的 OutputFormat。底层的 OutputFormat 已经通过 HCatOutputFormat 进行封装了。对于每个分区都会创建一个封装了底层 RecordWriter 的 RecordWriter，尽管它们都使用相同的 SerDe 来写这些新记录。当所有的分区都写完后，HCatalog 会使用一个 OutputCommitter 来将元数据信息写入到元数据存储中。

案例研究

全球有很多公司和组织使用 Hive。本章提供的案例将详细介绍有趣的和独特的使用场景和我们面临的问题，以及如何使用 Hive 这个独特的 PB 级别数据数据仓库来解决这些问题。

23.1 m6d.com(Media6Degrees)

23.1.1 M 6D 的数据科学，使用 Hive 和 R

——Ori Stitelman

在本案例研究中，我们考察了 m6d 的数据科学团队使用 Hive 对综合的海量数据提取信息的众多方法中的一种。m6d 是一家面向展示广告的公司。我们所扮演的角色就是通过创建定制的机器学习算法来为广告宣传活动寻找最好的新前景。这些算法是用于一个交付引擎之上的，其被绑定到无数个实时竞价交易，从而提供基于用户客户端行为的和按照网络地理位置提供广告条展示的方式。m5d 广告展示引擎每天都涉及到数十亿的竞价次数和进行数千万次的广告展示。自然，这样的一个系统会产生大量的数据。由本公司的广告展示交付系统产生的大部分的记录是存储在 m6d 公司的 Hadoop 集群中的，也因此，Hive 是我们的科学家对这些日志进行数据分析的主要工具。

Hive 为我们的科学家团队提供了提取和处理大量数据的一种方式。事实上，其允许我们分析在使用 Hive 之前无法进行有效分析的海量数据，并对其样本抽取和数据聚合。尽管事实上 Hive 允许我们以比之前快很多倍的速度来访问海量数据，但其并不能改变这样一个事实，那就是，以前我们所熟悉的数据科学家并不能以所产生的所有数据作为样本数据进行样本分析，也就是对全局数据进行分析而不是抽样分析。总之，Hive 为我们提供了一个提取海量数据的很好的工具。不过，数据科学家在数据科学领域使用的方法工具箱、或在统计学习领域所使用的方法如果不经实质性的改变的话，则是无法轻易适用于海量数据集分析的。

目前已经有了或者正在开发各种各样的软件包，来对海量数据集进行启发式的和非启发式的知识学习。这些软件中有一些是独立的软件实现，例如 Vowpal Wabbit 和 BBR，而其他一些是基于像 HadoopMahout 这样的大型基础架构或其他众多的针对 R 的“海量数据”处理包进行实现的。这些算法一部分是利用并行编程方法实现的，而其他则是依赖于不同的方法来实现可伸缩性的。

我们团队的几个数据科学家对于统计学习使用的主要工具是 R。R 提供了众多的包来支持众多的统计算法。更重要的是，我们对于 R 具有很多的经验，我们知道其是如何执行的，并了解它们的特性，而且非常熟悉其技术文档。不过，R 的一个主要缺点是，默认情况下其需要将所有的数据集载入到内存中。这是一个主要的限制。还有就是，一旦 R 中的数据比可以载入内存的数据要大时，系统就会出现内存交换，导致系统抖动并显著降低处理速度¹。

我们并不倡导不使用可以使用的新工具。很明显，利用好这些可伸缩技术是非常重要的，但是我们只能有那么多的时间对新技术进行调查和测试。所以现在只剩下一个选择，要么对数据进行采样来适应我们更熟悉的工具，要么使用可以用于海量数据分析的新工具。如果我们决定使用新工具的话，那么我们就可以分析更多的数据，因此也就可以降低我们的估算误差。这是非常有吸引力的。对于那些要求结果精确的情况来说这种方式是非常吸引人的。不过，学习使用新工具需要时间成本，也因需要时间学习新工具而不能去解决其他对公司有价值的问题。

一种替代方式就是我们可以对数据进行向下抽样，以便可以使用我们手头上的旧工具进行分析。不过这样我们需要处理一定的精度损失，会增加我们的估算误差。不过，这样我们就能以我们熟悉的工具来进行数据处理了。因此也就能保持使用我们当前的工具箱，不过会丢失一些精准度。然而，并非只有这两种可行的方法。在本案例研究中，我们推荐一种既能够保持现有工具箱的功能，同时又能在使用更大的样本数据集或整个数据集时保证计算精度或减小误差的方式。

图 23-1 展示了为某个广告排名设计的算法得出的值的分布情况。更高的分数表示具有更高概率的转换。本图清楚地表明，较高分数的转化率要比较低分数段的转化率低。也就是，分数在 1 以上的比分数在 0.5 和 1 之间的转化率要低。考虑到某些活动只有目标比例非常小的用户群，因此具有最好前景的是最顶端的得分者。

图 23-1 中这条表示分数和转换率之间关系的曲线是使用统计编程包 R 中的广义相加模型 (GAM)²生成的。这里就不对 GAM 进行详细的介绍了。这个案例研究的目的可以认为是一个黑盒，其可以预测出每个分数的转换率。浏览器则可以根据预测的转换率

1. <http://www.r-bloggers.com/taking-r-to-the-limit-large-datasets-predictive-modeling-with-pmm1-and-adapa/>。

2. 关于广义相加模型 (GAM) 的更多细节信息请参考 Hastie 等人在 2001 年发布的用于分析的 GAM 实现的 R 包，这个神奇的包可以通过如下链接进行下载：<http://cran.r-project.org/>。

重新进行排名，这样，预测的转换率就变成了新的分数。

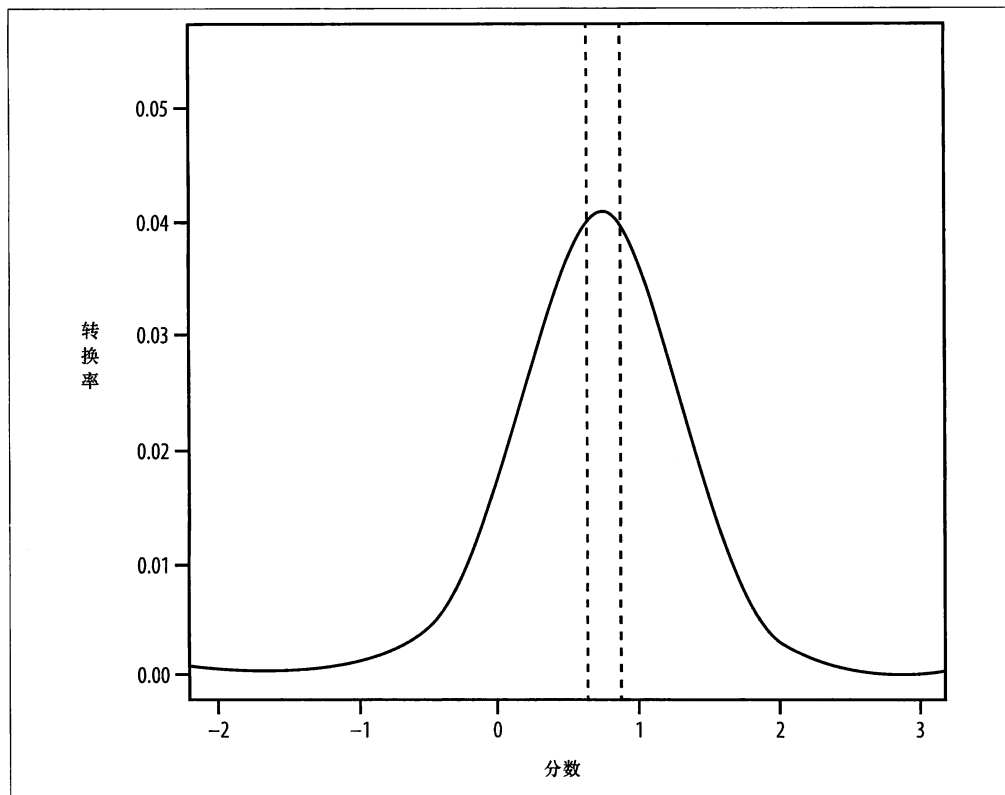


图 23-1 转换率和得分的概率

可以通过下面的方式来产生新的排名。首先，需要为每个浏览器提取分数，然后在设定的一段时间内跟踪它们，例如 5 天，并记录下它们所需的动作，然后进行转换。假设 Hive 中有张名为 `scorable` 的表，其具有表 23-1 所示的信息，并按照 `date` 和 `offer` 进行分区。

表 23-1 样例表 `scorable` 中的字段信息

名称	类型	描述
<code>score</code>	<code>double</code>	由初始算法产生的分数，这个分数没有合理排序
<code>convert</code>	<code>int</code>	转换的变量是一个二进制变量，其值为 1，如果独立浏览在随后 5 天内执行了指定的动作；反之如果没有执行，则其值为 0
<code>date</code>	<code>int</code>	浏览被赋予特定分数的日期
<code>offer</code>	<code>int</code>	一次出现的 ID

下面这个查询语句可以用于从表 `scoretable` 中抽取一组数据，用于在 R 中生成 GAM 曲线，来预测前面所述表中不同级别分数的预测转换率：

```
SELECT score,convert
FROM scoretable
WHERE date >= (...) AND date <= (...)
AND offer = (...);
1.2347 0
3.2322 1
0.0013 0
0.3441 0
```

然后通过如下代码将这些数据加载到 R 中，再使用前面所提的表的数据生成预测转换率曲线：

```
library(mgcv)
gl=gam(convert~s(score),family=binomial,data=[data frame name])
```

这种方式存在一个问题，那就是只能使用有限几天的数据来进行分析。因为一旦使用的数据集太大，甚至只要取稍微大于 3 天的数据就会导致 R 无法稳定工作。此外，如果要处理 3 天的数据量，每次执行都需要 10 分钟的时间进行初始化。因此，对于一个的计分算法，3 天的数据对于大约 300 个竞价分析来说大约需要消耗 50 个小时。

使用一个稍微不同的方式，通过简单地从 Hive 中提取数据，并利用 `mgcv` 中提供的 `gam` 函数的允许频率权重的功能，同样的分析可以使用更多的数据，获取更多的信息，而执行速度可以更快。通过在 Hive 中获取分数的最近近似值，并为每个最近近似值估算一个频率权重，通过 `GROUP BY` 语句进行转换组合。这是处理大数据集的通用方式，而且这里面并不会因为四舍五入近似关系导致结果信息的不准确，因为没有理由认为，个体分数间相差 0.001 会有任何的不同。如下这个查询语句将产生这样一个数据集：

```
SELECT round(score,2) as score,convert,count(1) AS freq
FROM scoretable
WHERE date >= [start.date] and date <= [end.date] and offer =
[chosen.offer]
GROUP BY round(score,2),convert;
1.23 0 500
3.23 1 22
0.00 0 127
0.34 0 36
```

这种方式产生的结果数据集比之前那种没有使用频率权重的方式产生的数据集要小得多。事实上，每个提供的初始数据集都含有数百万条记录，而这个新数据集相对每个提供的数据集缩小到了 6 500 条。这样可以通过如下的命令将新数据集载入到 R 中并产生新的 GAM 结果：

```
library(mgcv)
```



```
g2=gam(convert~s(score),family=binomial,weights=freq,
data=[frequency weight data frame name])
```

前面对于仅仅 3 天的数据每份提供的数据集创建 GAM 就需要 10 分钟的时间，而后者使用频率权重的方式可以在 10 秒钟左右处理基于 7 天的数据的 GAM 计算。因此，通过使用频率权重，对于 300 个估价，使用之前的方式需要 50 小时，而使用新的方法后只需要 50 秒。同时增加的速度也允许使用超过两倍的数据获得更加精确的预测转换概率。总之，频率权重方式可以在很少的时间内获得更精确的 GAM 预测估算值。

在当前的案例研究中，我们展示了如何通过对连续变量取近似值和使用频率权重进行分组，我们既可以通过使用更多的数据获得更精确的估值，又能消耗更少的计算资源，最终可以最快地进行估算。这个例子只展示了只有单一功能，按照分数计算的模型。一般来说，这种方式适用于低数据特性或者较大数据量的稀疏特性。上述的方法可以扩展到到高维问题，但需要使用其他一些小技巧。处理高维问题的一个方法就是对变量或者特性进行分桶，转换成二进制变量后，再使用 GROUP BY 进行查询，并对这些特性计算频率权重。然而，随着功能数量的增长，这些功能特性并不稀疏，再使用这种方式几乎就没有什么价值了，这时就需要寻找其他的解决办法，或者是可以处理这种大数据集的工具。

23.1.2 M6D UDF 伪随机

——David Ha 和 Runit Patel

对数据进行排序然后获取最大的 N 个值，这种需求很直截了当。用户对整个数据集基于某些标准进行排序，然后限制结果集为 N 条。但有些时候需要对元素进行分组，然后保留每个分组中的排序后的前 N 条记录。例如，计算每名歌词艺术家的排名前 10 的歌曲，或者按照商品类别和国家得到最畅销的前 100 种商品。很多的数据平台都提供了一个名为 rank() 的函数，其适用于这些使用场景。在 Hive 中我们可以通过实现用户自定义函数来达到同样的目的。我们将这个函数命名为 p_rank()，这样可以和 Hive 中使用的 rank() 有所区别。

假设我们有如表 23-2 所示的商品销售数据，我们希望查看按照类别和国家的前 3 名畅销的商品：

表 23-2 样例表 p_rank_demo 中的数据内容

类别	国家	商品名称	销售数量
movies	us	chewblanca	100
movies	us	war stars iv	150
movies	us	war stars iii	200
movies	us	star wreck	300

续表

类别	国家	商品名称	销售数量
movies	gb	titanus	100
movies	gb	spiderella	150
movies	gb	war stars iii	200
movies	gb	war stars iv	300
office	us	red pens	30
office	us	blue pens	50
office	us	black pens	60
office	us	pencils	70
office	gb	rulers	30
office	gb	blue pens	40
office	gb	black pens	50
office	gb	binder clips	60

在大多数系统中，如下 SQL 都是可以执行的：

```
SELECT
  category, country, product, sales, rank
FROM (
  SELECT
    category, country, product, sales,
    rank() over (PARTITION BY category, country ORDER BY sales DESC) rank
  FROM p_rank_demo) t
WHERE rank <= 3
```

如果想通过 HiveQL 获得相同的结果，那么第一步就需要将数据分成组。我们可以使用 DISTRIBUTE BY 语句进行分组。我们需要保证具有相同类别和国家的记录都发送到一个 reducer 上：

```
DISTRIBUTE BY
  category,
  country
```

下一步就是使用 SORT BY 语句对每组数据按照销量降序排列。因为 ORDER BY 会触发全局数据排序，所以 SORT BY 所涉及的数据会在同一个特定的 reducer 中进行排序。这里需要重新写上 DISTRIBUTE BY 语句中的划分列名：

```
SORT BY
  category,
  country,
  sales DESC
```

将所有内容都放在一起，就是如下这个样子：

```
ADD JAR p-rank-demo.jar;
CREATE TEMPORARY FUNCTION p_rank AS 'demo.PsuedoRank';

SELECT
  category, country, product, sales, rank
FROM (
  SELECT
    category, country, product, sales,
    p_rank(category, country) rank
  FROM (
    SELECT
      category, country, product,
      sales
    FROM p_rank_demo
    DISTRIBUTE BY
      category, country
    SORT BY
      category, country, sales desc) t1) t2
WHERE rank <= 3
```

子查询 t1 重新组织数据，以保证相同的商品类别和国家下的数据按照销售数量降序排列。第 2 个查询 t2 会使用到 p_rank() 函数，并将其命名为 rank，其对于每组中的行都会增加一个排名。最外层的查询会限制只保留排名前三的值。经排序后的结果如表 23-3 所示。

表 23-3 对样例表 p_rank_demo 进行 RANK 排序后的数据内容

类别	国家	商品名称	销售数量	RANK 序号
movies	gb	war stars iv	300	1
movies	gb	war stars iii	200	2
movies	gb	spiderella	150	3
movies	us	star wreck	300	1
movies	us	war stars iii	200	2
movies	us	war stars iv	150	3
office	gb	binder clips	60	1
office	gb	black pens	50	2
office	gb	blue pens	40	3
office	us	pencils	70	1
office	us	black pens	60	2
office	us	blue pens	50	3

这里是以原生 UDF 的方式实现 `p_rank()` 函数的，其参数都是确定的组属性，在本例中，就是类别和国家。这个函数可以记住上一次的参数值，因此只要成功和参数匹配，就会一直增加数值并返回排列值。一旦和参数不匹配，这个函数就会重置排列值为 1，然后重新开始计算。

这仅是个说明如何使用 `p_rank()` 函数的简单的例子。用户当然也可以按照类别和国家获取最畅销的第 10 位到第 15 位的商品。或者，用户已经计算好了每个商品类别和国家下的商品的个数，那么用户也可以结合 JOIN 使用 `p_rank()` 计算百分比。例如，假设在“movies (电影)”和“us (美国)”组下面有 1000 种产品，那么第 50 名、第 70 名和第 95 名的 RANK 值就分别对应于 500、700 和 950。这里需要明确的是，`p_rank()` 并非是 `rank()` 函数的替代函数，因为两者在某些情形下是有差异的。例如，对于相同的值，`rank()` 函数返回的值是相同的，但是 `p_rank()` 函数仍然会进行累加计算，因此具体使用需要按照期望选择并在数据上进行测试。

下面展示的是具体的代码实现。这份代码是属于公共领域的，所以用户可以随意使用、改进和修改它，以满足个人的需求：

```
package demo;

import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspectorFactory;

public class PsuedoRank extends GenericUDF {
    /**
     * The rank within the group. Resets whenever the group changes.
     */
    private long rank;

    /**
     * Key of the group that we are ranking. Use the string form
     * of the objects since deferred object and equals do not work
     * as expected even for equivalent values.
     */
    private String[] groupKey;

    @Override
    public ObjectInspector initialize(ObjectInspector[] oi)
        throws UDFArgumentException {
        return PrimitiveObjectInspectorFactory.javaLongObjectInspector;
    }

    @Override
```

```

    public Object evaluate(DeferredObject[] currentKey) throws
HiveException {
        if (!sameAsPreviousKey(currentKey)) {
            rank = 1;
        }
        return new Long(rank++);
    }

/**
 * Returns true if the current key and the previous keys are the same.
 * If the keys are not the same, then sets {@link #groupKey} to the
 * current key.
 */
private boolean sameAsPreviousKey(DeferredObject[] currentKey)
throws HiveException {
    if (null == currentKey && null == groupKey) {
        return true;
    }
    String[] previousKey = groupKey;
    copy(currentKey);
    if (null == groupKey && null != previousKey) {
        return false;
    }
    if (null != groupKey && null == previousKey) {
        return false;
    }
    if (groupKey.length != previousKey.length) {
        return false;
    }
    for (int index = 0; index < previousKey.length; index++) {
        if (!groupKey[index].equals(previousKey[index])) {
            return false;
        }
    }
    return true;
}

/**
 * Copies the given key to {@link #groupKey} for future
 * comparisons.
 */
private void copy(DeferredObject[] currentKey)
throws HiveException {
    if (null == currentKey) {
        groupKey = null;
    } else {
        groupKey = new String[currentKey.length];
        for (int index = 0; index < currentKey.length; index++) {
            groupKey[index] = String.valueOf(currentKey[index].get());
        }
    }
}

```

```

    }
}

@Override
public String getDisplayString(String[] children) {
    StringBuilder sb = new StringBuilder();
    sb.append("PsuedoRank (");
    for (int i = 0; i < children.length; i++) {
        if (i > 0) {
            sb.append(", ");
        }
        sb.append(children[i]);
    }
    sb.append(")");
    return sb.toString();
}
}

```

23.1.3 M6D 如何管理多 MapReduce 集群间的 Hive 数据访问

尽管 Hadoop 集群规模可以设计成 10 到 10 000 个节点，但是有时特定的部署需求会涉及要在不止一个文件系统或者 JobTracker 上运行任务。在 M6D 中，我们有这样的需求，例如我们有一些需要 Hadoop 和 Hive 在每小时或每天都可以按时完成的关键业务报告。不过我们的系统也支持数据科学家和销售工程师定期执行的一些特定报告。尽管使用公平调度器和能力调度器已经满足了我们的大部分需求，我们仍需要更高的调度隔离。同时，因为 HDFS 没有快照或者增量备份功能特性，我们因此需要一个对应的解决方案来防止意外的数据删除或者意外的删除表操作近而避免数据丢失。

我们的解决方案就是运行 2 个独立的 Hadoop 集群。在主集群上，数据可以被设置为 2 份或者 3 份数据冗余，而且同时会被复制到第 2 个集群上。这种方式可以保证我们对于时效性强的需求还可以有足够的资源同时提供给临时用户进行使用。此外，我们可以防止任何意外删除表或数据的情况。这种方式确实会增加部署和管理 2 个集群的开销，而这种开销在我们的使用场景下是合理的。

我们的 2 个集群分别被称为生产环境和研究环境。其都具有各自的专有数据节点 (DataNode) 和任务节点 (TaskTracker)。每个 NameNode 和 JobTracker 节点都是 DRBD 和 Linux-HA 的故障恢复方案。这 2 个集群都是部署在同一个交换网络的 (见表 23-4 和表 23-5)。

表 23-4 生产环境配置

NameNode	hdfs.hadoop.pvt:54310
JobTracker	jt.hadoop.pvt:54311

表 23-5 探索环境

NameNode	rs01.hadoop.pvt:34310
JobTracker	rjt.hadoop.pvt:34311

1. 使用 Hive 执行跨集群查询

生产集群上存在一张名为 `zz_mid_set` 的表，但是我们期望不使用 `distcp` 命令就可以在研究集群上查询这张表。通常来说，我们会尽量避免这样的操作，因为其破坏了我们的隔离设计，但是很高兴的是，这样的操作是可以做到的。

通过 `describe extended` 命令除了可以查看表具有的字段信息外还可以查看其实际存储的 HDFS 路径：

```
hive> set fs.default.name;
fs.default.name=hdfs://hdfs.hadoop.pvt:54310
hive> set mapred.job.tracker;
mapred.job.tracker=jt.hadoop.pvt:54311
hive> describe extended zz_mid_set;
OK
adv_spend_id      int
transaction_id    bigint
time              string
client_id         bigint
visit_info        string
event_type        tinyint
level            int

location:hdfs://hdfs.hadoop.pvt:54310/user/hive/warehouse/zz_mid_set
Time taken: 0.063 seconds
hive> select count(1) from zz_mid_set;
1795928
```

在第 2 个集群上，使用 `CREATE TABLE` 语句创建相同的表，表的类型需要是外部表 (`EXTERNAL`)，这样即使在第 2 个集群上执行了删除表操作，也不会真实地删除第 1 个表中的数据。需要注意的是，这里我们需要指定完整的 URL 路径。事实上，当用户通过相对路径来指定表存储路径时，Hive 实际会在元数据库中存储完整的 URL 路径：

```
hive> set fs.default.name;
fs.default.name=hdfs://rs01.hadoop.pvt:34310
hive> set mapred.job.tracker;
mapred.job.tracker=rjt.hadoop.pvt:34311
hive> CREATE TABLE EXTERNAL table_in_another_cluster
( adv_spend_id int, transaction_id bigint, time string, client_id bigint,
visit_info string, event_type tinyint, level int)
LOCATION 'hdfs://hdfs.hadoop.pvt:54310/user/hive/warehouse/zz_mid_set';
```

```
hive> select count(*) FROM table_in_another_cluster;
1795928
```

需要注意的是，之所以这样的跨集群操作可以工作，是因为两个集群的网络是相通的。我们所提交的任务所在的 TaskTracker 节点需要能够访问另一个集群的 NameNode 节点和所有的 DataNode 节点。Hadoop 的设计理念中有一项就是转移计算而不转移数据，就是将计算尽量地转移到数据所在的位置。通过调度将计算任务转移到数据所在的节点上。在这种情况下，TaskTracker 会连接另一个集群的 DataNode。这就意味着会造成通用性能下降和网络占用增加。

2. 不同集群间的 Hive 数据冗余

对于 Hadoop 和 Hive 而言保持数据冗余要比传统关系型数据库容易得多。和传统数据库在执行多事务时会频繁改变底层数据不同，Hadoop 和 Hive 中的数据通常是“一次写入的”。增加新分区不会影响到已经存在的其他分区，而且通常来说，是按照时间日期来增加新分区的。

我们早期所使用的备份系统是一个独立的系统，也就是使用 `distcp` 命令进行操作，然后按照一定的时间间隔使用生成的 Hive 语句来增加分区。当我们想备份一张新表时，我们会先拷贝来一份已有的代码，然后修改下这个脚本的配置来处理新的表和分区。经过一段时间，我们制定了一个可以更加自动化对表和分区进行备份的系统。

这个处理过程在创建分区的同时会创建一个空的 HDFS 文件，名为：

```
/replication/default.fracture_act/hit_date=20110304,mid=3000
```

备份进程会不断地扫描需要备份的目录结构。如果其发现一个新的文件，那么就会在 Hive 元数据库中查找其对应的表和分区，然后使用查找结果来备份这个分区。成功备份后这个文件就会被删除掉。

如下的代码片段就是这个程序的主循环处理部分。首先，我们会做一些检查来确保表是存在于目标元数据存储中：

```
public void run(){
    while (goOn){
        Path base = new Path(pathToConsume);
        FileStatus [] children = srcFs.listStatus(base);
        for (FileStatus child: children){
            try {
                openHiveService();
                String db = child.getPath().getName().split("\\.")[0];
                String hiveTable = child.getPath().getName().split("\\.")[1];
                Table table = srcHive.client.get_table(db, hiveTable);
                if (table == null){
```



```

        throw new RuntimeException(db+" "+hiveTable+
            " not found in source metastore");
    }
    Table tableR = destHive.client.get_table(db,hiveTable);
    if (tableR == null){
        throw new RuntimeException(db+" "+hiveTable+
            " not found in dest metastore");
    }
}

```

通过数据库名和表名我们就可以在元数据存储中找到其对应的存储路径信息。之后，我们会做一个检查来保证这个信息并非已经存在：

```

URI localTable = new URI(tableR.getSd().getLocation());
FileStatus [] partitions = srcFs.listStatus(child.getPath());
for (FileStatus partition : partitions){
    try {
        String replaced = partition.getPath().getName()
            .replace(".", "/").replace("'", "");
        Partition p = srcHive.client.get_partition_by_name(
            db, hiveTable, replaced);
        URI partUri = new URI(p.getSd().getLocation());
        String path = partUri.getPath();
        DistCp distCp = new DistCp(destConf.conf);
        String thdfile = "/tmp/replicator_distcp";
        Path tmpPath = new Path(thdfile);
        destFs.delete(tmpPath,true);
        if (destFs.exists( new Path(localTable.getScheme()+
            "://" +localTable.getHost()+":"+localTable.getPort()+
path) ) ){
            throw new RuntimeException("Target path already exists "
                +localTable.getScheme()+"://" +localTable.getHost()+
                "://" +localTable.getPort()+path );
        }
    }
}

```

Hadoop 的 DistCP 并不适合通过编程的方式来运行。不过，我们可以传递一组字符串数据给其主函数。然后通过其返回值是否是 0 来判断是否成功执行：

```

String [] dargs = new String [4];
dargs[0]="-log";
dargs[1]=localTable.getScheme()+"://" +localTable.getHo
st()+":"+
        localTable.getPort()+thdfile;
dargs[2]=p.getSd().getLocation();
dargs[3]=localTable.getScheme()+"://" +localTable.getHo
st()+":"+
        localTable.getPort()+path;
int result =ToolRunner.run(distCp,dargs);
if (result != 0){

```

```

        throw new RuntimeException("DistCP failed "+ dargs[2]
+" "+dargs[3]);
    }

```

最后，我们拼接好 ALTER TABLE 语句来增加分区：

```

String HQL = "ALTER TABLE "+hiveTable+
    " ADD PARTITION (" +partition.getPath().getName()
    +" ) LOCATION '"+path+"'";
destHive.client.execute("SET hive.support.concurrency
=false");

destHive.client.execute("USE "+db);
destHive.client.execute(HQL);
String [] results=destHive.client.fetchAll();
srcFs.delete(partition.getPath(),true);
} catch (Exception ex){
    ex.printStackTrace();
}
} // for each partition
} catch (Exception ex) {
    //error(ex);
    ex.printStackTrace();
}
} // for each table
closeHiveService();
Thread.sleep(60L*1000L);
} // end run loop
} // end run

```

23.2 Outbrain

——David Funk

Outbrain 是领先的内容发现平台。

23.2.1 站内线上身份识别

有时，当用户想查看网站的流量情况时，很难弄清楚这些流量实际来源于哪里，特别是来源于用户网站之外的流量情况。如果用户的网站具有很多结构不同的 URL 的话，那么就无法简单地将所有的链接 URL 和用户登录页面进行匹配。

1. 对 URL 进行清洗

我们期望达到的目的就是可以将链入的链接分成站内的、直接链入的或其他 3 个分组。如果所属组类型是其他的话，那么我们将仅仅保存原始的 URL 链接。这样，就可以将像对用户站点进行的 Google 搜索这样的链接从网站流量中区分出来，等等。如果链入

的链接是空的或者值为 null，那么我们将其标记为直接链入的那组。

从现在开始，我们将假定所有的 URL 网址都已经解析到主机名或域名了，而无论用户目标具体到什么级别的粒度。就我个人而言，我喜欢使用域，因为它更简单。据说，Hive 只有一个主机名函数，但不是域名函数。

如果你只有原始 URL，则有几个选项可供选择。通过 HOST 选项，正如下面例子所展示的，可以是个给出的链接中完整的主机名，如 news.google.com 或 www.google.com，而其中的域名将缩短到最低的逻辑层次，像 google.com 或 google.com.uk。

```
Host = PARSE_URL(my_url, 'HOST')
```

也许用户正在使用一个 UDF 来处理这种情况。不管怎样，我并不在乎。重要的是我们要使用这些来进行匹配，所以用户需要根据自己的使用场景来做出最合适的选择。

2. Determining referrer type

因此，回到这个例子。比方说，我们有 3 个网站：mysite1.com、mysite2.com 和 mysite3.com。现在，我们可以把每个页面的 URL 转换成适当的类别。我们假设有一个表，表名为 referrer_identification，其字段如下：

```
ri_page_url STRING
ri_referrer_url STRING
```

现在，我们可以很容易地通过如下查询来添加链接类型：

```
SELECT ri_page_url, ri_referrer_url,
       CASE
         WHEN ri_referrer_url is NULL or ri_referrer_url = "" THEN 'DIRECT'
         WHEN ri_referrer_url is in ('mysite1.com','mysite2.com','mysite3.com')
       THEN 'INSITE'
         ELSE ri_referrer_url
       END as ri_referrer_url_classed
FROM
  referrer_identification;
```

3. Multiple URL

这都是非常简单的。但是如果使用的是一个广告网络呢？如果我们有成百上千的网站呢？如果每个站点可以有任意数量的 URL 结构呢？

如果是这样的话，我们可能也有一个包含每个 URL 的表，以及它属于什么类型的网站。让我们将这张表命名为 site_url，其有如下 2 个字段：

```
su_site_id INT
su_url STRING
```

让我们为之前的那张表 `referrer_identification` 添加一个新的字段:

```
ri_site_id INT
```

现在我们开始讨论这个问题。我们要做的是通过每个链入网址，看它是否与任何相同的站点 ID 匹配。如果是匹配的话，那么这是一个站内链接，否则不是站内链接。所以，让我们通过如下查询进行确认:

```
SELECT
  c.c_page_url as ri_page_url,
  c.c_site_id as ri_site_id,
  CASE
    WHEN c.c_referrer_url is NULL or c.c_referrer_url = '' THEN 'DIRECT'
    WHEN c.c_insite_referrer_flags > 0 THEN 'INSITE'
    ELSE c.c_referrer_url
  END as ri_referrer_url_classed
FROM
  (SELECT
    a.a_page_url as c_page_url,
    a.a_referrer_url as c_referrer_url,
    a.a_site_id as c_site_id,
    SUM(IF(b.b_url <> '', 1, 0)) as c_insite_referrer_flags
  FROM
    (SELECT
      ri_page_url as a_page_url,
      ri_referrer_url as a_referrer_url,
      ri_site_id as a_site_id
    FROM
      referrer_identification
    ) a
  LEFT OUTER JOIN
    (SELECT
      su_site_id as b_site_id,
      su_url as b_url
    FROM
      site_url
    ) b
  ON
    a.a_site_id = b.b_site_id and
    a.a_referrer_url = b.b_url
  ) c
```

对于这个查询语句有几点需要说明。在本例中，我们使用的是外连接，因为我们希望有一些外部链入链接不与其匹配，这将让它们通过。因此，我们只会抓住确实匹配的条目，如果有任何这样的链接，我们知道它们来自网站内的某处。

23.2.2 计算复杂度

假设用户要计算用户网站、网络或其他什么东西的独立访客数量的话。我们将使用一个非常简单的假想表 `daily_users` 来表示：

```
du_user_id STRING
du_date STRING
```

不过，如果有非常多的用户而且集群中又没有足够的机器的话，那么在集群中计算一个月的用户数据都会变得非常困难：

```
SELECT
  COUNT(DISTINCT du_user_id)
FROM
  daily_users
WHERE
  du_date >= '2012-03-01' and
  du_date <= '2012-03-31'
```

在所有的可能性中，如果用户集群没有太多问题，则可以使其通过 `map` 阶段，但是在 `reduce` 阶段就会出现这个问题。问题就是，它能够访问所有的记录，但却不能同时对其进行计数。当然，用户也不能每天都对其进行计算，因为这样做可能会有些多余。

1. 为什么这是个问题

计数复杂度是 $O(n)$ ，其中 n 是记录的数量，但它有一个比较高的常数因子。我们可能会想出一些聪明的切割方式，来稍微降低一点计算复杂度，但更容易的方式就是减小 n 的值。虽然有一个高 $O(n)$ 并不好，但大多数真正的问题出现在后面。如果用户处理某个问题需要运行的时间是 $n^{1.1}$ ，那么谁在乎如果 $n = 2$ 还是 $n = 1$ 呢。这样确实会较之前慢，但远远没有 $n=1$ 和 $n=100$ 之间的区别那么大。

所以，如果假设每天都有 m 条数目，而平均冗余是 x 的话，那么我们的第 1 个查询将是 $n=31*m$ 条记录。我们通过创建一个用来保存每天重复版本的临时表将查询记录减少到 $n=31*(m-x)$ 。

2. 加载一个临时表

首先，创建临时表：

```
CREATE TABLE daily_users_deduped (dud_user_id STRING)
PARTITIONED BY (dud_date STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
```

然后我们写一个每天都可以执行一次的查询模板版本，然后使用它来更新我们的临时

表。我一般将这些操作称为“metajobs”，所以我们可以称为 mj_01.sql:

```
INSERT OVERWRITE TABLE daily_users_deduped
PARTITION (dud_date = ':date:')

SELECT DISTINCT
    du_user_id
FROM
    daily_users
WHERE
    du_date = ':date:'
```

接下来，我们写一个脚本，来替换组装这个文件，然后再在指定的日期范围内运行这个脚本。为此，我们需要涉及到 3 个函数，分别为：`modify_temp_file` 函数，其用于替换一个变量；`fire_query` 函数，其实质上就是对指定文件执行 `hive -f` 命令；最后一个函数是 `delete`，用于删除文件。

```
start_date = '2012-03-01'
end_date = '2012-03-31'

for date in date_range(start_date, end_date):
    femp_file = modify_temp_file( 'mj_01.sql' , { ':date:' :my_date})
    fire_query(temp_file)
    delete(temp_file)
```

3. 查询临时表

运行这个脚本，然后就可以得到一个 $n=31*(m-x)$ 大小的表。现在，就无需一个大的 `reduce` 执行过程就可以查询这个表了。

```
SELECT
    COUNT(DISTINCT (dud_uuid))
FROM
    daily_users_deduped
```

如果这还不够的话，那么还可以按照日期来去重，也许可以每次两个日期，而不管时间间隔是多少。如果仍然有困难，那么还可以根据用户 ID 将记录散列到不同类，也许可以基于用户 ID 的第 1 个字符，进一步缩小 n 的值。

基本思路是，如果可以将 n 缩小的话，那么高 $O(n)$ 也没有什么大不了的。

23.2.3 会话化

为了分析网络流量，我们常常希望能够基于各种各样的标准来测量热度。一种方法就是将用户行为分解到会话中，一次会话代表单一的一次“使用”所包含的一系列操作。一个用户在一天内或者一月中的某几天可以多次访问某个网站，但每一次访问肯定是不一样的。

那么，什么是一个会话呢？一种定义是指相隔不超过 30 分钟的一连串的活动就是一个会话。也就是说，如果你去你的第 1 个页面，等待 5 分钟，然后去第 2 个页面，那么这是相同的会话。等待 30 分钟后再到第 3 页，仍然是相同的会话。等待 31 分钟跳转到第 4 页，这次会话将被打破了，这将是第 4 个访问页面了，而是第 2 个会话中的第 1 个页面。

一旦我们获得这些中断信息，我们就可以查看会话的属性信息，来看看发生了什么事而导致中断的。常规的方式就是通过会话长度来对链入的页面进行比较。所以，我们可能需要查清楚谷歌或 Facebook 是否给予这个网站更好的热点，这也许可以通过会话长度来进行测量。

乍一看，这似乎是一个完美的迭代过程。对于每个页面，保持倒计时，直到你找到第 1 个页面。但 Hive 是不支持迭代的。

不过，还是可以解决这个问题的。我想将这个处理过程分为 4 个阶段。

- ① 识别哪些页面浏览量是会话初始者，或“起源”页面。
- ② 对于每个页面，将其划分到正确的来源页面。
- ③ 将所有的页面浏览量聚合到每个来源页面。
- ④ 对每个来源页面进行标记，然后计算每个会话的热度。

这种方式将产生一个表，其中每一行都表示一个完整的会话，然后用户就可以查询任何想要的信息了。

1. 设置

首先定义表 `session_test` 的字段如下：

```
st_user_id STRING
st_pageview_id STRING
st_page_url STRING
st_referrer_url STRING
st_timestamp DOUBLE
```

这些内容都很简单，不过我需要提一下 `st_pageview_id` 表示的每个事物（在这种情况下就是一个页面）的唯一 ID。否则，多次查看完全相同的页面可能会令人比较困惑。本示例中，时间戳以秒为单位。

2. 找到来源页面浏览量

好的，下面让我们开始第一步（令人震惊吧！）。首先看看我们是如何找到这页面浏览会话的起始页面的。好吧，如果我们假定任何超过 30 分钟的停留就意味着一个新会话的话，那么任意的会话起始页都不可能停留超过 30 分钟或更少的时间。这是一个典型

的案例总结条件。我们要做的就是，计算每个访问页面的次数。然后，任何计数为零的页面就一定是一个起始页面。

为了能做到这一点，我们需要比较所有可能在这个页面之前的页面。这是一个代价非常大的操作，因为它需要执行一个笛卡尔交叉乘积。为了防止数据膨胀到不可收拾的大小，我们应该使用尽可能多的约束条件来限制数据量。在当前情况下，限制条件只有用户 ID，但是如果用户有一个包含众多独立站点的大型网络的话，那么还可以按照每个源进行分组：

```
CREATE TABLE sessionization_step_one_origins AS

SELECT
  c.c_user_id as ssoo_user_id,
  c.c_pageview_id as ssoo_pageview_id,
  c.c_timestamp as ssoo_timestamp
FROM
  (SELECT
    a.a_user_id as c_user_id,
    a.a_pageview_id as c_pageview_id,
    a.a_timestamp as c.c_timestamp,
    SUM(IF(a.a_timestamp + 1800 >= b.b_timestamp AND
      a.a_timestamp < b.b_timestamp,1,0)) AS c_nonorigin_flags
  FROM
    (SELECT
      st_user_id as a_user_id,
      st_pageview_id as a_pageview_id,
      st_timestamp as a_timestamp
    FROM
      session_test
    ) a
  JOIN
    (SELECT
      st_user_id as b_user_id,
      st_timestamp as b_timestamp
    FROM
      session_test
    ) b
  ON
    a.a_user_id = b.b_user_id
  GROUP BY
    a.a_user_id,
    a.a_pageview_id,
    a.a_timestamp
  ) c
WHERE
  c.c_nonorigin_flags
```

这个 SQL 可能有点长。不过其中重要的部分是计算是否是起始页码的计数器，也就是

我们定义的 `define c_nonorigin_flags`。基本上，计算过程如下行所示：

```
SUM(IF(a.a_timestamp + 1800 >= b.b_timestamp AND
      a.a_timestamp < b.b_timestamp,1,0)) as c_nonorigin_flags
```

我们将其进行分解，一部分一部分进行介绍。首先，从子查询 `a` 开始。我们使用别名 `b` 表示那些候选数据。因此，第一部分，其中的 `a.a_timestamp+1800 >=b.b_timestamp`，表示候选数据时间戳不能比限定的时间戳早 30 分钟；第二部分，`a.a_timestamp < b.b_timestamp` 这段 SQL 片段表示候选时间戳比限定时间戳值要大，这是一个检查过程，如果不通过则返回 `FALSE`。同时，因为这是个交叉运算，因为不能使用候选时间戳作为自己的限定时间戳，否则返回 `FALSE`。

现在，产生表 `sessionization_step_one_origins`，其字段信息如下：

```
ssoo_user_id STRING
ssoo_pageview_id STRING
ssoo_timestamp DOUBLE
```

3. 将 PV 分桶到起始页面中

开始第 2 步的一个好的理由就是，我们需要找到某个页面所属的起始页面。做法很简单，每个页面的起始页面必定是其之前的最近的页面。为此，我们进行另外一个大的表连接操作来检查页面时间戳和所有潜在的起始页面间的最小差值：

```
CREATE TABLE sessionization_step_two_origin_identification AS

SELECT
  c.c_user_id as sstoi_user_id,
  c.c_pageview_id as sstoi_pageview_id,
  d.d_pageview_id as sstoi_origin_pageview_id
FROM
  (SELECT
    a.a_user_id as c_user_id,
    a.a_pageview_id as c_pageview_id,
    MAX(IF(a.a_timestamp >= b.b_timestamp, b.b_timestamp, NULL)) as
c_origin_timestamp
  FROM
    (SELECT
      st_user_id as a_user_id,
      st_pageview_id as a_pageview_id,
      st_timestamp as a_timestamp
    FROM
      session_test
    ) a
  JOIN
    (SELECT
      ssoo_user_id as b_user_id,
      ssoo_timestamp as b_timestamp
```

```

FROM
    sessionization_step_one_origins
) b
ON
    a.a_user_id = b.b_user_id
GROUP BY
    a.a_user_id,
    a.a_pageview_id
) c
JOIN
(SELECT
    ssoo_usr_id as d_user_id,
    ssoo_pageview_id as d_pageview_id,
    ssoo_timestamp as d_timestamp
FROM
    sessionization_step_one_origins
) d
ON
    c.c_user_id = d.d_user_id and
    c.c_origin_timestamp = d.d_timestamp

```

这里还有很多内容要讲。首先，让我们看看如下这行语句：

```

MAX(IF(a.a_timestamp >= b.b_timestamp, b.b_timestamp, NULL)) as
c_origin_timestamp

```

我们再次使用约束值和候选值的思路进行计算。在这种情况下，b 是每一个约束值 a 的候选值。一个起始候选值是不会比页面访问的时间晚的，所以对于这类情况，我们期望找到符合标准的最新起始页。其中 null 值是无关紧要的，因为我们要保证获取一个最低值，总是有至少一个可能的起始页面的(即使是这个页面本身)。这不会得到我们期望的起始页面，但是可以给我们时间戳，这样我们就可以根据时间戳来判断是否是起始页面。

在这里，我们仅仅是将这个时间戳与所有其他潜在的起始页面时间戳进行匹配，然后我们就可以知道哪些页面属于哪些起始页面。最终产生表 `sessionization_step_two_origin_identification`，其字段信息如下：

```

sstoi_user_id STRING
sstoi_pageview_id STRING
sstoi_origin_pageview_id STRING

```

值得一提的是，这不是识别起始页面的唯一方法。用户可以基于引入，标记出所有外部链接、主页 URL 或空白链入页面(表示是直接流量)作为一个起始会话。用户还可以基于动作，只测量鼠标点击后的动作。还是有很多选择方案的，但最重要的是确定什么样的会话是起始会话。

4. 对起始页面进行聚合

对于这一点，处理起来非常容易。第 3 步，我们将会对起始页面进行聚合，这个过程真的，真的简单。对于每个起始页面，计算其对应的页面浏览量：

```
CREATE TABLE sessionization_step_three_origin_aggregation AS

SELECT
  a.a_user_id as ssto_a_user_id,
  a.a_origin_pageview_id as ssto_a_origin_pageview_id,
  COUNT(1) as ssto_a_pageview_count
FROM
  (SELECT
    ssoo_user_id as a_user_id
    ssoo_pageview_id as a_origin_pageview_id
  FROM
    sessionization_step_one_origins
  ) a
JOIN
  (SELECT
    sstoi_user_id as b_user_id,
    sstoi_origin_pageview_id as b_origin_pageview_id
  FROM
    sessionization_step_two_origin_identification
  ) b
ON
  a.a_user_id = b.b_user_id and
  a.a_origin_pageview_id = b.b_origin_pageview_id
GROUP BY
  a.a_user_id,
  a.a_origin_pageview_id
```

5. 按照起始页面类型进行聚合

现在是最后一步了，我们可以不必保留页面的所有的属性信息，尤其对于在前面的处理步骤之一的起始页面来说。然而，如果用户需要注意很多细节的话，那么在最后的处理阶段也是可以很容易将需要的信息添加进来的。下面是第 4 步：

```
CREATE TABLE sessionization_step_four_qualitative_labeling

SELECT
  a.a_user_id as ssfql_user_id,
  a.a_origin_pageview_id as ssfql_origin_pageview_id,
  b.b_timestamp as ssfql_timestamp,
  b.b_page_url as ssfql_page_url,
  b.b_referrer_url as ssfql_referrer_url,
  a.a_pageview_count as ssqfl_pageview_count
(SELECT
  ssto_a_user_id as a_user_id,
```

```

        sstoa_origin_pageview_id as a_origin_pageview_id,
        sstoa_pageview_count as a_pageview_count
FROM
    sessionization_step_three_origin_aggregation
) a
JOIN
(SELECT
    st_user_id as b_user_id,
    st_pageview_id as b_pageview_id,
    st_page_url as b_page_url,
    st_referrer_url as b_referrer_url,
    st_timestamp as b_timestamp
FROM
    session_test
) b
ON
    a.a_user_id = b.b_user_id and
    a.a_origin_pageview_id = b.b_pageview_id

```

6. 衡量热度

现在，使用我们的最终表，我们可以做任何我们想要做的事情。假设我们想知道会话数量、平均每个会话页面浏览量、每次会话的加权平均综合浏览量以及最大或最小浏览量。那么我们可以选择任何我们想要的标准，或根本没有任何标准。但是在这种情况下，我们通过链接 URL 能找到答案，其流量来源具有最好的热度。只是为了好玩，让我们也看看谁给了我们最独特的用户：

```

SELECT
    PARSE_URL(ssfql_referrer_url, 'HOST' ) as referrer_host,
    COUNT(1) as session_count,
    AVG(ssfql_pageview_count) as avg_pvs_per_session,
    SUM(ssfql_pageview_count)/COUNT(1) as weighted_avg_pvs_per_session,
    MAX(ssfql_pageview_count) as max_pvs_per_session,
    MIN(ssfql_pageview_count) as min_pvs_per_session,
    COUNT(DISTINCT ssfql_usr_id) as unique_users
FROM
    sessionization_step_three_origin_aggregation
GROUP BY
    PARSE_URL(ssfql_referrer_url, 'HOST' ) as referrer_host

```

然后我们就可以得到结果了。我们可以查看到哪个 URL 页面热度最大，以及确定忠实用户是哪些，等等。一旦我们拥有一个包含有这一切信息的临时表，尤其是具有一个更完整的定性属性信息的表，我们就可以回答各种各样的用户热度问题。

23.3 NASA 喷气推进实验室

23.3.1 区域气候模型评价系统

——Chris A. Mattmann、Paul Zimdars、Cameron Goodale、Andrew F. Hart、

Jinwon Kim、Duane Waliser、Peter Lean共同编写

自 2009 年以来，我们在美国宇航局（NASA）喷气推进实验室（JPL）的团队就已经积极发展引出了一个区域气候模型评价系统（RCMES）。这个系统起先是在美国复苏和再投资法案（ARRA）资助下展开的，此系统有以下几个目标。

- ① 便于评价和分析区域气候模式模拟输出，其可以通过对质量受控的参考数据集的可用性的观察和对各种传感器的分析理解进行评价和分析。这是一个有效的数据库结构，是一组用于计算度量模型评价指标和诊断的计算工具集合，而且其具有可伸缩的和友好的用户界面。
- ② 方便汇集大量的复杂和异构软件工具和数据访问的功能，用于展示、重组、重新格式化和可视化，这样便于将如偏差图这样的最终产品很容易地传递给最终用户。
- ③ 支持区域气候变化的评估，并进行影响分析，而且还需要通知决策者（如地方政府、农业部门、国家政府、水文学家等），这样他们可以做出对于大金融和社会具有重大影响的重要的决策。
- ④ 可以克服数据格式和元数据的异构性的问题（例如 NetCDF3/4，CF 元数据规范，HDF4/5，HDF-EOS 元数据规范）。
- ⑤ 处理时空差异（如将数据进行 180/80 的经纬度分析，例如数据可以是一个 360/360 度的经纬度网格），并确保可能最初是日数据的数据，是可以和月数据进行对比的。
- ⑥ 支持弹性扩展，在进行区域研究时，需要特别的遥感数据和气候模型输出数据，并需要进行一系列的分析，然后就会摧毁特定的系统实例。换句话说，支持瞬态分析以及快速构建/解构 RCMES 实例。图 23-2 显示了区域气候模型评价系统的体系结构和数据流。

为实现这些目标，我们已经构建了一个多层面的系统，如图 23-2 所示。从左到右看下这个图，从观测采集的可用的参考数据集，特别是从卫星遥感采集的数据集，根据用于评价的气候模型所需的气候参数进入到系统。这些参数都存储在各种任务的数据集中，这些数据被安置在一些外部存储库中，最终送入到 RCMES 系统的数据库组件（RCMED：区域气候模型评价数据库）中。

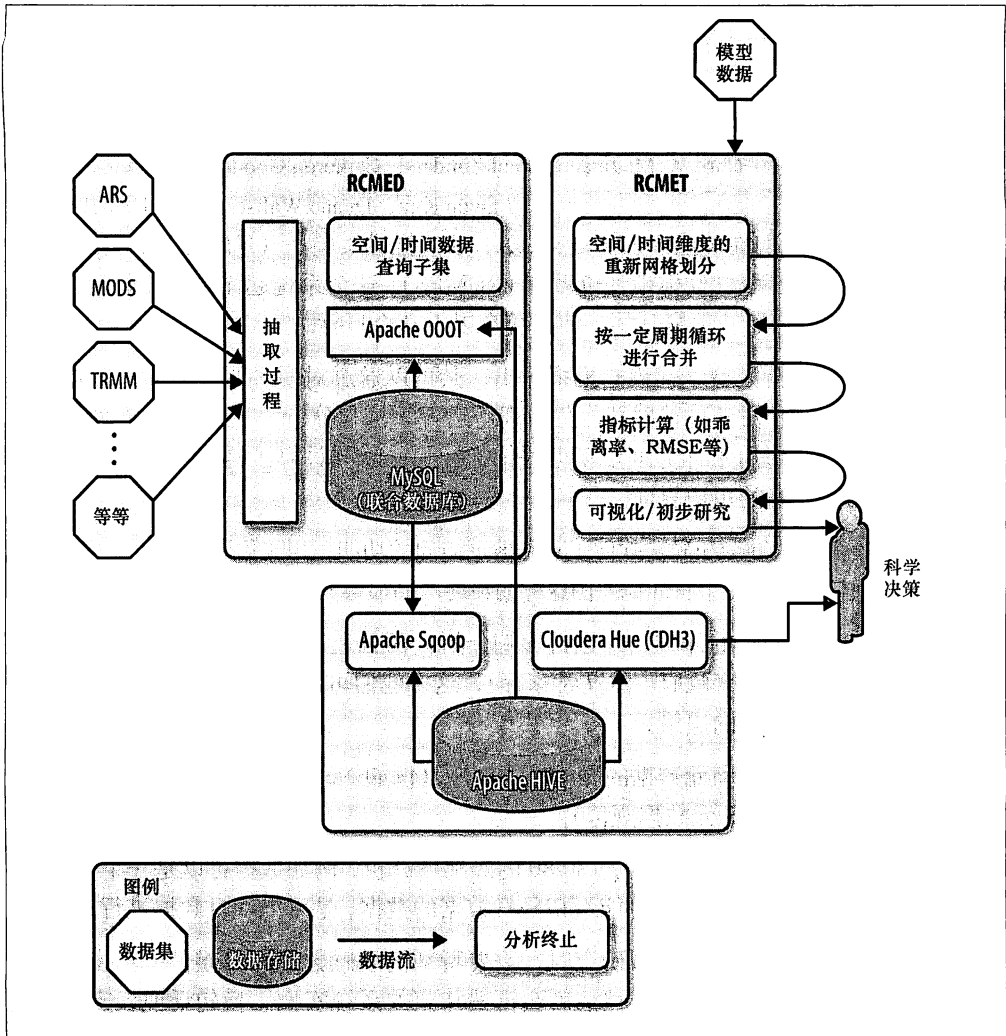


图 23-2 JPL 架构图

举一个例子，AIRS 是 NASA 的大气红外探测器，其可以提供很多参数，包括表面空气温度、湿度和重力势；MODIS 是 NASA 的热感成像光谱仪，其可以提供的参数包括云分数；而 TRMM 是 NASA 的热带降雨测量任务，其提供参数包括每月的降雨量。这些信息是在我们 RCMES 系统网站上都进行了总结，网址是 <http://rcmes.jpl.nasa.gov/rcmed/param>，如图 23-3 所示。

数据集是使用 Apache OODT 抽取器框架加载到 RCMED 中的，其所需的参数，以及参数值、空间和时间约束(以及可选的高度约束)都被装载进去了，而且还可以进行潜在的更改(如规范化，使用相同的坐标系统，从不同单位值进行换算)，最终装载到一个

MySQL 数据库中。加载到 MySQL 数据库中的数据、RCMED，通过空间/时间查询和构造子集 web 服务公开给外部客户，具体内容就是另一个话题了。对于所有的意图和目的，其提供了和 OPeNDAP 同样的功能。

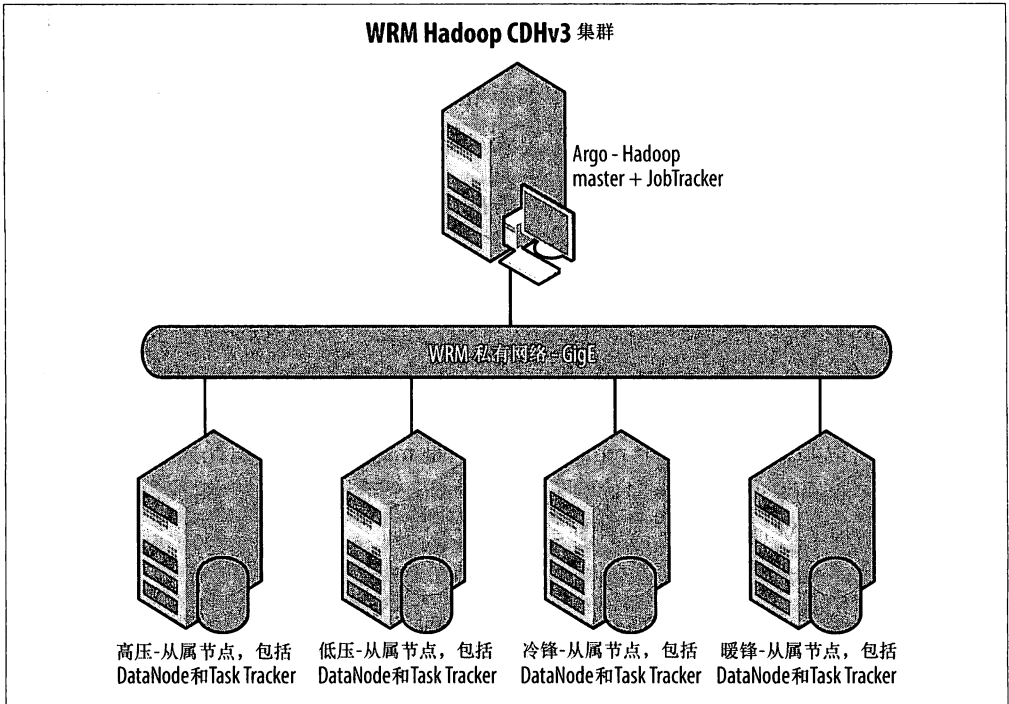


图 23-3 JPL 物理架构图

右边的图显示的是区域气候模型评估工具包(RCMET)。其为用户提供了能够从 RCMED 和在其他地方产生的气候模型输出数据进行引用的能力，并可以重组这些数据，用于在时间和空间上进行匹配，并将模型数据评估的模型输出与用户选择的参考数据进行比较参考。此时，系统允许按照季节性周期合成(例如，N 年的所有一月份，或所有夏季)，并为最终指标计算准备数据，也就是说，比较了模型输出值与遥感数据观测参数及其值。系统支持多种指标，如偏差计算、均方根误差(RMSE)，并生成相关的可视化图形，包括传统的饼图和为科学使用/决策支持的泰勒图。

23.3.2 我们的经验：为什么使用 Hive

那么，在哪里使用 Hive 呢？在载入了 60 亿行（经度、维度、时间、数据值、高度）数据集到 MySQL 后，系统崩溃了，并经历过数据丢失。这可能部分是因为我们最初的策略是将所有的数据都存储到单一的一张表中了。后来，我们调整了策略通过数据集和参数进行分表，这有所帮助但也因此引入了额外的消耗，而这并非是我们愿意接受的。

相反，我们决定尝试使用 Apache Hive 技术。我们安装了 Hive 0.5 + 20，使用 CDHv3 和 Apache Hadoop(0 20 2 + 320)。CDHv3 还包含有许多其他相关工具，包括 Sqoop 和 Hue 这些在我们的架构中都标识出来了，如图 23-3 底部所示。

我们使用 Apache Sqoop 转储数据到 Hive 中，然后通过写一个 Apache OODT 包装器，来使 Hive 按照空间/时间约束查询数据，然后将结果提供给 RCMET 和其他用户(图 23-2 中间部分显示)。RCMES 集群的完整的架构如图 23-3 所示。我们有 5 台机器，包括图中所示的一个主/从配置，通过一个运行 GigE 的私人网进行连接。

23.3.3 解决这些问题我们所面临的挑战

在将数据从 MySQL 迁移到 Hive 的过程中，我们经历了在做一些简单的任务时，响应时间缓慢的问题，例如一个简单的计数 DB 查询(例如:`hive> select count(datapoint_id) from dataPoint;`)。初始化时，我们向单个表中载入了 25 亿条数据，并在机器配置信息中记录下来，Hive 对这 25 亿条记录执行计数查询用了大约 5~6 分钟，(查询完整的 68 亿条记录大约需要 15~17 分钟)。Reduce 过程也比较快(因为我们使用的是一个对于*的 count 操作，所以我们会经历一个 reduce 阶段)，但是 map 阶段需要消耗大部分的时间(大于占总执行时间的 95%)。那个时候我们的系统由 6 个系统(4 x 四核)组成，每台系统大约有 24 GB 的 RAM (所有的机器如图 23-3 所示，再加上一个从另一个集群借来的同类型的机器)。

我们试图添加更多的节点，增加 map tasktrackers(许多不同的#s)，改变 DFS 块大小(32MB、64MB、128 MB、256MB)，利用 LZ0 压缩，并改变许多其他配置变量(例如 io、sort.factor、io.sort.mb)，都没有有效地降低完成全局计数所需要的时间。但我们却发现一直存在一个高 I/O 等待节点，而无论我们执行多少任务。数据库的大小大约是 200GB，而使用 MySQL 对 25 亿和 67 亿行数据执行计数只需要花几秒钟时间。

Hive 社区成员加入到了我们公司，为我们提供了新的视角，期间 HDFS 读取速度提高到大约 60 MB / 秒，对比本地磁盘读取速度是 1 GB /秒，这当然还取决于网络速度和 namenode 的负荷情况。社区成员提出的建议是，我们在 Hadoop 任务中大约需要 16 个 Mapper 才能和一个本地非 Hadoop 任务的 I/O 性能相当。此外，Hive 社区成员建议我们通过减少每个 Mapper 处理的分割大小(输入大小)来增加总体 Mapper 的数量(也就是增加并发量)，并指出我们应该检查以下参数：`mapred.min.split.size`、`mapred.max.split.size`、`mapred.min.split.size.per.rack` 和 `mapred.min.split.size.per.node`，并建议这些参数值应设置为 64 MB。最后，社区建议我们查看一个基准计算过程，也就是通过使用 `count(1)`而不是 `count(datapoint_id)`来计算行数，因为没有列引用意味着没有序列化和反序列化的过程，所以因为前者更快，例如，如果用户的表是 RCFile 格式存储的。(译者注：原文是后者更快，实际上后者是需要序列化和反序列化过程的。)

基于上述反馈，我们可以为 RCMES 的 Hive 集群基于一个计数基准查询进行调优，并

在规定的响应时间内返回一个计数查询，最终利用上述资源，可以在 15 秒内从 RCMET 中对数十亿行数据进行空间/时间查询，这使 Hive 对于我们的系统架构而言，成为一个可行的和绝佳的选择。

我们已经描述了在 JPL RCMES 系统中使用 Apache Hive 的情况。我们在一个案例研究中描述了我们想要通过 Hive 探索云计算技术并替代 MySQL，并从配置需求上使它的规模水平可以存储数百亿行数据，并有弹性地摧毁和重建存储于其中的数据。

Hive 很好地满足了我们的系统需求，而我们正积极寻找更多的方法将其集成到 RCMES 系统中。

23.4 Photobucket

Photobucket 是当前因特网上最大的专业网上相簿服务公司。其在 2003 年由 Alex Welch 和 Darren Crystal 创立，随后 Photobucket 很快成为互联网上最流行的网站之一，并吸引了超过一亿名用户和数十亿的存储和共享媒体。用户和系统数据分布在成百上千个 MySQL 实例上、成千上万个 Web 服务器上 and PB 级别的文件系统上。

23.4.1 Photobucket 公司的大数据应用情况

在 2008 年之前，Photobucket 还没有专门的内部分析系统。业务用户提出的问题横跨数百台 MySQL 实例并最终在 Excel 中手动聚合。

在 2008 年，Photobucket 首次开始着手实施数据仓库建设，致力于解决由一个快速增长的公司所带来的日益复杂的数据处理问题。

第一次迭代的数据仓库是一个开源的系统，包括一个 Java SQL 优化器和一组底层的 PostGreSQL 数据库。这个系统直到 2009 年都工作完好，但是其架构上的缺陷很快明显凸现。工作数据集迅速变得比实际可提供的内存大，再加上在 PostgreSQL 节点上重新对数据进行划分非常之困难，导致我们不得不对集群进行扩大。

在 2009 年，我们开始调查系统，使我们能够随着数据量的不断增长不断地向外扩展，使之仍然能够满足我们与业务用户签订的服务等级协议 (SLA)。Hadoop 迅速成为消费和分析每日由系统生成的 TB 级别的数据最受欢迎的工具，但是阻碍全面使用的一个负面因素就是对于简单的 ad-hoc 查询都要编写复杂的 MapReduce 程序。值得庆幸的是，Facebook 几周后开源的 Hive 很好地破解了这个 ad-hoc 查询复杂的问题。

Hive 相对于以前的数据仓库实现具有很多优势。

关于为什么我们选择 Hadoop 和 Hive，这里列举了几个例子。

- ① 能够处理结构化和非结构化数据。
- ② 从 Flume、Scribe 或 MountableHDFS 中实时导出数据到 HDFS 中。

- ③ 可以通过 UDF 进行功能扩展,。
- ④ 一个专门为构建 OLAP 与 OLTP 的文档充分的、类 SQL 的接口。

23.4.2 Hive 所使用的硬件资源信息

对于数据节点使用 Dell R410, 4×2TB 硬盘, 24GB RAM; 对于管理节点使用 Dell R610, 2×146GB (RAID 10) 硬盘和 24GB RAM。

23.4.3 Hive 提供了什么

Photobucket 公司使用 Hive 的主要目标是为业务功能、系统性能和用户行为提供答案。为了满足这些需求, 我们每晚都要通过 Flume 从数百台服务器上的 MySQL 数据库中转储来自 Web 服务器和自定义格式日志 TB 级别的数据。这些数据有助于支持整个公司许多组织, 比如行政管理、广告、客户支持、产品开发和操作, 等等。对于历史数据, 我们保持所有 MySQL 在每月的第一天创建的所有的数据作为分区数据并保留 30 天以上的日志文件。Photobucket 使用一个定制的 ETL 框架来将 MySQL 数据库中数据迁移到 Hive 中。使用 Flume 将日志文件数据写入到 HDFS 中并按照预定的 Hive 流程进行处理。

23.4.4 Hive 支持的用户有哪些

行政管理依赖于使用 Hadoop 提供一般业务健康状况的报告。Hive 允许我们解析结构化数据库数据和非结构化的点击流数据, 以及业务所涉及的数据格式进行读取。

广告业务使用 Hive 筛选历史数据来对广告目标进行预测和定义配额。产品开发无疑是该组织中产生最大数量的特定的查询的用户了。对于任何用户群, 时间间隔变化或随时间而变化。Hive 是很重要的, 因为它允许我们通过对在当前和历史数据中运行 A/B 测试来判断在一个快速变化的用户环境中新产品的相关特性。

在 Photobucket 公司中, 为我们的用户提供一流的系统是最重要的目标。从操作的角度来看, Hive 被用来汇总生成跨多个维度的数据。在公司里知道最流行的媒体、用户、参考域是非常重要的。控制费用对于任何组织都是重要的。一个用户可以快速消耗大量的系统资源, 并显著增加每月的支出。Hive 可以用于识别和分析出这样的恶意用户, 以确定哪些是符合我们的服务条款, 而哪些是不符合的。也可以使用 Hive 对一些操作运行 A/B 测试来定义新的硬件需求和生成 ROI 计算。Hive 将用户从底层 MapReduce 代码解放出来的能力意味着可以在几个小时或几天内就可以获得答案, 而不是之前的数周。

23.5 SimpleReach

——Eric Lubow

在 SimpleReach 中, 我们使用 Cassandra 来存储我们所有的社交网络产生的原始数据。

行的键的格式是一个账号 ID（其也是 MongoDB 的 ObjectId）和一个内容元素 ID（被跟踪的内容元素的 URL 链接的 MD5 哈希值），这两者间使用下划线进行分割，结果集中的数据也是按照这个分隔符进行划分的。这行中的列是类似于如下展示的混合在一起的一组列：

```
4e87f81ca782f340420000a_8c825814de0ac34bb9103e2193a5b824
=> (column=meta:published-at, value=1330979750000, timestamp=
1338919372934628)
=> (column=hour:1338876000000_digg-diggs, value=84, timestamp=
1338879756209142)
=> (column=hour:1338865200000_googleplus-total, value=12, timestamp=
1338869007737888)
```

为了能够访问这些组合列，我们需要知道列名对应的十六进制键的值。在我们的例子中，也就是我们需要执行列名(meta:'published-at')的十六进制的键值。这个十六进制键和值的形式如下：

```
00046D65746100000C7075626C69736865642D617400 =meta:published-at
```

一旦将列名转换成十六进制格式，Hive 查询就可以对之进行处理了。查询语句的第一部分是 LEFT SEMI JOIN，其用于模拟一个子查询 SQL。后面所有的使用 SUBSTR 和 INSTR 的引用都是来处理不同情况的组合列的。因为我们已经知道“hour:*”列（例如，SUBSTR(r.column_name, 10, 13)）的第 10~第 23 个字符是时间戳，所以我们可以将其截取出来并作为返回值返回，或者用作其他对比。INSTR 用于对比列名并保证返回的结果集在输出中总是位于相同位置的相同列。作为 Ruby 函数的一部分的 SUBSTR 也用于对比。SUBSTR 返回值是一个以毫秒表示的时间戳（long 型的），start_date 和 end_date 同样是这样的以毫秒表示的时间戳。这意味着传入的值可以作为列名的一部分进行匹配。

这个查询的目的是将数据从 Cassandra 中导出成 CVS 文件，最终为我们的出版商提供聚合后的数据。其是通过我们 Rails 栈中的一个离线处理任务完成的。具有一个完整的 CSV 文件意味着 Hive 查询中必须要包含有所有的使用到的列的列名（这意味着我们需要在没有数据的地方补上数据）。我们可以通过使用 CASE 语句将我们的宽行转换成固定列的表。

如下是处理 CSV 文件的 HiveQL 语句：

```
SELECT CAST(SUBSTR(r.column_name, 10, 13) AS BIGINT) AS epoch,
SPLIT(r.row_key, '_')[0] AS account_id,
SPLIT(r.row_key, '_')[1] AS id,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'pageviews-total') > 0
THEN r.value ELSE '0' END AS INT)) AS pageviews,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'digg-digg') > 0
THEN r.value ELSE '0' END AS INT)) AS digg,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'digg-referrer') > 0
```

```

THEN r.value ELSE '0' END AS INT)) AS digg_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'delicious-total') > 0
THEN r.value ELSE '0' END AS INT)) AS delicious,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'delicious-referrer') > 0
THEN r.value ELSE '0' END AS INT)) AS delicious_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'googleplus-total') > 0
THEN r.value ELSE '0' END AS INT)) AS google_plus,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'googleplus-referrer') > 0
THEN r.value ELSE '0' END AS INT)) AS google_plus_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'facebook-total') > 0
THEN r.value ELSE '0' END AS INT)) AS fb_total,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'facebook-referrer') > 0
THEN r.value ELSE '0' END AS INT)) AS fb_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'twitter-tweet') > 0
THEN r.value ELSE '0' END AS INT)) AS tweets,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'twitter-referrer') > 0
THEN r.value ELSE '0' END AS INT)) AS twitter_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'linkedin-share') > 0
THEN r.value ELSE '0' END AS INT)) AS linkedin,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'linkedin-referrer') > 0
THEN r.value ELSE '0' END AS INT)) AS linkedin_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'stumbleupon-total') > 0
THEN r.value ELSE '0' END AS INT)) AS stumble_total,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'stumbleupon-referrer') > 0
THEN r.value ELSE '0' END AS INT)) AS stumble_ref,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'social-actions') > 0
THEN r.value ELSE '0' END AS INT)) AS social_actions,
SUM(CAST(CASE WHEN INSTR(r.column_name, 'referrer-social') > 0
THEN r.value ELSE '0' END AS INT)) AS social_ref,
MAX(CAST(CASE WHEN INSTR(r.column_name, 'score-realtime') > 0
THEN r.value ELSE '0.0' END AS DOUBLE)) AS score_rt
FROM content_social_delta r
LEFT SEMI JOIN (SELECT row_key
FROM content
WHERE HEX(column_name) = '00046D65746100000C7075626C69736865642D617400'
AND CAST(value AS BIGINT) >= #{start_date}
AND CAST(value AS BIGINT) <= #{end_date}
) c ON c.row_key = SPLIT(r.row_key, '_')[1]
WHERE INSTR(r.column_name, 'hour') > 0
AND CAST(SUBSTR(r.column_name, 10, 13) AS BIGINT) >= #{start_date}
AND CAST(SUBSTR(r.column_name, 10, 13) AS BIGINT) <= #{end_date}
GROUP BY CAST(SUBSTR(r.column_name, 10, 13) AS BIGINT),
SPLIT(r.row_key, '_')[0],
SPLIT(r.row_key, '_')[1]

```

这个查询的输出是以逗号分隔的文件（CSV 文件），内容如下面例子所示（为清晰展示对输出中一些行进行了转行并增加了空行进行分隔）：

```
epoch,account_id,id,pageviews,digg,digg_ref,delicious,delicious_ref,
```

```
google_plus,google_plus_ref,fb_total,fb_ref,tweets,twitter_ref,  
linkedin,linkedin_ref,stumble_total,stumble_ref,social_actions,social_  
ref,score_rt
```

```
1337212800000,4eb331eea782f32acc000002,eaff81bd10a527f589f45c186662230e,  
39,0,0,0,0,0,0,0,2,0,20,0,0,0,0,0,22,0
```

```
1337212800000,4f63ae61a782f327ce000007,940fd3e9d794b80012d3c7913b837dff,  
101,0,0,0,0,0,0,44,63,11,16,0,0,0,0,55,79,69.64308064
```

```
1337212800000,4f6baedda782f325f4000010,e70f7d432ad252be439bc9cf1925ad7c,  
260,0,0,0,0,0,0,8,25,15,34,0,0,0,0,23,59,57.23718477
```

```
1337216400000,4eb331eea782f32acc000002,eaff81bd10a527f589f45c186662230e,  
280,0,0,0,0,0,0,37,162,23,15,0,0,0,2,56,179,72.45877173
```

```
1337216400000,4ebd76f7a782f30c9b000014,fb8935034e7d365e88dd5be1ed44b6dd,  
11,0,0,0,0,0,0,1,1,4,0,0,0,0,0,5,29.74849901
```

23.6 Experiences and Needs from the Customer Trenches

标题：来自 Karmasphere 的视角

——Nanda Vijaydev

23.6.1 介绍

在超过 18 个月的时间里，Karmasphere 一直忙于越来越多的使用 Hadoop 的公司，这些公司都转向使用 Hive 作为分析师团队和业务团队提供服务的最优方式。本章的第一部分说明了我们在客户环境中不断反复使用 Hive 进行分析的一些实际场景的使用技术。

我们所涵盖的使用场景例子如下。

- ① 为 Hive 提供最优数据格式化类型。
- ② 分区和性能。
- ③ 使用 Hive 函数（包括正则、Explode 函数和连词）进行文本分析。

随着我们一直合作的公司计划并生产中使用时，他们一直在寻找一些增强功能，使基于 Hive 获取 Hadoop 更容易使用、更富有成效、更强大，而且可以让他们的组织中更多的人进行使用。

当他们将 Hadoop 和 Hive 加入到他们现有的数据架构中后，他们还想让从 Hive 查询的结果系统化、可以进行共享并可以集成到其他数据存储、电子表格、BI 工具和报告系

统中。

特别地，这些公司有如下要求。

- ① 获取数据，检测原始格式，并创建元数据的便捷方法。
- ② 在一个集成的、多用户环境下协同工作。
- ③ 探索和迭代分析数据。
- ④ 可保存和重用路径。
- ⑤ 对数据、表和列的安全的细粒度控制，并区分访问不同的业务线数据。
- ⑥ 业务用户不需要 SQL 技能就可访问并进行分析。
- ⑦ 调度查询，并将生成的结果自动导出到非 Hadoop 的数据存储中。
- ⑧ 与 Microsoft Excel、Tableau、Spotfire 以及其他电子表格、报告系统、仪表板、BI 工具进行集成。
- ⑨ 可以管理基于 Hive 的功能，包括进行查询、结果输出、可视化以及 Hive 的标准组件，如 UDF 和 SerDe。

23.6.2 Customer Trenches 的用例

1. Customer trenches #1: 为 Hive 优化存储格式

许多 Hive 用户反复反馈的一个问题就是 Hive 中的数据使用什么存储格式进行存储以及如何使用这些数据。

Hive 本身内置可以支持许多种数据格式，但有一些自定义的专有格式就不支持了。有一些数据格式支持为用户解决如何从一个行数据中提取出独立的部份。有时候，写一个标准的 HiveSerDe 来支持一个自定义的数据格式是最优方法。而在其他情况下，使用现有的 Hive 分隔符和 Hive UDF 可能是最方便的解决方案。我们合作地使用 Hadoop 来提供个性化服务，并使用 Hive 对多个输入数据流进行分析。公司的一个有代表性的案例就是：他们收到的是来自他们的一个日志数据提供者的格式，而这种格式不能轻易地分裂成列。他们试图想出一个办法使得无需编写一个自定义 SerDe 就可以解析数据并运行查询。

数据包含顶层的头信息和底层的多个详细信息。详细信息部分是一个嵌套在顶级对象中的 JSON 对象，类似于如下这样的数据集：

```
{ "top" : [
  {"table":"user",
   "data":{
     "name":"John Doe","userid":"2036586","age":"74","code":"297994",
    "status":1}},
```

```

{"table":"user",
  "data":{
    "name":"Mary Ann","userid":"14294734","age":"64","code":"142798",
    "status":1}},
{"table":"user",
  "data":{
    "name":"Carl Smith","userid":"13998600","age":"36","code":"32866",
    "status":1}},
{"table":"user",
  "data":{
    "name":"Anil Kumar":"2614012","age":"69","code":"208672", "status":1}},
{"table":"user",
  "data":{
    "name":"Kim Lee","userid":"10471190","age":"53","code":"79365",
    "status":1}}
}]

```

与客户交谈后，我们意识到他们感兴趣的是上面的示例中“data”标签下的详细字段信息。为帮助他们解决这个问题，我们使用 Hive 中自带的函数 `get_json_object`，使用方法如下。

第一步是创建一个表，其使用的是样本数据：

```

CREATE TABLE user (line string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION 'hdfs://hostname/user/uname/tablefolder/'

```

然后使用 Hive 等功能得到 JSON 对象，我们可以得到的嵌套 JSON 元素并使用 UDF 对其进行解析：

```

SELECT get_json_object(col0, '$.name') as name, get_json_object(col0,
'$.userid') as uid,
  get_json_object(col0, '$.age') as age, get_json_object(col0, '$.code') as
code,
  get_json_object(col0, '$.status') as status
FROM
  (SELECT get_json_object(user.line, '$.data') as col0
  FROM user
  WHERE get_json_object(user.line, '$.data') is not null) temp;

```

查询详细信息如下。

- a. 在内部查询中提取以‘data’作为标识嵌套的 JSON 对象，并将其取别名为 `col0`。
- b. 然后将 JSON 对象分成适当的列并使用它们的标记名作为对应的列别名。

查询结果如下，这是一个 CSV 文件，其中第一行是字段名称：

```
"name","uid","age","code","status"
```

```
"John Doe", "2036586", "74", "297994", "1"
"Mary Ann", "14294734", "64", "142798", "1"
"Carl Smith", "13998600", "36", "32866", "1"
"Kim Lee", "10471190", "53", "79365", "1"
```

2. Customer trenches #2: 分区和性能

使用分区来保存通过数据流或定期添加到 Hadoop 的数据是一个最近我们看到多次的用例，而这是一个利用 Hadoop 和 Hive 来分析各种快速添加进来的数据集强大而非非常有价值的方式。Web、应用、产品和传感器日志这些数据，只是 Hive 用户经常需要 ad-hoc、重复执行和预定查询的数据。

Hive 分区在正确设置后，可以允许用户查询仅在特定的分区下的数据，从而将极大地提高性能。当为表建立分区时，文件应该位于类似如下例子中给出的目录下：

```
hdfs://user/uname/folder/"yr"=2012/"mon"=01/"day"=01/file1, file2,
file3
                                     /"yr"=2012/"mon"=01/"day"=02/file4, file5
                                     .....
                                     /"yr"=2012/"mon"=05/"day"=30/file100, file101
```

通过上述目录结构，我们可以看到表可以设置年、月和日来设置分区。查询的时候可以使用 yr、mon、和 day 作为过滤字段，同时也可以限制在特定的查询时间访问特定的值下面的数据。我们可以观察下路径中文件夹的名称，分区的文件夹名称都是如 yr=、mon=和 day=这样的标识的。

在和一个高科技公司合作时，我们发现他们的文件夹没有这个明确的分区命名，而且他们不能改变他们现有的目录结构。但他们仍然希望受益于分区。他们的样品目录结构如下所示：

```
hdfs://user/uname/folder/2012/01/01/file1, file2, file3
                               /2012/01/02/file4, file5
                               .....
                               /2012/05/30/file100, file101
```

在这种情况下，我们仍然可以通过使用 ALTER table 语句显式地添加分区并为表添加绝对路径位置。一个简单的外部脚本可以读取目录并为 ALTER TABLE 语句中添加 yr=、mon=、day=这样的信息并提供对应的有效的具体的文件夹名称(如 yr=2012, mon=01, ...)。脚本的输出是一组使用具体的目录结构的 Hive SQL 语句，而且保存在一个简单的文本文件中。

```
ALTER TABLE tablename
ADD PARTITION (yr=2012, mon=01, day=01) location '/user/uname/folder/
2012/01/01/';

ALTER TABLE tablename
```



```
ADD PARTITION (yr=2012, mon=01, day=02) location '/user/uname/folder/
2012/01/02/';
...
ALTER TABLE tablename
ADD PARTITION (yr=2012, mon=05, day=30) location '/user/uname/folder/
2012/05/30/';
```

当在 Hive 中执行这些语句时，指定的目录下的数据就会出现使用 ALTER TABLE 语句创建并定义的逻辑分区中。



提示

用户应该确保表是通过 PARTITIONED BY 语句为年、月和日创建分区字段的。

3. Customer trenches #3: 使用 Regex、Lateral View Explode、Ngram 和其他一些 UDF 进行文本分析

许多与我们合作的公司都有文本分析的场景，包括简单到复杂的情况。理解和使用 Hive regex 函数、范式和其他字符串处理函数可以解决大量的此类使用场景。

一个与我们合作的大型制造业客户有很多机器生成压缩文本数据存储到了 Hadoop 中。这个数据的格式如下。

- ① 每个文件中具有多行数据，而一个按照时间分区的数据桶内包含有许多这样的文件。
- ② 每一行数据都有许多按照 /r/n (回车和换行) 进行划分的列。
- ③ 每列数据的形式是一个“名称：值”对。

用例的要求如下。

- ① 读取每一行数据并将每列转换成“名称-值”对。
- ② 在特定的列中，进行词频统计和单词模式分析来分析关键词和特定的消息内容。

下面的示例展示了这个客户的样本数据资料（其中省略了一些文本内容）：

```
name:Mercury\r\ndescription:Mercury is the god of commerce, ... \r\ntype:
Rocky planet
name:Venus\r\ndescription:Venus is the goddess of love... \r\ntype:Rocky
planet
name:Earch\r\ndescription:Earth is the only planet ... \r\ntype:Rocky
planet
name:Mars\r\ndescription: Mars is the god of War... \r\ntype:Rocky planet
name:Jupiter\r\ndescription:Jupiter is the King of the Gods... \r\ntype:
Gas planet
name:Saturn\r\ndescription:Saturn is the god of agriculture... \r\ntype:
Gas planet
```

```
name:Uranus\r\ndescription:Uranus is the God of the Heavens...\r\ntype:
Gas planet
name:Neptune\r\ndescription:Neptune was the god of the Sea...\r\ntype:Gas
planet
```

数据具有如下几方面特点。

- ① 行星的名字和他们的包含类型的描述信息。
- ② 每一行的数据是由一个分隔符分隔。
- ③ 在每一行有 3 个部分，包括“名称”、“描述”和“类型”，按照/r/n进行字段划分。
- ④ 其中“描述”是一个大的文本内容。

第一步是使用此示例数据创建初始表：

```
CREATE TABLE planets (col0 string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION 'hdfs://hostname/user/uname/planets/'
```

接下来，我们运行一系列的查询，从一个使用了函数的简单的查询开始。需要注意的是用几种不同的方式编写的查询来满足相同的需求。如下查询语句的目的是演示 Hive 对文本解析一些关键功能。

首先，我们使用一个 split 函数来将数据划分的不同列保存到数组中：

```
SELECT split(col0, '(\\r\\n)') AS splits FROM planets;
```

接下来，我们 LATERAL VIEW EXPLODE 函数来将划分（也就是这个数组）进行展开。这个查询的结果将是每行都是一个“名称-值”对。我们只选择那些以‘desc’开头的行。LTRIM 这个函数是用来去除左端的空白字符的。

```
SELECT ltrim(splits) AS pairs FROM planets
LATERAL VIEW EXPLODE(split(col0, '(\\r\\n)')) col0 AS splits
WHERE ltrim(splits) LIKE 'desc%'
```

现在我们描述的信息转换成了“名称-值”对，并选择有值的数据。这可以以不同的方式完成的。我们使用根据“:”进行分割并选择“值”那部分数据：

```
SELECT (split(pairs, ':'))[1] AS txtval FROM (
SELECT ltrim(splits) AS pairs FROM planets
LATERAL VIEW EXPLODE(split(col0, '(\\r\\n)')) col0 AS splits
WHERE ltrim(splits) LIKE 'desc%')tmp1;
```

需要注意的是对于内部查询我们使用了临时标识符 tmp1。当用户使用子查询的输出作为外层查询的输入时，使用别名是必须的。步骤 3 处理后，我们获取到“描述”中每一行的“值”部分的数据。

在接下来的步骤中,我们使用 `ngrams` 函数来显示行星描述名称前 10 双字母组名单词。用户也可以使用如 `context_ngram`、`find_in_set`、`regex_replace` 和其他各种各样的基于文本分析的函数:

```
SELECT ngrams(sentences(lower(txtval)), 2, 10) AS bigrams FROM (
SELECT (split(pairs, ':'))[1] AS txtval FROM (
SELECT ltrim(splits) AS pairs FROM planets
LATERAL VIEW EXPLODE(split(col0, '(\\|\\|\\|\\r\\|\\|\\|\\n)')) col0 AS splits
WHERE ltrim(splits) LIKE 'desc%') tmp1) tmp2;
```

需要注意的是,我们已经使用了像 `lower` 这样的函数来将大写字母全部转换成小写,以及使用 `sentences` 函数将文本中内容分割成单词。

关于 Hive 中文本分析函数的更多信息,可以参考第 3 章中列举的函数。

生产环境下的 Apache Hive: 快速增长的需求和能力。

Hive 会保持继续成长,正如前面所定义的使用场景所展示的。在不同行业 and 不同规模的公司都将在 Hadoop 环境中使用 Hive 而受益无穷。一个强大和积极的贡献者社区,以及由 Hadoop 领先供应商提供的重大的研发投资,都将确保 Hive 已经是 Hadoop 之上的基于 SQL 的标准了,它将会成为利用 Hadoop 为大数据分析的标准组织中基于 SQL 处理的标准。

随着公司投入大量资源和时间来理解和构建 Hive 资源,在很多情况下,我们发现他们寻找额外的能力,使他们能够建立在他们的初始使用 Hive 的基础之上,并达到更快的扩展、在他们的组织内更广泛的应用。从处理这些客户希望 Hive 进化到到下一个级别的需求中,有一套共同的需求已经出现了。这些需求包括如下几方面。

- ① 多用户环境协作。Hadoop 提供了相对于传统的 RDBMS 的新的分析类别,在计算能力和成本上都具有优势。使用 Hadoop,组织机构可以将数据和人员进行分离,可以在他们可获取的每个字节的数据上执行分析,这些都通过一种方式,可以使他们能够与其他个体、团队、组织和系统分享他们的查询结果和见解。这个模型意味着为用户提供深入理解需要合作发现这些不同的数据集,再分享见解和整个组织中基于 Hive 分析的可用性。
- ② 增强生产力。Hive 的当前的实现提供了一个在 Hadoop 之上的系列批处理环境。这意味着,一旦用户向 Hadoop 集群中提交一个查询作业,他们必须等待查询完成之后才能向集群提交并执行另一个查询。这可以限制用户的生产力。企业采用 Hive 的一个主要原因是,它使他们的 SQL 技术数据专业人员可以更快和更容易使用 Hadoop。这些用户通常熟悉 SQL 编辑工具和 BI 产品。他们正在寻找类似的生产力环境例如增强语法高亮、代码自动完成。
- ③ Hive 资产管理。麦肯锡近期的一份报告通过他们的数据预测了缺乏熟练的工人,可以显著降低使组织牟利。像 Hive 这样的技术通过允许人们在 Hadoop 可以用 SQL 技能

进行分析来解决技能短缺问题。然而，组织意识到仅仅让他们的用户使用 Hive 是不够的。他们需要能够管理 Hive 资产，如查询语句（包含历史操作和版本信息）、众多的 UDF、SerDe 等，可以在以后进行分享和重用。组织想要构建这个 Hive 资产的知识存储库，而且用户可以很容易搜索到。

④ 为先进的分析技术对 Hive 进行扩展。许多公司正在寻找在 Hadoop 中重建他们在传统的 RDBMS 中分析的系统。虽然并不是 SQL 环境的所有功能都很容易转化为 Hive 函数，其中一些是因为数据存储的固有局限性，有一些高级分析功能（像 RANK，等等），这些 Hadoop 是可以进行处理的。此外，组织使用传统工具如 SAS 和 SPSS 花了巨大的资源和时间在构建分析模型上，并希望能够在 Hadoop 通过 Hive 查询更好地使用这些模型。

⑤ SQL 技能外扩展 Hive。因为 Hadoop 在组织中积蓄了大量优势，并成为 IT 基础设施之上的关键的数据处理和分析架构，这在具有不同的技能和能力的用户当中很流行。尽管 Hive 很容易适应具有 SQL 技能的用户，其他的懂得 SQL 并不多的用户也在寻求可以在基于 Hadoop 的 Hive 上像在传统的 BI 工具中通过拖拖拽拽就可以执行分析的功能。能够在 Hive 之上支持交互式表单，能够通过简单的基于 Web 的形式提示用户提供的列值是一种常见的功能。

⑥ 数据探索能力。传统的数据库技术提供数据浏览功能。例如，一个用户可以查看一个整数列的最小值，最大值。此外，用户还可以通过可视化的方式查看这些列，在他们执行分析数据之前理解数据分布。因为 Hadoop 存储了数百 TB 的数据，并且通常是 PB 级别的，对于特定的使用场景，顾客需要类似的功能。

⑦ 调度和操作 Hive 查询。当公司使用具有 Hadoop 的 Hive 发现了一些深刻见解时，他们也在寻求实施这些见解和安排在一个正则区间运行这些查询。虽然目前已经有了可用的开源替代方案，但当公司也想管理 Hive 查询的输出时就会功亏一篑。例如，将结果集转移到一个传统的 RDBMS 系统或 BI 堆栈。管理特定的用例，公司通常必须手工串起各种不同的开源工具或依靠可怜的 JDBC 连接器进行执行。

⑧ 关于 Karmasphere。Karmasphere 是一家软件公司，位于加州硅谷，专注于帮助分析师团队和业务用户使用 Hadoop 的大数据分析能力。他们的旗舰产品，Karmasphere 2.0，是基于 Apache Hive 的，并扩展实现了多用户图形化工作空间。

- a. 重用标准的 Hive 表、SerDe 和 UDF。
- b. 为分析师团队和业务用户提供社交化的、基于项目的大数据分析服务。
- c. 可以方便和其他集群进行数据整合。
- d. 基于启发式的识别和提供多种流行的数据存储格式来创建表。
- e. 可视化的和迭代式的数据探索和分析。

- f. 图形化的对基于 Hive 的数据集进行分析探索。
- g. 可以共享和调度查询以及结果，并提供可视化操作和展示。
- h. 和传统表格、报表、仪表盘、BI 工具很容易进行集成整合。

图 23-4 展示了 Karmasphere 2.0 的基于 Hive 的大数据分析环境的操作界面截图。

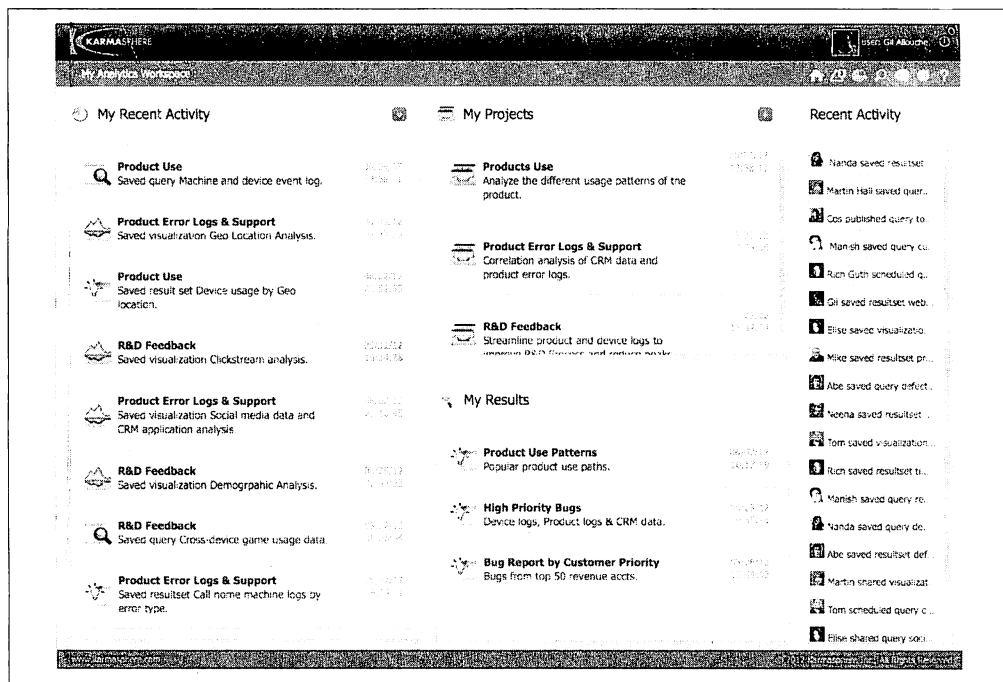


图 23-4 Karmasphere 2.0 的屏幕截图

⑨ Hive 特性调查。对于这些需求我们期望能得到反馈并最终能在快速发展的 Hive 社区中进行分享。如果你有兴趣看看别人的想法的话，那么请访问如下链接：<http://karmasphere.com/hive-features-survey.html>。

术语词汇表

亚马逊弹性 MapReduce (EMR)

亚马逊的 EMR 是基于亚马逊 EC2 (弹性计算云) 的托管 Hadoop 服务。

Avro

Avro 是一个新的序列化格式，其用于解决一些其他序列化格式演变过程中发现的常见问题。使用它的一些好处是其具有丰富的数据结构、快速的二进制格式，支持远程过程调用，而且内置模式演化。

Bash

Bash 是 Linux 和 Mac OS X 系统的默认命令行交互 shell。

S3 中的数据桶

数据桶是使用 S3 时用户可以具有和管理的最顶层容器的术语。一个用户可以具有很多的数据桶，其类似于物理硬盘的 root 根目录。

命令行交互界面 (CLI)

命令行交互界面 (也就是 CLI) 是指可以执行 Hive 语句“脚本”并和用户输入信息进行交互的命令行界面。

数据仓库

数据仓库指用于报告、趋势等分析的一组结构化数据组成的库。数据仓库可以提供数据批处理，是离线的，而不是像电子商务这样提供实时响应能力的系统。

Derby

Derby 是一个轻量级的 SQL 数据库，其可以嵌入到 Java 应用程序中。其运行在相同的进程中并将数据存储到本地文件中。Hive 元数据存储中默认使用 Derby 作为元数据库。

可以访问 <http://db.apache.org/derby/> 获得更多信息。

动态分区

动态分区是 HiveQL 对于 SQL 的一个扩展。使用动态分区可以允许用户将查询结果插入到表分区中，而分区字段是一个或者多个不同的分区列值，具体的分区值将由查询结果动态地生成。使用动态分区可以非常方便地将查询结果写入到新表的大量分区中，而无需为每个分区列值都写一个单独的查询语句。

Ephemeral Storage 临时性存储

对于虚拟亚马逊 EC2 集群的节点，节点上的硬盘存储空间就被称为临时性存储。这是因为和普通的物理存储集群不同，当这类集群关闭后这些存储数据就会被清除掉。因此，当使用 EC2 集群，例如亚马逊弹性 MapReduce 集群时，要注意将重要的数据备份到 S3 上。

外部表

外部表指使用不在 Hive 控制范围内的存储路径和内容的表。使用外部表会比较方便将数据和其他工具共享，但是需要其他的处理过程来管理数据的生命周期。也就是说，当创建了一个外部表时，Hive 并没有创建这个外部存储路径（对于分区表的话还应包括分区路径文件夹），同样地，当删除外部表后，并不会删除外部路径下的数据。

Hadoop 分布式文件系统（HDFS）

Hadoop 分布式文件系统（HDFS）是一个用于数据存储的分布式的、弹性的文件系统。HDFS 被优化为可以快速扫描硬盘上大的连续的数据块。集群中的分布式提供了数据存储的横向扩展。HDFS 文件的数据块在集群内会被复制成多份（默认情况下是 3 份）来防止当磁盘坏掉或者整个服务器坏掉时的数据丢失。

HBase

HBase 是使用 HDFS 作为持久化存储表数据的 NoSQL 数据库。HBase 是列式的键-值对存储的设计，用于提供传统的快速响应查询和行级别的数据更新和插入。列式存储意味着磁盘上的数据存储是按照多组列够成的，也就是列族，而不是按照行来组织的。这个特性使得可以快速查询列的子集数据。键-值对存储意味着行是按照一个唯一键和整行所对应的值来获取的。HBase 并没有提供一种 SQL 方言，但是可以使用 Hive 来查询 HBase 表。

Hive

Hive 是一个数据仓库工具。Hive 对存储在 HDFS、HBase 表或者其他存储上的数据进行了抽象，形成了表。Hive 查询语言（HQL）是结构化查询语言（SQL）的方言。

Hive 查询语言（HQL）

Hive 查询语言（HQL）是 Hive 自己的结构化查询语言（SQL）的一种方言。通常缩写

为 HiveQL 或者 HQL。

输入文件格式 (Input Format)

输入文件格式决定了输入数据流 (通常是来自于文件的) 是如何分割成记录的。*SerDe* 用于将记录分割成列。用户自定义的输入文件格式可以在创建表时通过 `INPUTFORMAT` 语句来进行指定。默认的输入文件格式也就是默认的 `STORED AS TEXTFILE` 语句实际是就是 `org.apache.hadoop.mapreduce.lib.input.TextInputFormat` 的简写形式。输出文件格式 (*Output Format*) 是类似的。

JDBC

Java 数据库连接 API (JDBC) 提供了使用 Java 代码访问包括 Hive 的 *SQL* 系统的接口。

Job 任务

在 *Hadoop* 上下文中, 一个 *Job* 就是一个提交给 *MapReduce* 的独立的工作流。其中包含有需要执行一个完整的计算, 从读取输入到生成输出的完整过程。*MapReduce* 框架中的 *JobTracker* 会将其分解成一个或者多个可以在集群中分布式执行的 *task* 任务。

JobTracker

JobTracker 是使用 *Hadoop* 的 *MapReduce* 的所有任务 (*Job*) 的最顶层控制器。*JobTracker* 接收提交上来的 *Job*, 决定执行什么 *task* 以及分发到哪里去执行, 监控他们的执行过程, 并且在需要的时候重跑失败的 *task*, 其还提供了一个网页控制台界面用户监控 *Job* 和 *task* 的执行过程, 查看执行日志, 等等。

任务流 (Job Flow)

任务流是 *亚马逊弹性 MapReduce (EMR)* 中使用到的一个术语, 其表示在一个为特定目的而创建的临时 *EMR* 集群上执行的一系列连续的 *job* 任务。

JSON

JSON 即 JavaScript Object Notation, 它是一种轻量级的数据交换格式, 非常适合于服务器与 JavaScript 的交互, 因此常用于网络应用程序。

Map 阶段

Map 阶段是指 *MapReduce* 处理过程中进行 Map 操作的那个阶段, 在这个阶段输入的键-值对集合会转换成一个新的键-值对集。对于每个输入键-值对, 可以产生零到多个输出键-值对。输入和输出键以及输入和输出值可能是完全不同的。

MapR

MapR 是一个商业版本的 *Hadoop*, *MapR 文件系统 (MapR-FS)* 代替了 *HDFS*。*MapR-FS* 是一个高性能的分布式文件系统。

MapReduce

MapReduce 是 Google 提出的一个软件架构，用于大规模数据集的并行运算。其将计算过程分为两个过程映射（Map）过程和化简（Reduce）过程，map 过程用来把一组键-值对映射成一组新的键-值对，并通过指定并发的 Reduce 过程，来保证所有映射的键-值对中的每一个共享相同的键组。MapReduce 通过把对数据集的大规模操作分发到网络上的每个节点实现可靠性。每个节点会周期性的把完成的工作和状态的更新报告回来。如果一个节点保持沉默超过一个预设的时间间隔，主节点（类同 Google 档案系统中的主服务器）记录下这个节点状态为死亡，并把分配给这个节点的数据发到别的节点。每个操作使用命名文件的不可分割操作以确保不会发生并行线程间的冲突。当文件被改名的时候，系统可能会把他们复制到任务名以外的另一个名字上去。

元数据存储（Metastore）

元数据存储是指存储如表结构信息这样的“元数据”信息的服务。Hive 需要使用元数据存储服务才能够使用。默认情况下，Hive 使用的元数据存储是内置的 Derby SQL Server 服务，其提供了有限的，单进程的 SQL 支持。生产环境下需要使用像 MySQL 这样的提供全功能的关系型数据库作为元数据存储。

NoSQL

NoSQL 指的是非关系型数据库，不支持关系型模型的数据管理、结构化查询语言、以及传统的 update 操作。这些数据存储将这些功能剔除掉了，目的是为大数据处理提供更具成本效益的可伸缩性、高可用性，等等。

ODBC

ODBC 全称是开放数据库连接 API，其为其他应用程序提供了访问 SQL 系统的接口，当然包括访问 Hive 的接口。通过来说 Java 应用程序是使用 JDBC API 进行连接的。

输出文件格式（Output Format）

输出文件格式（OutputFormat）决定了记录是如何写入到输出流中的，通常是如何写入到文件中的。一个 SerDe 处理了如何将每行记录序列化化成合适的字节流的问题。可以在创建表语句中通过 OUTPUTFORMAT 语句执行用户自定义的文件输出格式。默认的输出格式是 STORED AS TEXTFILE，其是 org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat 的简化形式。相关的可查看下输入文件格式（InputFormat）。

分区

分区是表数据的一个子集，同一个分区下的分区字段内容是相同的。在 Hive 中，和大多数支持数据分区的数据库一样，每个分区都对应着不同的物理存储路径（在 Hive 中就是表路径下的子文件夹）。使用分区有几个好处。分区字段的值在分区数据中其实是没的，这样就可以节约存储空间，同时对于使用 WHERE 语句的查询来说通过限制

特定的分区字段可以减少输入的数据量而提高执行效率，因为这样可以避免全表扫描。同样可以参考下*动态分区*。

Reduce

Reduce 过程是 MapReduce 处理过程的一部分，在这个过程中来自 map 阶段的键-值对会被进行处理。MapReduce 的一个重大的特性就是来自所有 map task 产生的键-值对，具有相同键的值会被分发到同一个 reduce task 中，因此值的集合可以根据实际情况进行“化简”。例如，一组整数可以进行相加或者求平均计算，而一组字符串可以进行去重处理，等等。

关系模型

关系模型是数据库管理系统的最常见的模型，其是基于数据组织和数据操作的逻辑模型。数据结构的声明以及如何操作数据都是由用户定义的，最常见就是使用*结构化查询语言* (SQL)。而相应的具体实现会将这些声明转换成存储、获取和操作数据等处理过程。

S3

S3 上*亚马逊网络服务* (AWS) 的分布式文件系统。在运行 *MapReduce* 任务时可以和 HDFS 结合使用或者替代 HDFS 进行使用。

SerDe

序列化器/反序列化器或简称为 SerDe，用于将记录的字段解析成列或字段，也就是反序列化过程。它也用来创建这些记录字节(也就是序列化过程)。相反，输入格式 (InputFormat) 用于将一个输入流转换为记录而输出格式 (OutputFormat) 用来将记录写到一个输出流。在创建 Hive 表时可以指定 SerDe。默认的 SerDe 支持在第 3.3 节“文本文件数据编码”中所介绍的字段分隔符，以及以及各种优化如简易解析。

结构化查询语言 (SQL)

结构化查询语言 (SQL) 是一种实现了关系模型查询和操纵数据的语言。缩写为 SQL。对于 SQL 虽然有一个经历了很多次修改的 ANSI 标准，但是所有的 SQL 方言都可以广泛地添加使用自定义扩展和改变。

Task

在 *MapReduce* 上下文中，*task* 是在单个集群节点上工作的最小单元，其作为整个 *job* 的一个处理单元。默认情况下，每个 *task* 都会启动单独的一个 JVM 进程。每个 *map* 和 *reduce* 调用都有其自身的 *task*。

Thrift

Thrift 是 Facebook 发明的 RPC 系统，其被集成到了 Hive 中。远程进程可以通过 Thrift

发送 Hive 语法到 Hive 中。

用户自定义聚合函数 (UDAF)

用户自定义聚合函数 (UDAF) 是指输入为多行 (或多行的多个字段) 而产生唯一一个“聚合的”结果数据的用户自定义函数, 例如求输入数据的行数, 计算一组值的和或平均值, 等等。这个术语简称 UDAF。请参考第 13 章“用户自定义函数”和第 13.9 节“用户自定义聚合函数”。

用户自定义函数 (UDF)

用户自定义函数 (UDF) 是指 Hive 用户用于扩展其处理操作所实现的函数。有时这个术语也包含 Hive 内置的函数, 其也通常表示那些一条输入行 (或一行数据中某个列) 并产生一条输出 (例如, 并不会改变输出记录的行数) 的函数。关于 UDF 的更多信息, 请参考第 13.9 节“用户自定义聚合函数”和第 13.10 节“用户自定义表生成函数”。

用户自定义表生成函数 (UDTF)

用户自定义表生成函数 (UDTF) 是指可以将每条记录的一个字段的值转换生成多行记录的 *用户自定义函数*。例子如 `explode` 函数, 这个函数可以将一个数组字段的值转换成多行, 而且对于 Hive v0.8.0 和更高版本, 可以将 `map` 类型的字段转换成键和值的多行数据。关于 UDTF 的详细信息, 请参考第 13 章“用户自定义函数”和第 13.9 节“用户自定义聚合函数”。

书末说明

《Hive 编程指南》书封面上的生物是一只黄边胡蜂（胡蜂属的），还有它的蜂巢。黄边胡蜂是南美的唯一的一种胡蜂，其是在欧洲移民时带到美洲的。这种胡蜂可以在整个欧洲以及亚洲大部分地区找到，其可以根据不同气候按照需要通过不同的蜂巢建造技术来进行适应。

胡蜂是一种社会性昆虫，和蜜蜂、蚂蚁是一样的。胡蜂的蜂巢里有一个蜂后，以及少量的雄性胡蜂，而大部分都是雌性的工蜂。雄蜂的使命就是和蜂后进行繁殖，其后不久它们就会死去。而雌性工蜂的作用就是建造蜂巢，搬运食物，以及照顾蜂后的卵。

胡蜂的巢和本书是一致的，因为蜂巢是由木浆形成的多层六角形格子构成的。最终形成的结果就是一个附着在短杆上的梨型巢。在寒冷地区，冬天的时候胡蜂会放弃蜂巢而转移到空的树干里甚至是人类的房屋里，蜂后和蜂卵会一直待到天气变暖为止。蜂卵孵化出新的胡蜂然后形成新的殖民地，而蜂巢会进行再次构造。本书的封面图片来自于 Johnson 的《博物学》。