获取教材和讲义 PPT 等各种课程资料请访问 http://dblab.xmu.edu.cn/node/422

=课程教材由林子雨老师根据网络资料编著=

厦门大学计算机科学系教师 林子雨 编著

http://www.cs.xmu.edu.cn/linziyu

2013 年 9 月

前言

本教程由厦门大学计算机科学系教师林子雨编著，可以作为计算机专业研究生课程《大数据技术基础》的辅助教材。

本教程的主要内容包括：大数据概述、大数据处理模型、大数据关键技术、大数据时代面临的新挑战、NoSQL 数据库、云数据库、Google Spanner、Hadoop、HDFS、HBase、MapReduce、Zookeeper、流计算、图计算和 Google Dremel 等。

本教程是林子雨通过大量阅读、收集、整理各种资料后精心制作的学习材料，与广大数据库爱好者共享。教程中的内容大部分来自网络资料和书籍，一部分是自己撰写。对于自写内容，林子雨老师拥有著作权。

本教程 PDF 文档及其全套教学 PPT 可以通过网络免费下载和使用（下载地址：http://dblab.xmu.edu.cn/node/422）。教程中可能存在一些问题，欢迎读者提出宝贵意见和建议！

本教程已经应用于厦门大学计算机科学系研究生课程《大数据技术基础》，欢迎访问 2013 班级网站 http://dblab.xmu.edu.cn/node/423。

林子雨的 E-mail 是：ziyulin@xmu.edu.cn。

林子雨的个人主页是：http://www.cs.xmu.edu.cn/linziyu。

林子雨于厦门大学海韵园

2013 年 9 月

第13章 Google Dremel

厦门大学计算机科学系教师 林子雨 编著

个人主页：http://www.cs.xmu.edu.cn/linziyu

课程网址：http://dblab.xmu.edu.cn/node/422

2013年9月

第 13 章 Google Dremel

Dremel 是一种可扩展的、交互式的实时查询系统，用于只读嵌套（nested）数据的分析。通过结合多级树状执行过程和列式数据结构，它能做到几秒内完成对万亿张表的聚合查询。系统可以扩展到成千上万的 CPU 上，满足 Google 上万用户操作 PB 级的数据，可以在 2 到 3 秒内完成 PB 级别数据的查询。在本章中，我们将描述 Dremel 的架构和实现，解释它为何是 MapReduce 计算的有力补充。此外，我们也描述了一种新的针对嵌套记录的列存储形式。

本章介绍 Dremel 的相关知识，内容要点如下：

- Dremel 概述
- Dremel 的数据模型
- 嵌套列式存储
- 查询语言
- 查询的执行

13.1Dremel 概述

13.1.1 大规模数据分析

大规模分析型数据处理，在互联网公司乃至整个行业中都已经越来越广泛。目前已经可以用廉价的存储来收集和保存海量的企业核心业务数据，但是，更重要的是，必须懂得如何让分析师和工程师便捷地利用这些数据。在数据探测、监控、在线用户支持、快速原型设计、数据管道调试以及其他任务中，交互的响应时间是一个至关重要的系统设计考虑因素。

执行大规模交互式数据分析对并行计算能力要求很高，例如，假设使用普通的硬盘，如果希望在 1 秒内读取 1TB 的压缩数据，那么就需要成千上万块硬盘。类似地，CPU 密集的查询操作也需要运行在成千上万个核上，并在数秒内完成。在 Google 公司里，大量的并行计算是使用普通 PC 组成的共享集群完成的。一个集群通常会部署大量“共享资源、产生不同负载、需要不同硬件参数”的分布式应用。对于一个分布式应用而言，某个工作任

务可能会比其他任务花费更多的时间，或者可能由于故障，或者被集群管理系统停止而永远不能完成。因此，处理好异常和故障是实现快速执行和容错的重要因素。

互联网和科学计算中的数据经常是没有关联的，因此，在这些领域，一个灵活的数据模型是十分必要的。在编程语言中使用的数据结构、分布式系统之间交换的消息、结构化文档等等，都可以用嵌套方式来很自然地描述。嵌套数据模型已经成为 Google 处理大部分结构化数据的基础，据报道，其他互联网公司也在使用这种嵌套数据模型。

13.1.2 Dremel 的特点

随着 Hadoop 的流行，大规模的数据分析系统已经越来越普及。但是，Hadoop 比较适合用于大规模数据的批量处理，而对于实时的交互式处理就有点显得力不从心，比如，Hadoop 通常无法做到让用户在 2 到 3 秒内迅速完成 PB 级别数据的查询。因此，数据分析师需要一个能将数据“玩转”的交互式系统，这样就可以非常方便快捷地浏览数据以及建立分析模型。Google 公司设计的 Dremel 就是一个能够满足这种实时交互式处理的系统，它具有以下几个主要的特点：

（1）Dremel 是一个大规模、稳定的系统。在一个 PB 级别的数据集上面，将任务缩短到秒级，无疑需要大量的并发。Google 一向是用廉价机器办大事的好手，但是，机器越多，出问题概率越大。如此大的集群规模，需要有足够的容错考虑，保证整个分析的速度不被集群中的个别慢(坏)节点所影响。

（2）Dremel 是 MapReduce 交互式查询能力不足的补充。和 MapReduce 一样，Dremel 也需要和数据运行在一起，将计算移动到数据上面，所以，它需要 GFS 这样的文件系统作为存储层。在设计之初，Dremel 并非是 MapReduce 的替代品，它只是可以执行非常快的分析，在使用的时候，常常用它来处理 MapReduce 的结果集或者用来建立分析原型。

（3）Dremel 的数据模型是嵌套(nested)的。互联网数据常常是非关系型的，这就要求 Dremel 必须有一个灵活的数据模型，这个数据模型对于获得高性能的交互式查询而言至关重要。因此，Dremel 采用了嵌套(nested)数据模型，有点类似于 Json。嵌套数据模型相对于关系模型而言具有明显的优势。对于传统的关系模型而言，不可避免地存在大量连接（Join）操作，因此，在处理如此大规模的数据的时候，往往是有心无力的。而嵌套数据模型却可以在 PB 级别数据上一展身手。

（4） Dremel 中的数据是用列式存储的。当采用列式存储时，在分析的时候就可以只

扫描需要的那部分数据，从而大大减少 CPU 和磁盘的访问量。同时，列式存储是压缩友好的，可以实现更高的压缩率，使得 CPU 和磁盘发挥最大的效能。对于关系型数据而言，在如何使用列式存储方面，我们都已经很有经验。但是，对于嵌套(nested)结构，Dremel 也通过巧妙的设计来实现列式存储，这是非常值得我们学习的。

（5） Dremel 结合了 Web 搜索 和并行 DBMS 的技术。首先，它借鉴了 Web 搜索中的“查询树”的概念，将一个相对巨大复杂的查询分割成较小、较简单的查询。大事化小，小事化了，能并发地在大量节点上跑。其次，和并行 DBMS 类似，Dremel 可以提供了一个类似 SQL 的接口，就像 Hive 和 Pig 那样。

Dremel 自从 2006 年开始就已经投入开发了，并且在 Google 公司已经有了几千用户。多种多样的 Dremel 实例被部署在 Google 公司里，每个实例拥有着数十至数千个节点。使用 Dremel 系统的例子包括：

- 分析网络文档；
- 追踪 Android 市场应用程序的安装数据；
- Google 产品的崩溃报告分析；
- Google Books 的 OCR 结果 ；
- 垃圾邮件分析；
- Google Maps 里地图部件调试；
- 管理中的 Bigtable 实例的 Tablet 迁移；
- Google 分布式构建系统中的测试结果分析；
- 数万个硬盘的磁盘 IO 统计信息；
- Google 数据中心上运行的任务的资源监控；
- Google 代码库的符号和依赖关系分析。

13.1.3 Dremel 的应用场景

为了说明交互式查询处理如何融入更广泛的数据管理领域，这里设计了一个应用场景。假想有一名 Google 的工程师 Alice，想从大量网页中提取新的信息。她运行 MapReduce 任务从输入数据中跑出数十亿条包括所有信息的记录，存储到分布式文件系统中。为了分析实验结果，她通过 Dremel 执行了几条交互式命令：

```
DEFINE TABLE t AS /path/to/data/*
```

SELECT TOP(signal1, 100), COUNT(*) FROM t

这些命令在几秒内就执行完了，之后她又做了其它一些查询来验证她的算法是否工作正常。她发现了 signal1 中有一个不正确的地方，为了更深入地考察其中的问题，她写了个 FlumeJava 程序，在之前输出的数据集上进行一些更复杂的分析计算。这一步解决后，她又创建了一个管道来不停地处理新进来的数据。此外，她还写了些 SQL 查询来聚合管道各个维度的结果输出，并把它加到了交互式的操作界面上。最后，她在一个目录里声明了她的新数据集，这样其他工程师可以找到并快速查询。

上述案例要求在查询处理器和其他数据管理工具之间互相协作。第一个组成部分是一个公用的存储层。GFS（Google File System）是公司中广泛使用的分布式存储层。GFS 使用冗余复制来保护数据不受硬盘故障影响，即使出现异常也能达到快速响应时间。对数据管理来说，一个高性能的存储层是非常重要的，它允许访问数据时不消耗太多时间在加载阶段。这个要求也导致数据库在分析型数据处理中不常被使用，因为，在 DBMS 加载数据以及执行单一查询前，可能要运行数个 MapReduce 分析任务。使用 GFS 的另外一个好处是，在文件系统中能使用标准工具便捷地操作数据，比如，迁移到另外的集群，改变访问权限，或者基于文件名定义一个数据子集的分析。

构建互相协作的数据管理组件的第二个要素是一个共享的存储格式。列式存储已经被证明适用于扁平的关系型数据，但是，使它适用于 Google 则需要转换到一个嵌套数据模型。图 13-1 展示了对嵌套数据进行列式存储的主要思想：一个嵌套字段比如 A.B.C，它的所有值被连续存储。因此，A.B.C 被读取时，不需读取 A.E、A.B.D 等等。

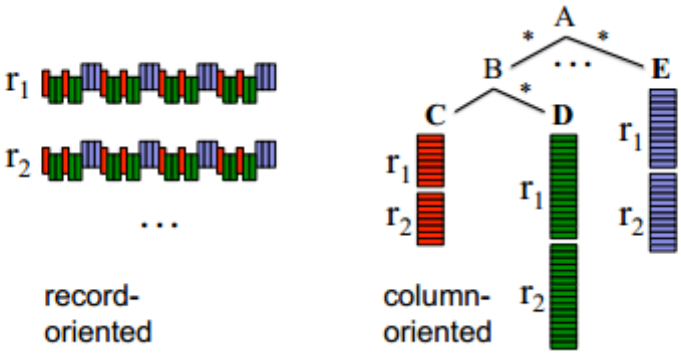

图 13-1 嵌套数据的行式存储和列式存储

13.2Dremel 数据模型

本节我们介绍 Dremel 的数据模型以及一些后续将会用到的术语。这个数据模型是基

于强类型嵌套记录的，它的抽象语法是：

π = dom | <A1 : π[*|?],…,An : π[*|?]>

π 是一个原子类型或者记录类型。在 dom 中原子类型包含整型、浮点数、字符串等等。记录则由一或多个字段组成。在记录中字段 i 标记为 Ai，以及拥有一个可选的多样性的标签。另外需要注意的是字段的类型，每个字段都属于某种类型，比如，required 表示有且仅有一个值；optional 表示“可选”，有 0 到 1 个值；repeated（*），表示“重复”，有 0 到 N 个值。其中，repeated 和 optional 类型是非常重要的，从它们身上抽象出一些重要的概念，以便用最少的代价来无损地描述出原始的数据。

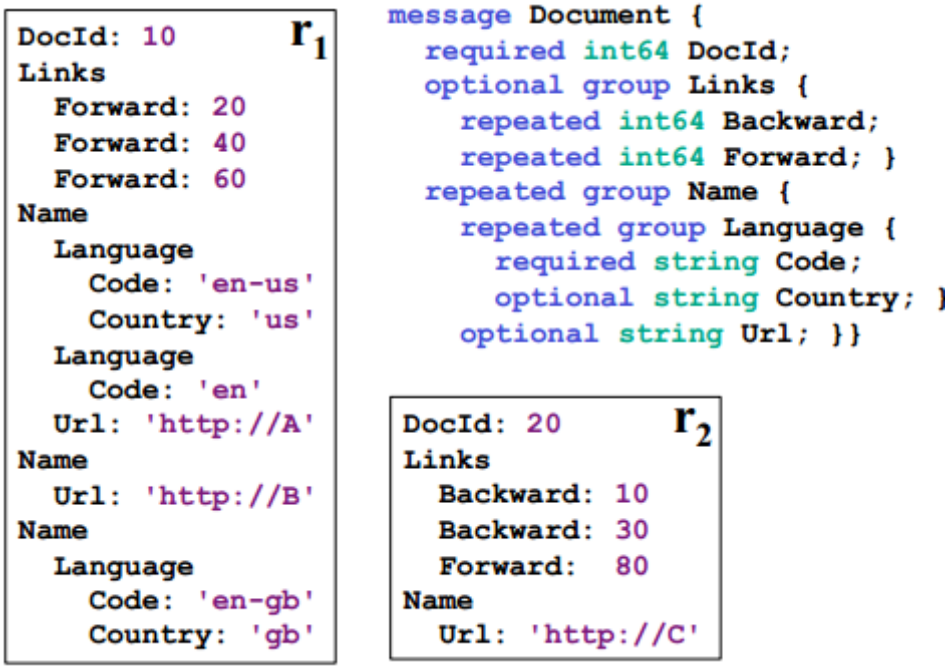

图 13-2 两个简单的嵌套记录和它们的 schema

图 13-2 描述了两个简单的嵌套记录和它们的 schema，用来表示一个网页。该 schema 定义使用了上面介绍的语法。一个网页文档拥有必需的整型 DocId 属性和可选的 Links 属性，以及包含在列表中的 Forward 和 Backward，列表中每一项代表其他网页的 DocId。一个网页文档可以有多个 Name，代表该网页所被引用的不同 URL。Name 包含一系列 Code 和 Country（可选）的组合。图 13-2 同时给出了遵循上述 schema 的两个示例记录，即 r1 和 r2。记录的结构通过缩进体现出来。在后续内容中，我们将使用这些记录的例子来解释相应的算法。Schema 中定义的字段形成了一个树状结构。一个嵌套字段的完整路径，是通过点号来表示，如 Name.Language.Code。

嵌套数据模型为 Google 的序列化、结构化数据奠定了一个平台无关的可扩展机制。

代码生成工具生成具体编程语言（如 C++、Java）的代码。跨语言的兼容性是通过对记录的标准二进制化表示来保证的，记录中的字段及相应的值被序列化后进行传输。通过这种方式，Java 写的 MapReduce 程序可以处理另外一个 C++生成的数据源中的记录。因此，如果记录采用列式存储，快速的记录重建（即从列式存储中组装出原来的记录），对于 MapReduce 和其它数据处理工具而言都是非常重要的。

13.3嵌套列式存储

如图 13-1 所示，我们的目标是连续地存储一个给定的字段的所有值来改善查询效率。在本节中，我们阐述了需要解决的问题：一个列式格式记录的无损表示、分割记录为列式存储、高效的记录装配。

13.3.1 重复深度、定义深度

让我们重新审视一下图 13-1 右边的列式存储结构，这是 Dremel 的目标，它就是要将图 13-2 中 Document 那种嵌套结构转变为列式存储结构。实现这个目标的方式多种多样，Google 信心满满地推出了它设计的最优化、最节省成本、效率最高的方法，并且引出了两个全新的概念，即重复深度和定义深度。因为 Dremel 会将记录肢解、再按字段各自集中存储，此举难免会导致数据失真，比如图 13-2 中，我们把 r1 和 r2 的 URL 列值放在一起得到 [“http://A”,"http://B","http://C"]，那么，我们怎么知道它们各自属于哪条记录、属于记录中的哪个 Name 呢？这里提出的两个概念——重复深度和定义深度，其实就是为了解决这种“失真”问题，从而实现无损表达。

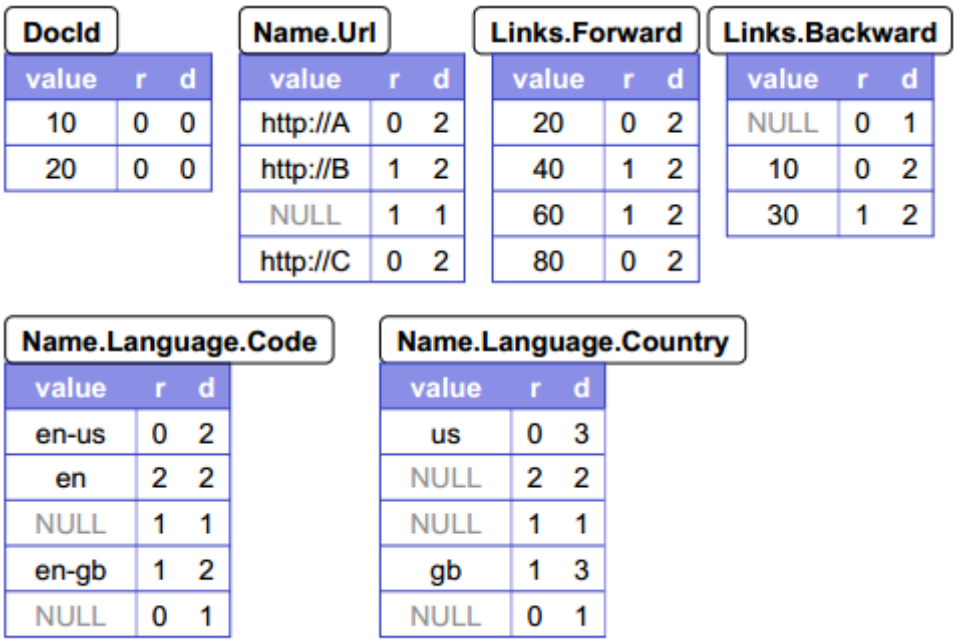

图 13-3 演示重复深度和定义深度的实例

- **重复深度（Repetition Level）**

注意在图 13-2 中的 Code 字段，可以看到它在 r1 中总共出现了 3 次。‘en-us’、‘en’在第一个 Name 中，而‘en-gb’在第三个 Name 中。为了消除这种字段值的含糊性，我们对字段值附加了“重复深度”，它可以告诉我们，在路径中的哪个重复字段重复出现了，依此来确定此值的位置。比如，在 Name.Language.Code 这个路径中，包含两个重复字段，Name 和 Language，因此，Code 字段的重复深度范围为 0 到 2，其中，0 意味着一个新记录的开始。现在让我们从上至下扫描纪录 r1，来看看重复深度的具体含义。当我们遇到‘en-us’，我们没看到任何重复的字段，也就是说，重复深度是 0。当我们遇到‘en’，字段 Language 重复了，Language 在 Name.Language.Code 中排在第 2 个级别，所以重复深度是 2。最后，当我们遇到‘en-gb’，Name 重复了（Name 后 Language 只出现过一次，因此要注意，这里的 Language 是没有重复的），Name 在 Name.Language.Code 中排在第 1 个级别，所以重复深度是 1。因此，从上至下扫描纪录 r1 以后，可以得到 r1 中 Code 的三个值（‘en-us’、‘en’和‘en-gb’）的重复深度分别是 0、2、1。

这里要注意，r1 中的第二个 Name 没有包含任何 Code 值。在重构记录时，为了确定‘en-gb’出现在第三个 Name 而不是第二个 Name 中，我们需要额外添加一个 NULL 值在‘en’和‘en-gb’之间（如图 13-3 所示）。在 Language 字段中，Code 字段是必需的值，也就是说，只要 Language 字段出现，就一定会有 Code 字段出现并且有值，所以，Code 字段值的缺失

意味着 Language 也没有定义。因此，在 r1 中的第二个 Name 中，没有 Code 字段值，就意味着没有 Language 字段值。一般来说，确定一个路径中（比如 Name.Language.Code 这条路径）有哪些字段被明确定义，需要一些额外的信息，这就需要引入“定义深度”的概念。

- **定义深度（Definition Level）**

定义深度从某种意义上来说是服务于重复深度的，因为，在重新装配记录时，只有在“仅靠重复深度无法确定字段值的位置”时（一般是字段值为 NULL 的情形），才需要去参考定义深度来确定字段值的位置。在 Dremel 系统中，所有列都是先存储来自 r1 的内容，后存储来自 r2 的内容，也就是说，对于所有的列而言，记录存储的顺序是一致的。这个顺序就像所有列值都包含的一个唯一主键，逻辑上能够将被肢解出来的列值串在一起，知道它们是否属于同一条记录，这也是保证记录被拆分之后不会失真的一个重要手段。既然顺序是十分必要的、不能失真的因素，那么，当某条记录的某一列的值为空时就，我们就不能简单地跳过，必须显式地为其存储一个 NULL 值，以保证记录顺序的完整有效。但是，仅仅 NULL 值本身所能诠释的信息是不够的，比如，记录中某个 Name.Language.Country 列为空，由于 Name 和 Language 字段都是 repeated 类型，而 Country 又是 optional 类型，所以，这种情形可能表示 Country 字段没有值，也可能表示 Language 字段没有值，这两种情况在装配算法中是需要区分处理的，不能失真，所以才需要引出定义深度，能够准确描述出这种情形。例如，我们看到 r1 没有 Backward 链接，而 Links 字段是定义了的（在级别 1），为了保护此信息，我们就需要为 Links.Backward 列添加一个 NULL 值，并设置其定义深度为 1，说明 Links 字段是有定义的。类似地，在 r2 中值为 NULL 的 Name.Language.Country 字段的定义深度为 1，因为，虽然 Country 字段没有值，但是，Name 字段是有定义的，而 Name 字段的级别是 1，所以，Name.Language.Country 的定义深度为 1；同样，在 r1 中值为 NULL 的两个 Name.Language.Country 字段的定义深度分别为 2（在 Name.Language 内）和 1（在 Name 内）。

- 编码（Encoding）

每列存储为一组块。每个块包括重复深度和定义深度以及压缩的字段值。NULL 是由定义深度来决定的，所以它不会显式地存储。字段路径对应的定义深度小于路径上可选和可重复字段个数总和的，即可认为是 NULL。如果值是有被定义的，那么它的定义深度也不会被存储。类似地，重复深度只在必要时存储。比如，定义深度 0 意味着重复深度 0，

所以后者可省略。事实上，图 13-3 中，没有为 DocId 存储深度。深度被打包为 bit 序列。我们只使用必需的位；比如，如果最大定义深度是 3，我们只需使用 2 个 bit。

13.3.2 将记录转换为列式存储

上面我们展示了使用列式格式表达出记录结构并进行编码。我们要面对的下一个挑战是如何高效地构造 column-stripe（图 13-1 中的右边部分），以及如何计算得到重复深度和定义深度。

计算重复深度和定义深度的基础算法在图 13-4 中给出。算法遍历记录结构然后计算每个列值的深度，当列值为缺失值时也不例外。在 Google 公司的各种应用中，经常会有一个 schema 包含了成千上万的字段，却只有其中少量的几百个字段在记录中被使用。因此，我们需要尽可能廉价地处理缺失字段。为了构造 column-stripe，我们创建一个树状结构，节点为 fieldWriter，它的结构与 schema 中的字段深度相符。基本的想法是，只在 fieldWriter 获得对应字段的值时才执行更新，而不尝试往下传递父节点的状态，除非绝对必要。子节点 fieldWriter 继承父节点的深度值。当任意值被添加时，一个子 fieldWriter 将深度值同步到父节点。具体算法如图 13-4 所示。

```
1 procedure DissectRecord(RecordDecoder decoder,
2             FieldWriter writer, int repetitionLevel):
3   Add current repetitionLevel and definition level to writer
4   seenFields = {} // empty set of integers
5   while decoder has more field values
6     FieldWriter chWriter =
7       child of writer for field read by decoder
8     int chRepetitionLevel = repetitionLevel
9     if set seenFields contains field ID of chWriter
10      chRepetitionLevel = tree depth of chWriter
11    else
12      Add field ID of chWriter to seenFields
13    end if
14    if chWriter corresponds to an atomic field
15      Write value of current field read by decoder
16      using chWriter at chRepetitionLevel
17    else
18      DissectRecord(new RecordDecoder for nested record
19        read by decoder, chWriter, chRepetitionLevel)
20    end if
21  end while
22 end procedure
```

图 13-4 将一个记录分解为多个列的算法

图 13-4 展示的是算法如何将一个记录分解为多个列，也就是扫描记录后，计算得到每个字段值的重复深度和定义深度。子过程 DissectRecord 需要一个 RecordDecoder 参数，RecordDecoder 用于遍历二进制记录。FieldWriters 的深度结构和 schema 一致。对于每条记录来说，根 FiledWriter 将作为算法的参数，同时将 repetitionLevel 设为 0。DissectRecord 过程的主要工作就是维护当前的 repetitionLevel。当前的 definitionLevel 是由当前的 writer 在 schema 中所处的位置决定的，设为字段路径上可选字段和重复字段的个数的总和。

算法中的 While 循环（第 5 行）重复迭代所有原子的类型或者记录类型的字段。集合 seenFileds 跟踪记录中是否已经出现过某个字段，同时用来标识最近重复出现的那个字段。chRepetitionLevel 被设置为最近重复字段的 RepetititonLevel，默认值为父亲 RepetitionLevel 的值（9-13 行）。可以看到过程 DissectRecord 被重复地调用。

每个非叶子 writer 都维护着一系列深度（重复深度和定义深度）。同时每个 writer 都附带一个版本号。简单来说，当一个深度增加时，该 writer 的版本号就会递增 1。这样有利于子 writer 高效地记住父亲的版本号。如果一个子 writer 想要得到自己的值（非空），它只要和父亲 writer 同步，随后就能获得新的的深度信息。

因为输入的数据通常包含几十万条记录，几千个字段，所以，在内存中保存这些信息显然是不合理的。一些深度信息可以暂存到磁盘中的文件中。对于一个空记录的无损编码来说，非原子字段（例如图 13-2 中的 Name.Language）可能需要它们自己的 column stripes，这些 column stripes 只包含深度，而没有非空的值。

13.3.3 记录的装配

从列式数据高效地装配记录，对于面向记录的数据处理工具（例如 MapReduce）而言是很重要的。给定一个字段的子集，我们的目标是重组原始记录就好像它们只包含选择的字段，其他字段就当不存在。核心想法是：我们为字段子集创建一个有限状态机（FSM），读取字段值和深度，然后顺序地将值添加到输出结果上。一个字段的 FSM 状态对应这个字段的 reader。状态的变化标记上了重复深度。一旦一个 reader 获取了一个值，我们将查看下一个值的重复深度来决定状态如何变化、跳转到哪个 reader。对于每一条记录，FSM 都是从开始状态到结束状态变化一次。

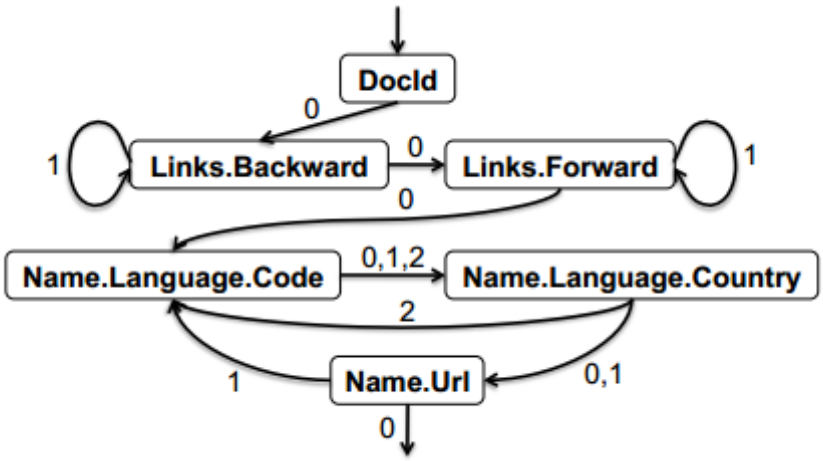

图 13-5 完整记录装配自动机

图 13-5 以 Document 为例展示了一个 FSM 重组一条完整记录的过程。开始状态是 DocId。一旦一个 DocId 值被读取，FSM 就转移到 Links.Backward。获取完所有重复字段 Backward 的值后，FSM 会跳向 Links.Forward，依此类推。这里要明确三个思路：第一，所有数据都是按图 13-3 那种类似一张张“表”的形式存储的；第二，算法会结合 schema，按照一定次序一张张地读取某些“表”（不是所有的，比如只统计 Forward 那就只会读取这一张“表”），次序是不固定的，这个次序也就是状态机内状态变迁的过程；第三，无论次序多么不固定，它都是按记录的顺序不断循环的（比如当前数据按顺序存储着 r1,r2,r3…，那么，就会进入第一个循环读取并装配出 r1，然后，进入第二个循环装配出 r2…），一个循环就是一个状态机从开始到结束的生命周期。通过对上面三点的思考，我们可以想到，在扫描过程中，需要不断做一件非常重要的事情——扫描到某张“表”的某一行时要判断这一行是不是属于下一条记录了，如果是，那么为了继续填充当前记录，就需要跳至下一张“表”继续扫描另一个字段值，否则就用此行的值装配当前记录，如此重复直到需要跳出最后一张“表”，一次循环结束（一个状态机结束，一条记录被装配完毕，进入下一个循环）。理解了这一点，就能理解为何要用状态机来实现算法了，因为循环内就是不断进行状态判断的过程。再深入思考一下，可以想到这个判断不仅是简单的“是否属于下一条记录”，对于 repeated 字段的子孙字段，还需要判断是否属于同一个记录的下一个祖先、并且是哪个层次的祖先。这里举一个例子，比如当前正在装配 r1 中的某个 Name 的某个 Language，扫描到了 Name.Language.Country 的某一行，如果此行重复深度为 0，表示属于下一条记录，说明当前 Name 下 Language 不会再重复了（当前 Name 的所有 Language 装配完毕），于是跳至 Name.Url 继续装配其他属性；如果为 1，表示属于 r1 的下一个 Name，

也说明当前 Name 下 Language 不会再重复（当前 Name 的所有 Language 装配完毕），那也跳到 Name.Url；如果为 2，表示属于当前 Name 的下一个 Language（当前 Name 的 Language 还未装配完毕），那就走一个小循环，跳回上一个 Name.Language.Code 以装配当前 Name 的下一个 Language。

FSM 的构造逻辑可以这么表示：令 *l* 为当前字段读取器为字段 *f* 所返回的下一个重复深度。在 schema 树中，我们找到它在深度 *l* 的祖先，然后选择该祖先节点的第一个叶子字段 *n*。这样的 FSM 状态变化可以简写为(*f*,*l*)->*n*。比如，让 *l*=1 为 *f*=Name.Language.Country 读取的下一个重复深度。它的重复深度为 1 的祖先是 Name，它的第一个叶子字段是 *n*=Name.Url。

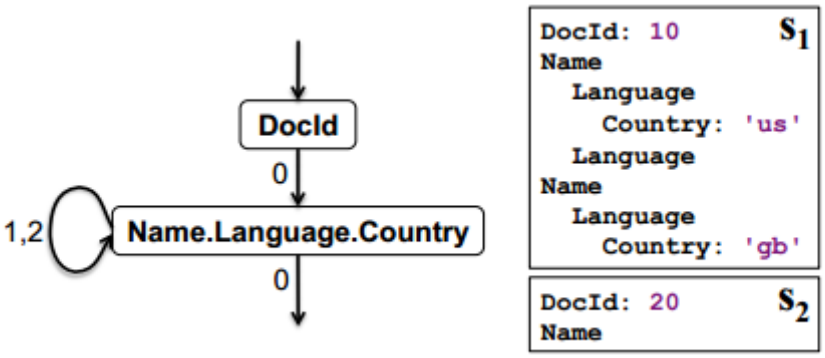

图 13-6 从两个字段中组装出记录的自动机及其组装结果

如果只有一个字段子集需要被处理，FSM 则更简单。图 13-6 描述了一个 FSM，读取字段 DocId 和 Name.Language.Country。图中展示了输出记录 s1 和 s2。注意，Dremel 的编码和装配算法保护了字段 Country 的封闭结构。这个对于应用访问过程很重要，比如，Country 出现在第二个 Name 的第一个 Language，在 XPath 中，就可以用此表达式访问：/Name[2]/Language[1]/Country。

13.4查询语言

Dremel 的查询语言基于 SQL，可在列式嵌套存储上高效地执行，这里简介一下查询语言的特点。每个 SQL 语句（被翻译成代数运算）以一个或多个嵌套表格和它们的 schema 作为输入，输出一个嵌套表格和它的 schema。图 13-7 描述了一个查询例子，执行了投影、选择和记录内聚合等操作。例子中的查询执行在图 13-2 中的 t = {r1,r2}表格上。字段是通过路径表达式来引用。查询最终根据某种规则产出一个嵌套结构的数据，不需要

用户在 SQL 中指明构造规则。

```
SELECT DocId AS Id,
  COUNT(Name.Language.Code) WITHIN Name AS Cnt,
  Name.Url + ',' + Name.Language.Code AS Str
FROM t
WHERE REGEXP(Name.Url, '^http') AND DocId < 20;
```

```
Id: 10                                  t1
Name
  Cnt: 2
  Language
    Str: 'http://A,en-us'
    Str: 'http://A,en'
Name
  Cnt: 0
```

```
message QueryResult {
  required int64 Id;
  repeated group Name {
    optional uint64 Cnt;
    repeated group Language {
      optional string Str; }}}
```

图 13-7 简单查询及其查询结果与输出的模式

为了解释这条查询究竟做了什么，我们考虑“选择”操作（WHERE 语句）。一个嵌套的记录可以看作一棵标记树，每个标记代表一个字段。选择操作会把不满足条件的树的分支裁剪掉。因此，只有那些 Name.Url 有定义的、且以‘http’开头的嵌套记录才会被取出来。接下来考虑“投影”操作，每个 SELECT 语句中的数值表达式会在与嵌套中重复最多输入字段的同层产生一个值。因此，字符串连接表达式在输入 schema 的 Name.Language.Code 层（即深度为 3 时）生成 Str 对应的值。Count 表达式说明了记录内的聚合操作。聚合操作在每个 Name 的子记录里完成，并产生一个 64 位的非负整形值表示 Name.Language.Code 在每个 Name 出现的次数。

此语言支持嵌套子查询、记录内聚合、top-k（排序）、joins（多表关联）和用户自定义函数等等。

13.5查询的执行

本节描述在数据分布式存储之后，如何尽可能并行地执行计算过程。核心概念就是实现一个树状的执行过程，将服务器分配为树中的逻辑节点，每个层次的节点履行不同的职责，最终完成整个查询。整个过程可以理解成一个任务分解和调度的过程。查询会被分解成多个子任务，子任务调度到某个节点上执行，该节点可以执行任务返回结果到上层的父节点，也可以继续拆解更小的任务调度到下层的子节点。此方案称为服务树（serving-tree）结构。

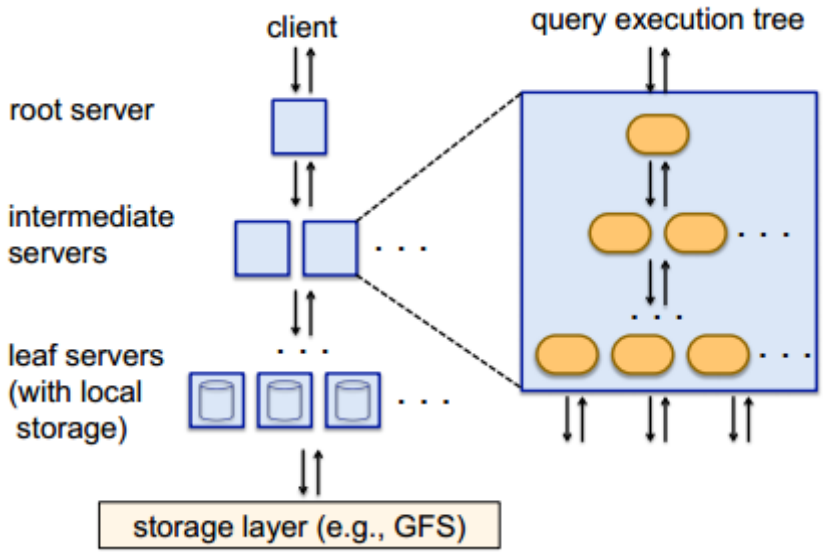

图 13-8 系统架构和在一个服务器节点内部的执行

- **树结构**

Dremel 使用一个多层次服务树来执行查询（见图 13-8）。一个根节点服务器接收到来的查询，从表中读取元数据，将查询路由到下一层。叶子服务器负责与存储层通讯，或者直接在本地磁盘访问数据。

一个简单的聚合查询如下：

SELECT A, COUNT(B) FROM T GROUP BY A

当根节点服务器收到上述查询时，它确定出所有 tablet，也就是表格的水平分割（把一个 column-stripe 理解成一个 table，称之为 T，table 被分布式存储和查询时可认为对 T 进行了水平拆分，tablet 就相当于 T 的一个分区），重写查询为如下：

SELECT A,SUM(c) FROM (R_1^1 UNION ALL ... R_n^1) GROUP BY A

R_1^1 到 R_n^1 是树中第 1 层的（1 到 n）节点返回的子查询结果：

R_i^1=SELECT A, COUNT(B) AS c FROM T_i^1 GROUP BY A

T_i^1可认为是 T 在第 1 层的服务器 i 上被处理时的一个水平分区（tablet）。每一层的节点所做的都是与此相似的重写（rewrite）过程。查询任务被一级级地分解成更小的子任务，最终落实到叶子节点，并行地对 T 的 tablet 进行扫描。在向上返回结果的过程中，中间层的服务器担任了对子查询结果进行聚合的角色。此计算模型非常适用于返回较小结果的聚合查询，这种查询也是交互式应用中最常见的场景。大型的聚合或者其他类型的查询可能更适合使用并行 DBMS 和 MR 来解决。

- **查询分发器**

Dremel 是一个多用户系统，多个查询通常会被同时执行。一个查询分发器会基于查询任务的优先等级和负载均衡对查询任务进行调度。它还能帮助实现容错机制，当一个服务器变得很慢或者一个 tablet 备份不可访问时可以重新调度。

每个查询任务的数据处理量通常比可执行的处理单元（slot）的数量要多。一个 slot 对应一个叶子服务器上的一个执行线程。比如，一个 3000 个叶子服务器的系统，每个叶子服务器使用 8 个线程，则拥有 24000 个 slot。所以，一个 table 分解为 100000 个 tablet，则会分配大约 5 个 tablet 到每个 slot。在查询执行时，查询分发器会统计各 tablet 的处理耗时。如果一个 tablet 耗时较长或不成比例，它会被重新调度到另一个服务器。一些 tablet 可能需要被重新分发多次。

叶子服务器读取列式结构数据中的 stripe。每个 stripe 的块被异步预取；预读缓存通常命中率为 95%。tablet 一般复制三份。当一个叶子服务器读取其中一个备份失败时，它就会去读取另一个备份。

查询分发器有一个重要参数，它表示在返回结果之前一定要扫描百分之多少的 tablet，设置这个参数到较小的值（比如 98%而不是 100%）通常能显著地提升执行速度，特别是当使用较小的复制系数时。

本章小结

本章描述了一种能对大数据进行交互式分析的分布式系统——Dremel。Dremel 由几个简单是组件组成，是一种通用的、管理可扩展数据的解决方案，能在短时间内完成对大规模数据的交互式查询与分析。同时，Dremel 具有很强的可扩展性、稳定性，它实现了对 MapReduce 的一种互补。

参考文献

[1]Melnik, Sergey, et al. "Dremel: interactive analysis of web-scale datasets." Proceedings of the VLDB Endowment 3.1-2 (2010): 330-339.

[2] 经典论文翻译导读之《Dremel: Interactive Analysis of WebScale Datasets》. http://blog.csdn.net/macyang/article/details/8566105

[3] 颜开. Google Dremel 原理 - 如何能 3 秒分析 1PB. http://blog.jobbole.com/29561/#jtss-tsina

附录 1:任课教师介绍

林子雨(1978－),男,博士,厦门大学计算机科学系助理教授,主要研究领域为数据库,数据仓库,数据挖掘.
主讲课程：《大数据技术基础》
办公地点：厦门大学海韵园科研 2 号楼
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu