**Dremel：Web规模数据集的交互式分析**

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis

Google, Inc.

{melnik,andrey,jlong,gromer,shiva,mtolton,theov}@google.com

# ABSTRACT

Dremel是一个可扩展的交互式点对点查询系统，用于分析只读嵌套数据。通过组合多级执行树和列式数据布局，它能够在几秒钟内在万亿行表上运行聚合查询。该系统可扩展到数千个CPU和数PB的数据，并且Google拥有数千名用户。在本文中，我们描述了Dremel的体系结构和实现，并解释了它如何补充基于MapReduce的计算。我们为嵌套记录提供了一种新颖的柱状存储表示，并讨论了几千个系统节点实例的实验。

# 1. INTRODUCTION

大规模分析数据处理已经在网络公司和各个行业中得到普及，尤其是由于低成本存储能够收集大量关键业务数据。 将这些数据放在分析师和工程师的指尖变得越来越重要; 交互式响应时间通常会在数据探索，监控，在线客户支持，快速原型设计，数据管道调试和其他任务中产生质的差异。

大规模执行交互式数据分析需要高度的并行性。 例如，使用今天的商品磁盘在一秒内读取1TB的压缩数据将需要数万个磁盘。 同样，CPU密集型查询可能需要在数千个内核上运行才能在几秒钟内完成。 在谷歌，大规模并行计算是使用共享的商用机器集群完成的[5]。 群集通常托管多个分布式应用程序，这些应用程序共享资源，具有广泛变化的工作负载，并在具有不同硬件参数的计算机上运行。 分布式应用程序中的单个工作程序执行给定任务可能比其他工作程序花费更长时间，或者可能永远不会由于集群管理系统的故障或抢占而完成。 因此，处理落后者和失败对于实现快速执行和容错至关重要[10]。

网络和科学计算中使用的数据通常是非关系的。 因此，灵活的数据模型在这些领域中至关重要。 编程语言，消息中使用的数据结构 由分布式系统，结构化文档等交换，自然地适用于 *嵌套* 表示。 在网络规模上规范化和重新组合这些数据通常是令人望而却步的。 嵌套数据模型是Google [21]的大多数结构化数据处理的基础，据报道是其他主要网络公司的基础。

本文描述了一个名为Dremel的系统[[[1]](#footnote-1)] 支持通过商用机器的共享集群对非常大的数据集进行交互式分析。 与传统数据库不同，它能够 *在原位* 嵌套数据上运行。 *原位* 是指 *在“就地”* 访问数据的能力，例如，在分布式文件系统（如GFS [14]）或另一个存储层（例如Bigtable [8]）中。 Dremel可以对通常需要一系列MapReduce（MR [12]）作业的数据执行许多查询，但只占执行时间的一小部分。 Dremel不是MR的替代品，通常与它一起用于分析MR管道的输出或快速原型化更大的规模计算。

Dremel自2006年开始投入生产，在Google内拥有数千名用户。 Dremel的多个实例部署在公司中，范围从数十个到数千个节点。 使用该系统的示例包括：

* 分析已爬网的Web文档 / 跟踪Android电子市场上应用程序的安装数据。/ Google产品的崩溃报告。/ 来自Google图书的OCR结果。/ 垃圾邮件分析。/ 在Google地图上调试地图图块。/ 平板电脑在托管Bigtable实例中的迁移。/ 测试结果在Google的分布式构建系统上运行。/ 数十万个磁盘的磁盘I / O统计信息。/ 对Google数据中心内的作业进行资源监控。/ Google代码库中的符号和依赖项。

Dremel建立在Web搜索和并行DBMS的概念之上。 首先，它的架构借用了分布式搜索引擎中使用的服务树的概念[11]。 就像Web搜索请求一样，查询会在树中向下推，并在每一步都被重写。 通过聚合从树的较低深度接收的回复来组装查询的结果。 其次，Dremel提供了一种高级的，类似SQL的语言来表达即席查询。 与Pig [18]和Hive [16]等层相比，它本身执行查询而不将它们转换为MR作业。

最后，重要的是，Dremel使用列条带存储表示，使其能够从二级存储中读取更少的数据，并且由于压缩成本更低而降低了CPU成本。 已经采用了列存储来分析关系数据[1]，但据我们所知，还没有扩展到嵌套数据模型。 我们提供的柱状存储格式得到了Google的许多数据处理工具的支持，包括MR，Sawzall [20]和FlumeJava [7]。

在本文中，我们做出以下贡献：

* 我们描述了一种用于嵌套数据的新型柱状存储格式。 我们提出了将嵌套记录分解为列并重新组合的算法（第4节）。
* 我们概述了Dremel的查询语言和执行。 两者都旨在有效地操作列条带嵌套数据，不需要重组嵌套记录（第5节）。
* 我们将展示如何将Web搜索系统中使用的执行树应用于数据库处理，并解释它们有效回答聚合查询的好处（第6节）。
* 我们在10004000节点上运行的系统实例上进行了万亿记录，多TB数据集的实验（第7节）。

此篇文章的结构如下。 在第2节中，我们将解释Dremel如何与其他数据管理工具结合用于数据分析。 其数据模型在第3节中介绍。上面列出的主要贡献在第4-8节中介绍。 相关工作在第9节中讨论。第10节是结论。

# 2. BACKGROUND

我们首先介绍一个场景，该场景说明了交互式查询处理如何适应更广泛的数据管理生态系统。 假设Google的工程师Alice提出了一个从网页中提取新类型信号的新想法。 她运行一个MR作业，通过输入数据曲调并生成包含新信号的数据集，存储在分布式文件系统中的数十亿条记录中。 为了分析她的实验结果，她启动了Dremel并执行了几个交互式命令：

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

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

她的命令在几秒钟内执行。 她还运行了一些其他查询来说服自己她的算法有效。 她发现了signal1的不规则性，并通过编写FlumeJava [7]程序来深入挖掘，该程序对输出数据集执行更复杂的分析计算。 一旦问题得到解决，她就会建立一个管道来连续处理输入的输入数据。 她制定了一些固定的SQL查询，这些查询可以跨越各种维度聚合管道的结果，并将它们添加到交互式仪表板中。 最后，她将新数据集注册到目录中，以便其他工程师可以快速定位和查询。

上述场景需要查询处理器和其他数据管理工具之间的互操作。 第一个成分是 *常见的存储层* 。 Google文件系统（GFS [14]）是公司广泛使用的一种分布式存储层。 尽管有错误的硬件，GFS仍使用复制来保存数据，并且在存在落后者的情况下实现快速响应时间。 高性能存储层对于 *现场* 数据管理 至关重要 。 它允许访问数据而无需耗时的加载阶段，这是分析数据处理中数据库使用的主要阻碍[13]，在DBMS能够加载数据之前，通常可以运行数十个MR分析。执行单个查询。另外一个好处是，可以使用标准工具方便地操纵文件系统中的数据，例如，转移到另一个集群，改变访问权限，或者基于文件名识别用于分析的数据子集。

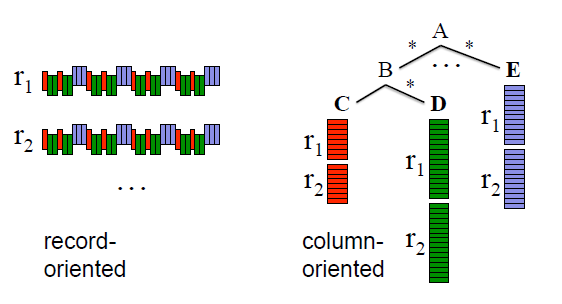


图1：嵌套数据的记录与列式表示

构建可互相协作的数据管理组件的第二个要素是 *共享存储格式* 。 柱状存储证明对于平面关系数据是成功的，但是使其适用于Google需要使其适应嵌套数据模型。 图1说明了主要思想：嵌套字段（如ABC）的所有值都是连续存储的。 因此，可以在不读取AE，ABD等的情况下检索ABC。我们解决的挑战是如何保留所有结构信息并且能够从任意字段子集重建记录。 接下来我们讨论我们的数据模型，然后转向算法和查询处理。

# 3. DATA MODEL

在本节中，我们将介绍Dremel的数据模型，并介绍一些稍后使用的术语。 数据模型起源于分布式系统（解释其名称为“Protocol Buffers”[21]），在Google上广泛使用，可作为开源实现使用。 数据模型基于强类型嵌套记录。 它的抽象语法由下式给出：

*τ* = **dom** | h*A*1 : *τ*[∗|?]*,...,An* : *τ*[∗|?]i

其中 *τ* 是原子类型或记录类型。 **dom**中的原子类型 包括整数，浮点数，字符串等。记录由一个或多个字段组成。 记录中的 字段 *i* 具有名称 *A* *i* 和可选的多重性标签。 *重复* 字段（ \* ）可能会在记录中多次出现。 它们被解释为值列表，即记录中的场出现顺序是显着的。 记录中可能缺少 *可选* 字段（ ？ ）。 否则， *需要* 一个字段 ，即必须恰好出现一次。

为了说明，请考虑图2.它描述了一个定义记录类型 Document 的模式 ，表示Web文档。 模式定义使用[21]中的具体语法。 Document 具有所需的整数 DocId和可选的 Links ，包含Forward和Backword列表，列表中每一项代表其他网页的DocId。。 文档可以有多个 Name，这些 Name 是可以引用文档的不同URL。 一个 Name 包含 Code和（可选） Country 对 的序列 。 图2还显示了 符合模式的 两个样本记录 *r* 1 和 *r* 2 。 使用缩进概述记录结构。 我们将使用这些示例记录来解释下一节中的算法。模式中定义的字段形成树层次结构。 嵌套字段 的完整 *路径* 使用通常的点符号表示，例如 Name.Language.Code 。

嵌套数据模型支持平台中立，可扩展的机制，用于在Google上序列化结构化数据。 代码生成工具为编程语言（如C ++或Java）生成绑定。 使用记录的标准二进制线上表示来实现跨语言互操作性，其中字段值在记录中出现时按顺序排列。这样，用Java编写的MR程序可以使用来自通过C ++库公开的数据源的记录。 因此，如果记录以列式表示形式存储，则快速组装它们对于与MR和其他数据处理工具的互操作非常重要。

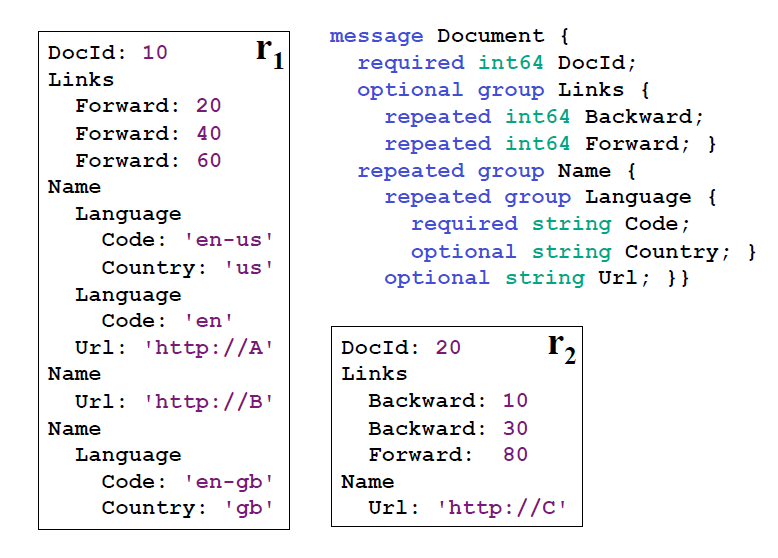


图2：两个示例嵌套记录及其架构

# 4. NESTED COLUMNAR STORAGE 嵌套的柱子存储

如图1所示，我们的目标是连续存储给定字段的所有值，以提高检索效率。 在本节中，我们将讨论以下挑战：以柱状格式（第4.1节），快速编码（第4.2节）和高效记录汇编（第4.3节）的记录结构的无损表示。

# 4.1 Repetition and Definition Levels重复深度和定义深度

仅凭值不能传达记录的结构。 给定repeated字段的两个值，我们不知道重复值的“深度”（例如，这些值是来自两个不同的记录，还是来自同一记录中的两个重复值）。 同样，如果缺少可选字段，我们不知道明确定义了哪些封闭记录。 因此，我们介绍了重复和定义深度的概念，这些概念在下面定义。 供参考，请参见图3，其中总结了我们的样本记录中所有原子字段的重复和定义深度。

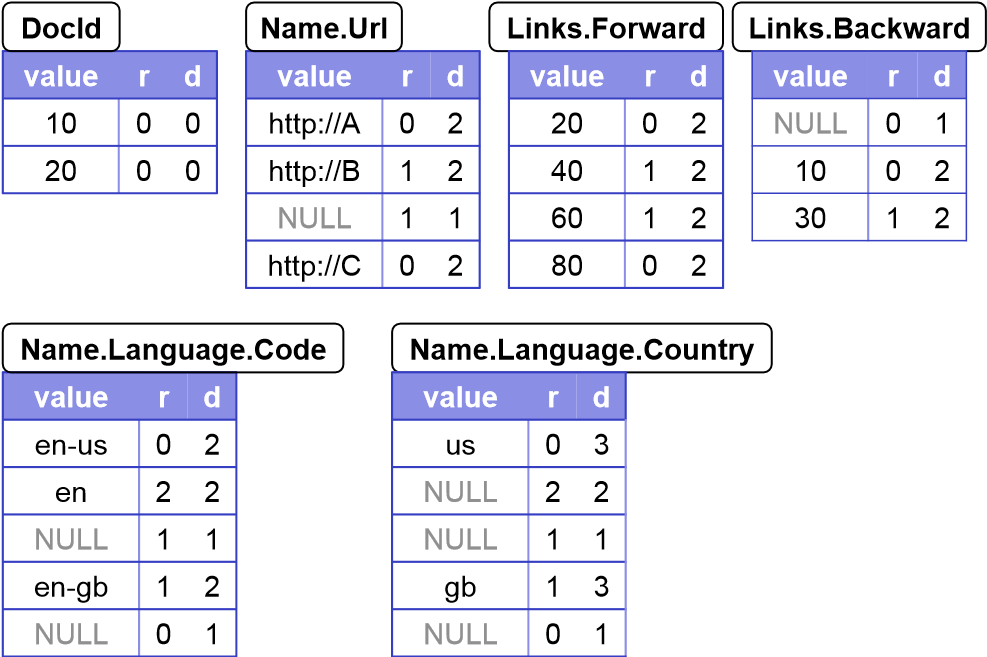


图3：图2中样本数据的列条带表示，显示重复深度（r）和定义深度（d）

*重复深度。* 考虑 图2中的 Code  。它在 *r* 1中 出现三次 。 'en-us'和'en'出现在第一个 Name中 ，而'en-gb'在第三个 Name中 。 为了消除这些事件的歧义，我们将重复深度附加到每个值。 它告诉我们 *在字段路径中重复的字段值已重复* 。 字段路径 Name.Language.Code 包含两个重复的字段， Name 和 Language。 因此， Code 的重复深度 在0到2之间; 深度0表示新记录的开始。 现在假设我们 扫描记录 *r* 1 自上而下。 当我们遇到'en-us'时，我们没有看到任何重复的字段，即重复深度为0.当我们看到'en'时，字段 Language 重复，所以重复深度为2.最后，当我们遇到'en-gb'， Name 重复了（ Language 在 Name 之后只出现一次 ），因此重复深度为1.因此， *r* 1 中 Code 值 的重复深度为 0,2,1 。

请注意*，R* 1 中的第二个 Name 不包含任何 Code  值。 要确定'en-gb'出现在第三个 Name 而不是第二 个 Name 中，我们在'en'和'en-gb'之间添加一个NULL值（参见图3）。 Code  是 Language中 的必填字段 ，因此缺少该 字段 意味着 未定义 Language 。 但是，通常，确定嵌套记录所在的深度需要额外的信息。

*定义深度。* 具有路径 *p* 的字段的每个值 ，尤其是  NULL，具有一个定义深度，指定 *p* *中可以未定义的* *字段* *数* *（因为它们是可选的或重复的）实际存在* 于记录中。为了说明，观察 *r* 1 没有 后向 链接。 但是， 定义了 字段 Links （在深度1）。 为了保留这些信息，我们将一个定义深度为1的NULL值添加到 Links.Backward 列。 类似地，Name.Language.Country的 *R* 2 中 定义深度为1，而它的在 *R* 1具有定义深度2（在 Name.Language 中）和1（在 Name 中）。

我们使用整型定义深度而不是 is-null ， 以便叶子字段的数据（例如， Name.Language.Country ）包含有关其父字段出现的信息; 第4.3节给出了如何使用此信息的示例。

上面概述的编码无损地保留了记录结构。 出于空间原因，我们省略了证明。

*编码。* 每列都存储为一组块。 每个块包含重复和定义深度（以下简称为深度）和压缩字段值。 NULL不会显式存储，因为它们由定义深度确定：任何小于字段路径中重复和可选字段数的定义深度表示NULL。 不会为始终定义的值存储定义深度。 同样，只有在需要时才存储重复深度; 例如，定义深度0表示重复深度0，因此可以省略后者。 实际上，在图3中，没有为 DocId 存储 深度 。 深度按比特序列打包。 我们只根据需要使用多个位; 例如，如果最大定义深度为3，则每个定义深度使用2位。

# 4.2 Splitting Records into Columns将记录拆分为列

上面我们以柱状格式呈现了记录结构的编码。 我们要解决的下一个挑战是如何有效地生成具有重复和定义深度的列条纹。

用于计算重复和定义深度的基本算法在附录A中给出。算法递归到记录结构中并计算每个字段值的深度。 如前所述，即使缺少字段值，也可能需要计算重复和定义深度。Google使用的许多数据集都很少; 具有数千个字段的模式并不罕见，在给定记录中仅使用其中的一百个。 因此，我们尝试尽可能廉价地处理丢失的字段。 为了生成列条带，我们创建了一个 *字段编写器* 树 ，其结构与模式中的字段层次结构相匹配。 基本思想是只有在 拥有自己的数据 时才更新字段编写器 ， 除非绝对 必要， 否则不要尝试将父状态传播到 树中 。子节点writer继承父节点的深度值。当任意值被添加时，一个子writer将深度值同步到父节点。

# 4.3 Record Assembly记录装配

有效地从柱状数据组装记录对于面向记录的数据处理工具（例如，MR）是至关重要的。 给定一个字段子集，我们的目标是重建原始记录，就好像它们只包含所选字段一样，所有其他字段都被剥离。 关键思想是：我们创建一个有限状态机（FSM），读取每个字段的字段值和深度，并将值顺序附加到输出记录。 FSM状态对应于每个所选字段的字段阅读器。 状态转换用重复深度标记。 一旦读者获取了一个值，我们就会查看下一个重复深度，以决定下一个读者使用什么。 对于每个记录，FSM从开始到结束状态遍历一次。

图4显示了在我们运行的示例中重建完整记录的FSM。 开始状态是 DocId 。 一旦 DocId 值被读取时，FSM过渡到 Links.Backward。 在完成所有重复的 Backward 值 之后 ，FSM跳转到 Links.Forward 等。记录汇编算法的细节在附录B中。

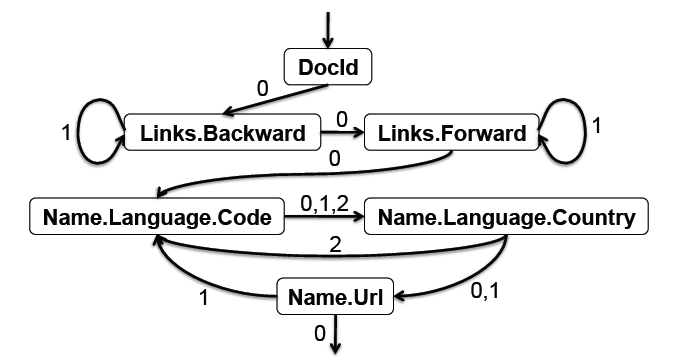


图4：完整的记录组件自动机。 边缘标有重复深度。

为了描绘如何构造FSM转换，让 *l* 为当前字段阅读器为字段 *f* 返回的下一个重复深度 。 从 模式树 中的 *f* 开始 ，我们发现它的祖先在深度 *l* 重复 并选择该 祖先内部 的第一个叶子字段 *n* 。 这为我们提供了FSM转换 （ *f，l* ）→ *n* 。 例如，令 *l* = 1 是 *f* = Name.Language.Country 读取的下一个重复深度 。 其重复深度为1的祖先是 Name ，其第一个叶子字段是 *n* = Name.Url 。 FSM构造算法的细节见附录C.

如果只需要检索一个字段子集，我们构造一个更简单的FSM，执行起来更便宜。 图5描绘了用于读取字段 DocId 和 Name.Language.Country 的FSM 。 该图显示 了自动机生成的输出记录 *s* 1 和 *s* 2 。 请注意我们的编码和汇编算法 保留 Country 字段的封闭结构 。 这对于需要访问的应用程序很重要，例如， 出现在 第二个 Name 的第一 Language 中的 Country 。 在XPath中，这将对应于评估/ Name [2] / Language [1] / Country 等表达式的能力 。

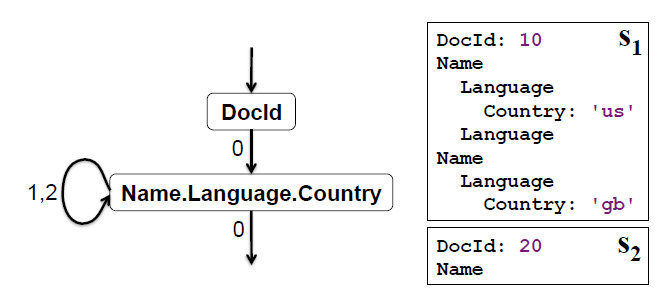


图5：自动机用于汇集来自两个字段的记录及其生成的记录

# 5. QUERY LANGUAGE

Dremel的查询语言基于SQL，旨在在柱状嵌套存储上高效实现。 正式定义语言超出了本文的范围; 相反，我们说明它的特点。 每个SQL语句（以及它转换为的代数运算符）将一个或多个嵌套表及其模式作为输入，并生成嵌套表及其输出模式。 图6描绘了执行投影，选择和记录内聚合的示例查询。 通过 图2中 的表 *t* = { *r* 1 *，r* 2 } 评估查询 。使用路径表达式引用字段。 查询生成嵌套结果，但查询中不存在记录构造函数。

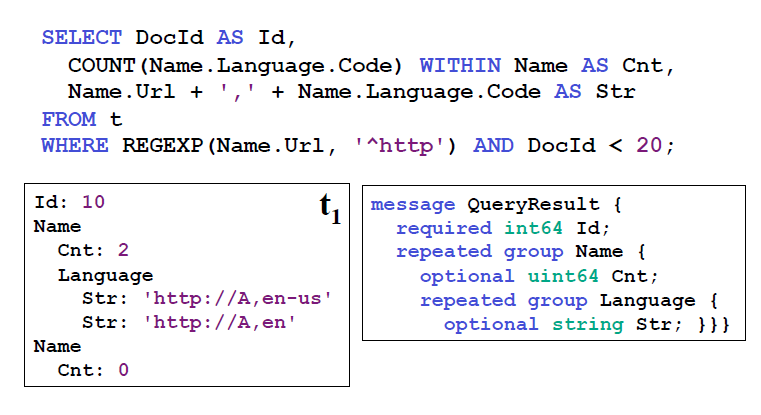


图6：示例查询，结果和输出模式

要解释查询的作用，请考虑选择操作（WHERE子句）。 将嵌套记录视为带标签的树，其中每个标签对应一个字段名称。 选择运算符修剪掉不满足指定条件的树的分支。因此，只保留那些嵌套记录，其中 定义了 Name.Url 并以 http 开头 。 接下来，考虑投影。 SELECT子句中的每个标量表达式都会在与该表达式中使用的最重复输入字段相同的嵌套深度上发出一个值。 因此，字符串连接表达式 在输入模式中 的 Name.Language.Code 深度下Str 值 。 COUNT表达式说明了记录内聚合。 聚合在每个 Name 子记录中完成，并 为每个 Name下Name.Language.Code 的出现次数， 作为非负64位整数（uint64）。

该语言支持嵌套子查询，内部和内部记录聚合，top-k，连接，用户定义的函数等; 这些特征中的一些在实验部分中举例说明。

# 6. QUERY EXECUTION

为简单起见，我们在只读系统的上下文中讨论了核心思想。 许多Dremel查询都是一次性聚合; 因此，我们专注于解释这些并在下一节中将它们用于实验。 我们将对连接，索引，更新等的讨论推迟到未来的工作中。

*树的结构。* Dremel使用多级服务树来执行查询（参见图7）。 根服务器接收传入的查询，从表中读取元数据，并将查询路由到服务树中的下一级。 叶子服务器进行通信 使用存储层或访问本地磁盘上的数据。 考虑下面的简单聚合查询：

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

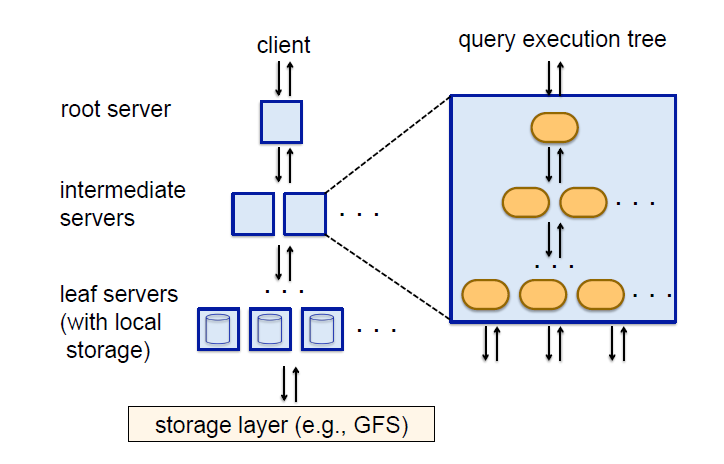


图7：服务器节点内的系统架构和执行

当根服务器收到上述查询时，它确定 包含 *T的* 所有 *tablets*，即表的水平分区， 并按如下方式重写查询：

SELECT A, SUM(c) FROM ( UNION ALL ... *Rn*1 ) GROUP BY A

表是发送到节点的查询的结果 1 *，...，n* 在服务树的1级：

*Ri*1 = SELECT A, COUNT(B) AS c FROM *Ti*1 GROUP BY A

*Ti*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（*tablets*）。每个层执行类似的重写。 最终，查询到达叶子，叶子并行扫描 *T* 中 的*tablets* 。 在向上返回的过程中，中间服务器执行部分结果的并行聚合。 上面介绍的执行模型非常适合返回小型和中型结果的聚合查询，这是一类非常常见的交互式查询。 大型聚合和其他类查询可能需要依赖于并行DBMS和MR中已知的执行机制。

*查询调度程序。* Dremel是一个多用户系统，即通常同时执行多个查询。 查询调度程序根据优先级调度查询并平衡负载。 它的另一个重要作用是 在一台服务器比其他服务器慢得多或平板电脑副本无法访问时 提供 *容错功能* 。

每个查询中处理的数据量通常大于可执行的处理单元数，我们称之为 *插槽slots* 。 插槽对应于叶子服务器上的执行线程。 例如，每个使用8个线程的3,000个叶子服务器的系统具有24,000个插槽。 因此，可以通过为每个插槽分配大约5个*tablets* 来处理跨越100,000个*tablets* 的表格。 在查询执行期间，查询调度程序计算*tablets* 处理时间的直方图。 如果*tablets* 需要不成比例的长时间处理，它会在另一台服务器上重新安排它。 某些*tablets* 可能需要多次重新分配。

叶子服务器以列​​式表示读取嵌套数据的条带。 每个条带中的块都是异步预取的; 预读缓存通常可实现95％的命中率。 *tablets* 通常是三向复制的。 当叶子服务器无法访问一个*tablets* 副本时，它将转移到另一个副本。

查询调度程序遵循一个参数，该参数指定在返回结果之前必须扫描的*tablets* 的最小百分比。 正如我们很快证明的那样，将这样的参数设置为较低的值（例如，98％而不是100％）通常可以 显着 加快执行速度 ，尤其是在使用较小的复制因子时。

每个服务器都有一个内部执行树，如图7右侧所示。内部树对应于物理查询执行计划，包括标量表达式的评估。 为大多数标量函数生成优化的类型特定代码。 project-select-aggregate查询的执行计划 由一组迭代器组成，这些迭代器以锁步方式扫描输入列，并发出使用正确的重复和定义深度注释的聚合和标量函数的结果，在查询执行期间完全绕过记录程序集。 有关详情，请参阅附录D.

一些Dremel查询（例如top-k和count-distinct）使用已知的一次通过算法返回近似结果（例如，[4]）。

# 7. EXPERIMENTS

在本节中，我们将评估Dremel在Google使用的几个数据集上的性能，并检查柱状存储对嵌套数据的有效性。 我们研究中使用的数据集的属性总结在图8中。在未压缩的非复制形式中，数据集占据大约1 PB的空间。 所有表都是三向复制的，除了一个双向复制表，并包含100K到800K不同大小的*tablets*。 我们首先检查单个机器上的基本数据访问特征，然后展示柱状存储如何有利于MR执行，最后关注Dremel的性能。 在常规业务操作期间，在与许多其他应用程序相邻的两个数据中心中运行的系统实例上进行实验。 除非另有说明，否则执行时间是五次运行的平均值。 下面使用的表和字段名称是匿名的。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table name** | **Number of records** | **Size (unrepl., compressed)** | **Number of fields** | **Data center** | **Repl. factor** |
| T1 | 85 billion | 87 TB | 270 | A | 3× |
| T2 | 24 billion | 13 TB | 530 | A | 3× |
| T3 | 4 billion | 70 TB | 1200 | A | 3× |
| T4 | 1+ trillion | 105 TB | 50 | B | 3× |
| T5 | 1+ trillion | 20 TB | 30 | B | 2× |

图8：实验研究中使用的数据集

*本地磁盘。* 在第一个实验中，我们检查了柱状与面向记录的存储的性能权衡，扫描了 包含大约300K行 的表 *T* 1 的1GB片段 （参见图9）。 数据存储在本地磁盘上，压缩列表示大约需要375MB。 面向记录的格式使用较重的压缩，但在磁盘上产生的大小相同。 该实验是在双核Intel机器上完成的，磁盘提供70MB / s的读取带宽。两者执行时的系统环境是相同的，互不影响；OS缓存在每次扫描前都被清空。

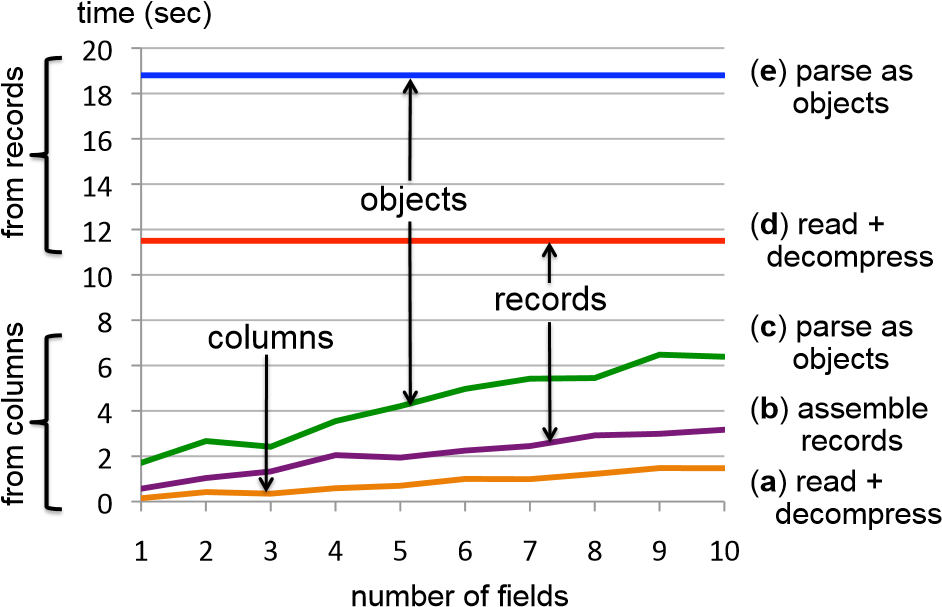


图9：从本地磁盘读取时的性能细分（表 *T* 1的 300K记录片段 ）

该图显示了五个图表，说明了读取和解压缩数据所需的时间，以及为一部分字段汇编和解析记录。 图（a）-（c）概述了柱状存储的结果。 这些图中的每个数据点是通过对30次运行的测量值求平均而获得的，在每次运行中，随机选择给定基数的一组​​列。 图（a）显示了读取和解压缩时间。 图（b）添加了从列组装嵌套记录所需的时间。 图（c）显示了将记录解析为强类型C ++数据结构所需的时间。

图（d）-（e）描述了访问面向记录的存储上的数据的时间。 图（d）显示了读取和解压缩时间。 大部分时间都花在减压上; 实际上，压缩数据可以在大约一半的时间内从磁盘读取。 如图（e）表明，解析在读取和解压缩时间之外再增加50％。 这些费用是针对所有领域支付的，包括不需要的领域。

该实验的主要内容如下：当读取少数列时，柱状表示的增益大约为一个数量级。 柱状嵌套数据的检索时间随字段数呈线性增长。 记录汇编和解析很昂贵，每个都可能使执行时间加倍。 我们观察到其他数据集的类似趋势。 一个自然要问的问题是顶部和底部图形交叉的位置，即 记录方式的存储开始优于柱状存储。 根据我们的经验，交叉点通常位于数十个字段，但它在数据集之间有所不同，取决于是否需要记录汇编。

*MR和Dremel。* 接下来，我们将比较在柱状与面向记录的数据上执行MR和Dremel。 我们考虑访问单个字段的情况，即性能增益最明显。 可以使用图9的结果外推多列的执行时间。在该实验中，我们计算 表 *T* 1 的字段 txtField 中 的平均项数 。 MR执行使用以下Sawzall [20]程序完成：

numRecs: table sum of int;

numWords: table sum of int;

emit numRecs <- 1;

emit numWords <- CountWords(input.txtField);

记录数存储在变量 numRecs中 。 对于每条记录， numWords 增加 CountWords 函数 返回的 input.txtField中 的 项 数。 程序运行后，平均术语频率可以计算为 numWords */*numRecs 。 在SQL中，此计算表示为：

*Q*1: SELECT SUM(CountWords(txtField)) / COUNT(\*) FROM T1

图10显示了两个MR作业和Dremel在对数刻度上的执行时间。 两个MR工作都运行在3000个工作节点上。 类似地，使用3000节点Dremel实例来执行查询 *Q* 1 。 Dremel和MR-on-columns读取约0.5TB的压缩柱状数据，而MR-on-records读取的读取量为87TB。 如图所示，通过从记录导向切换到柱状存储（从几小时到几分钟），MR获得了一个数量级的效率。 通过使用Dremel（从几分钟到几秒）实现另一个数量级。

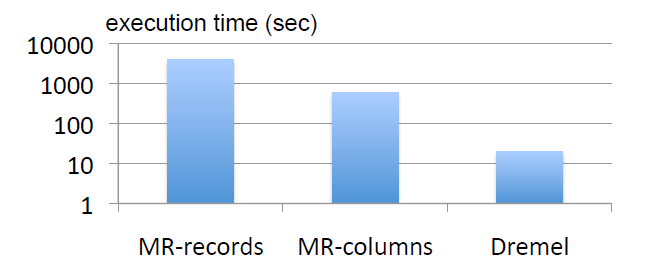


图10：柱状与面向记录的存储上的MR和Dremel执行（3000个节点，850亿条记录）

*服务树拓扑。* 在下一个实验中，我们将展示服务树深度对查询执行时间的影响。 我们在表 *T* 2 上考虑两个GROUP BY查询 ，每个 查询 都使用 对数据进行单次扫描。表 *T* 2 包含240亿个嵌套记录。 每条记录都有一个 包含数字 Amount 的重复字段item 。 字段 item.amount 在数据集中重复约400亿次。 第一个查询按country汇总 item.amount：

*Q*2: SELECT country, SUM(item.amount) FROM T2

GROUP BY country

它返回几百条记录，从磁盘读取大约60GB的压缩数据。第二个查询在 具有选择条件的文本字段domain 上执行GROUP BY 。它读取大约180GB并产生大约110万个不同的域：

*Q*3: SELECT domain, SUM(item.amount) FROM T2

WHERE domain CONTAINS ’.net’

GROUP BY domain

图11显示了每个查询的执行时间与服务器拓扑的关系。在每个拓扑中，叶子服务器的数量保持恒定在2900，以便我们可以假设相同的累积扫描速度。在2级拓扑（1：2900）中，单个根服务器直接与叶子服务器通信。对于3级拓扑，比例为1:100:2900，也就是说，根节点和叶子节点之间有100台中间服务器。4级拓扑为1：10：100：2900。

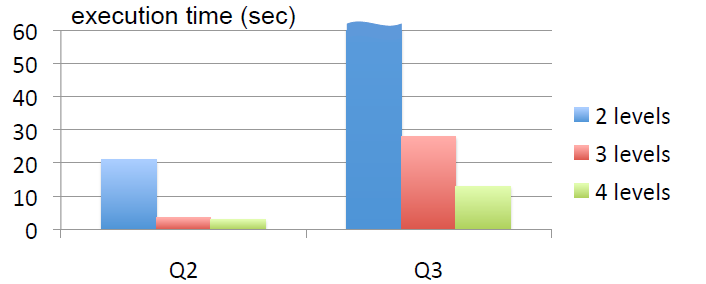


图11：作为函数的执行时间

查询 *Q*2在服务树中使用3级拓扑时在3秒内运行，并且不会从额外拓扑中获益。相反， 由于并行性增加，*Q*3的执行时间减半。在2级拓扑， *Q*3不在图表中，因为根服务器需要按顺序聚合从数千个节点接收的结果。此实验说明了返回多个组的聚合如何从多级服务树中受益。

*每片直方图。* 为了更深入地了解查询执行期间发生的事情，请考虑图12。该图显示了叶子服务器对特定 *Q*2和 *Q*3运行的*tablets*处理速度。从*tablets*计划在可用插槽中执行的时刻开始测量时间，即排除在作业队列中等待的时间。此度量方法会计算同时执行的其他查询的影响。每个直方图下的面积对应于100％。如图所示，99％的 *Q*2（或*Q*3）*tablets*在一秒（或两秒）内处理。

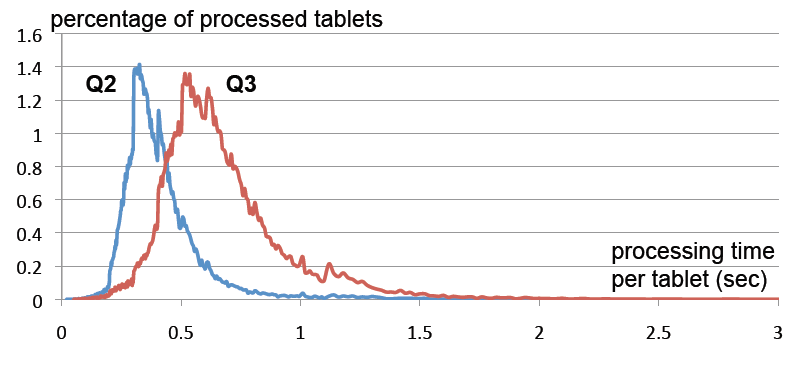


图12：处理时间的直方图

*记录内聚合。* 作为另一个实验，我们检查 在表*T*3上运行的Query *Q*4的性能。该查询说明了记录内聚合：它计算记录中出现的a.b.c.d 值之和大于a.b.p.q.r 值之和的所有记录。这些字段在不同的嵌套深度重复。由于列条带化，从磁盘读取只有13GB（超出70TB），查询在15秒内完成。如果不支持嵌套，在*T*3上运行此查询将非常昂贵。

*Q*4 : SELECT COUNT(c1 *>* c2) FROM (SELECT SUM(a.b.c.d) WITHIN RECORD AS c1,

SUM(a.b.p.q.r) WITHIN RECORD AS c2

FROM T3)

*可扩展性。* 以下实验说明了系统在万亿记录表上的可扩展性。 下面的查询*Q*5选择 表*T*4中的前20个aid 及其出现次数 。该查询扫描4.2TB的压缩数据。

*Q*5: SELECT TOP(aid, 20), COUNT(\*) FROM T4

WHERE bid = {value1} AND cid = {value2}

使用系统的四种配置执行查询，范围从1000到4000个节点。执行时间如图13所示。在每次运行中，总耗用的CPU时间几乎相同，大约为300K秒，而用户感知时间随着系统规模的增大而近似线性降低。这一结果表明，较大的系统在资源使用方面可以与较小的系统一样有效，但允许更快的执行。

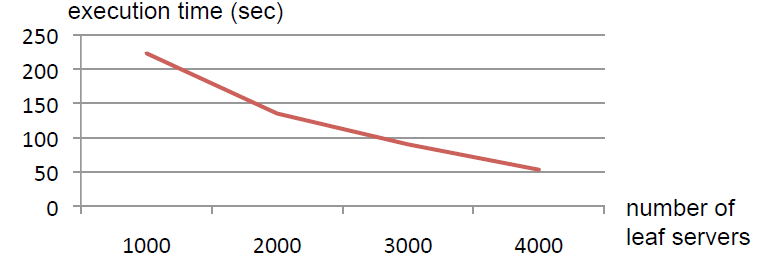


图13：使用 万亿行表*T*4上的top-k查询*Q*5将系统从1000个节点缩放到4000个节点

*Stragglers。* 我们的上一个实验显示了落后者的影响。下面的查询*Q*6在万亿行表 *T*5上运行 。与其他数据集相比， *T*5 是双向复制的。因此，落后者减慢执行的可能性更高，因为重新安排工作的机会较少。

*Q*6: SELECT COUNT(DISTINCT a) FROM T5

查询 *Q*6读取超过1TB 的压缩数据。 检索区域 的压缩比约为10.如图14所示，99％ 的*tablets*的处理时间低于5秒每*tablets*每槽。但是，在2500节点系统上执行时，一小部分*tablets*需要更长的时间，从而将查询响应时间从不到一分钟减慢到几分钟。下一节总结了我们的实验结果和我们学到的经验教训。

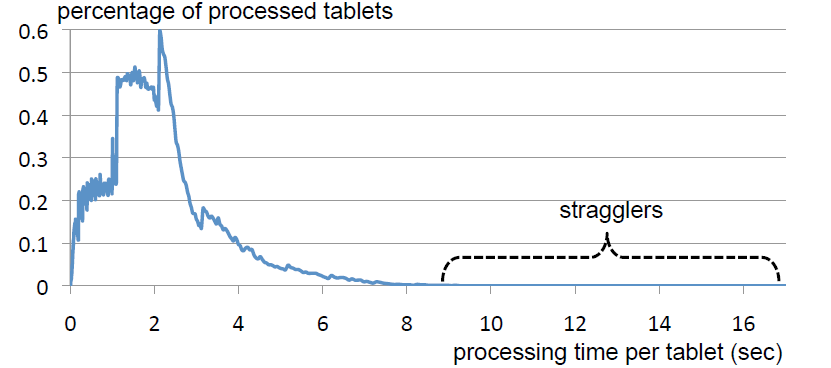


图14：查询 *Q*5上 *Ť*5示出stragglers在 2× 复制

# 8. OBSERVATIONS

Dremel每月扫描数万亿条记录。图15显示了在一个Dremel系统的典型月度工作负载中的查询响应时间分布，以对数标度表示。如图所示，大多数查询在10秒内处理，完全在交互范围内。有些查询在共享群集上实现的扫描吞吐量接近每秒1000亿条记录，在专用计算机上甚至更高。

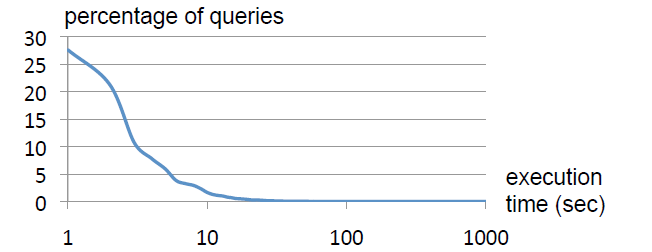


图15：每月工作负载中的查询响应时间分布

上述实验数据表明以下观察结果：

* 基于扫描的查询可以在交互式速度上以高达一万亿条记录的磁盘驻留数据集执行。
* 对于包含数千个节点的系统，可以实现列数和服务器数量的近线性可扩展性。
* MR可以像DBMS一样受益于柱状存储。
* 记录汇编和解析很昂贵。需要优化软件层（超出查询处理层）以直接使用面向列的数据。
* MR和查询处理可以互补的方式使用; 一层的输出可以输入另一个输入。
* 在多用户环境中，较大的系统可以从规模经济中受益，同时提供质量更好的用户体验。
* 如果准确的交易速度是可以接受的，则可以更早地终止查询并查看大部分数据。
* 可以快速扫描大量的Web比例数据集。在紧迫的时间范围内达到最后几个百分点是很难

Dremel的代码库很密集; 它包含不到100K行的C ++，Java和Python代码。

# 9. RELATED WORK

MapReduce（MR）[12]框架旨在解决长期批处理作业环境中大规模计算的挑战。与MR一样，Dremel提供容错执行，灵活的数据模型和 *现场* 数据处理功能。MR的成功带来了广泛的第三方实施（特别是开源Hadoop [15]），以及许多将并行DBMS与MR相结合的混合系统，由Aster，Cloudera，Greenplum和Vertica等供应商提供。HadoopDB [3]是这种混合类别的研究系统。最近的文章[13,22]对比了MR和并行DBMS。我们的工作强调两种范式的互补性。

Dremel旨在大规模运营。虽然可以想象并行DBMS可以扩展到数千个节点，但我们并不知道任何已发布的工作或行业报告。我们也不熟悉研究MR柱状存储的先前文献。

我们嵌套数据的柱状表示建立在几十年前的想法上：结构与内容的分离和转置表示。最近关于列商店工作的评论，包括压缩和查询处理，可以在[1]中找到。许多商业DBMS支持使用XML存储嵌套数据（例如，[19]）。 XML存储方案试图将结构与内容分离，但由于XML数据模型的灵活性而面临更多挑战。一个使用柱状XML表示的系统是XMill [17]。 XMill是一种压缩工具。它存储所有字段的结构，并且不适合选择性地检索列。

Dremel中使用的数据模型是[2]中讨论的复值模型和嵌套关系模型的变体。 Dremel的查询语言建立在[9]的思想基础之上，它引入了一种在访问嵌套数据时避免重构的语言。相反，在XQuery和面向对象的查询语言中通常需要重组，例如，使用嵌套的for循环和构造函数。我们不了解[9]的实际实现。最近在嵌套数据上运行的类似SQL的语言是Pig [18]。其他用于并行数据处理的系统包括Scope [6]和DryadLINQ [23]，并在[7]中有更详细的讨论。

# 10. CONCLUSIONS

我们介绍了Dremel，这是一个用于大型数据集交互式分析的分布式系统。Dremel是一个定制的，可扩展的数据管理解决方案，由更简单的组件构建。它补充了MR范式。我们讨论了它在真实数据的万亿记录，多TB数据集上的性能。我们概述了Dremel的关键方面，包括其存储格式，查询语言和执行。在未来，我们计划更深入地涵盖正式代数规范，连接，可扩展性机制等领域。

# 11. ACKNOWLEDGEMENTS

Dremel受益于Google的许多工程师和实习生的投入，特别是Craig Chambers，Ori Gershoni，Rajeev Byrisetti，Leon Wong，Erik Hendriks，Erika Rice Scherpelz，Charlie Garrett，Idan Avraham，Rajesh Rao，Andy Kreling，Li Yin ，Madhusudan Hosaagrahara，Dan Belov，Brian Bershad，Lawrence You，Rongrong Zhong，Meelap Shah和Nathan Bales。

# 12. REFERENCES

1. D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-Oriented Database Systems. *VLDB*, 2(2), 2009.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
3. A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *VLDB*, 2(1), 2009.
4. Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *RANDOM*, pages 1–10, 2002.
5. L. A. Barroso and U. Holzle.¨ *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
6. R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *VLDB*, 1(2), 2008.
7. C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*, 2010.
8. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.
9. L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. *SIGMOD Rec.*, 18(2), 1989.
10. G. Czajkowski. Sorting 1PB with MapReduce. Official Google Blog, Nov. 2008. At http://googleblog.blogspot.com/ 2008/11/sorting-1pb-with-mapreduce.html.
11. J. Dean. Challenges in Building Large-Scale Information Retrieval Systems: Invited Talk. In *WSDM*, 2009.
12. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
13. J. Dean and S. Ghemawat. MapReduce: a Flexible Data Processing Tool. *Commun. ACM*, 53(1), 2010.
14. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.
15. Hadoop Apache Project. http://hadoop.apache.org.
16. Hive. http://wiki.apache.org/hadoop/Hive, 2009.
17. H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *SIGMOD*, 2000.
18. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-so-Foreign Language for Data Processing. In *SIGMOD*, 2008.
19. P. E. O’Neil, E. J. O’Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.
20. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4), 2005.
21. Protocol Buffers: Developer Guide. Available at http://code.google.com/apis/protocolbuffers/docs/overview.html.
22. M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Commun. ACM*, 53(1), 2010.
23. Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K.´ Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

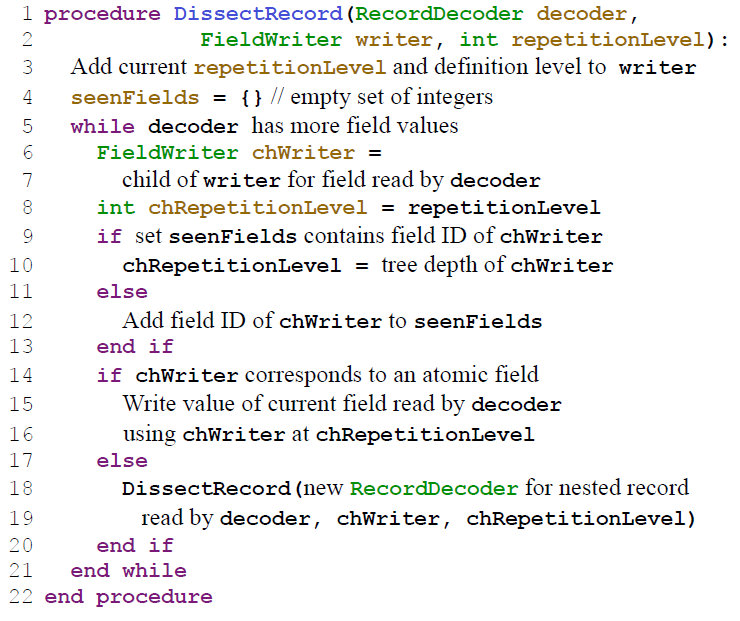


图16：用于将记录分解为列的算法

# APPENDIX A. COLUMN-STRIPING ALGORITHM列条带算法

用于分解记录为列，则算法在图16所示步骤 DissectRecord 传递一个实例 RecordDecoder ，其被用于遍历二进制编码记录。 FieldWriter 形成一个与输入模式同构的树层次结构。根 FieldWriter 被传递到算法对于每一个新的记录，与 repetitionLevel 设置为0的的主要工作 DissectRecord 过程是保持当前 repetitionLevel 。当前 definitionLevel 由当前 writer 的树位置唯一确定 ，作为字段路径中可选字段和重复字段数的总和。

算法的while循环（第5行）迭代给定记录中包含的所有原子和记录值字段。 setFields 集跟踪记录中是否显示了一个字段。它用于确定最近重复的字段。子重复深度chRepetitionLevel 设置为最近重复的字段的值，否则默认为其父深度（第9-13行）。在嵌套记录上递归调用该过程（第18行）。

在4.2节中，我们描绘了 FieldWriter 如何累积深度并将它们懒洋洋地传播到较低深度的编写器。这样做如下：每个非叶写入器保持一系列（重复，定义）深度。每个作者还有一个与之关联的“版本”号。简单地说，只要添加一个深度，编写器版本就会增加1。孩子们记住他们同步的最后一个父母的版本就足够了。如果子编写器获得其自己的（非null）值，则它通过获取新深度将其状态与父级同步，然后才添加新数据。

由于输入数据可能包含数千个字段和数百万条记录，因此将所有深度存储在内存中是不可行的。某些深度可能临时存储在磁盘上的文件中。对于空（子）记录的无损编码，非原子字段（ 如图2中的Name.Language ）可能需要具有自己的列条带，仅包含深度但不包含非NULL值。

# B. RECORD ASSEMBLY ALGORITHM记录组装算法

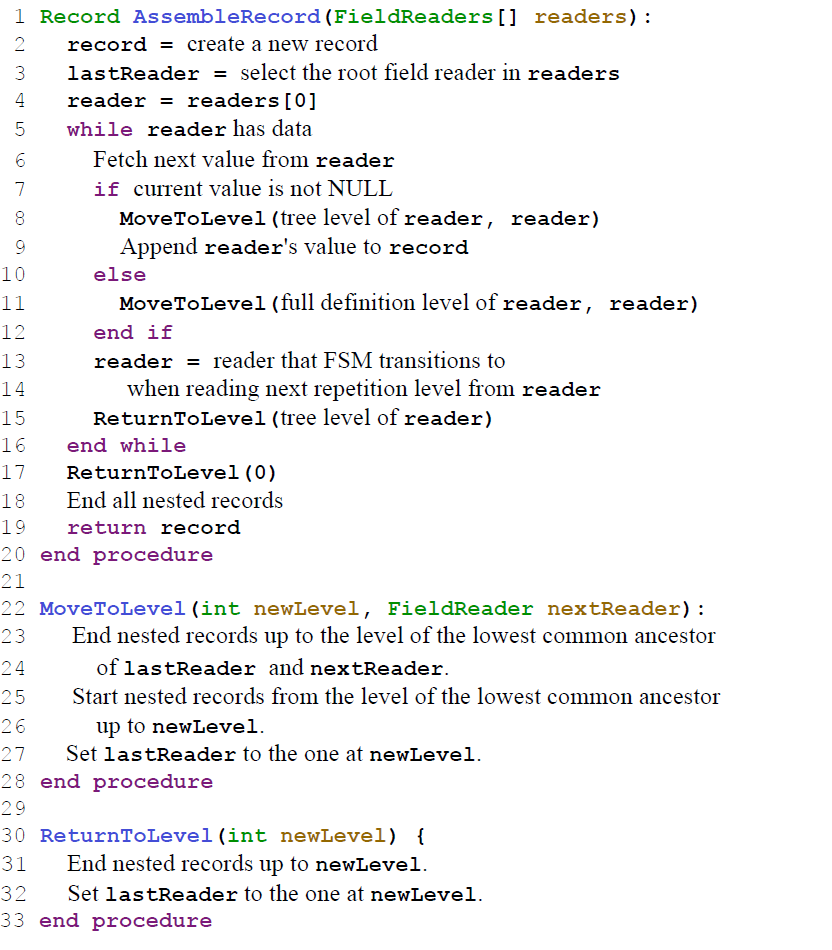


图17：用于从列组装记录的算法

在他们的在线表示中，记录被布置为 字段标识符的对，后跟字段值。嵌套记录可以被认为具有“开始标记”和“结束标记”，类似于XML（实际二进制编码可能不同，详见[21]）。在下文中，写入开始标记被称为“开始”记录，而写入结束标记被称为“结束”它。

AssembleRecord 过程将一组 FieldReader 和（隐式）FSM 作为输入，并在读取器之间进行状态转换。变量 reader 将当前 FieldReader 保存在主程序中（第4行）。变量lastReader 保存最后一个读取器，其值附加到记录中，并且可用于图17中所示的所有三个过程。主要的while循环位于第5行。我们从当前 reader 获取下一个值。如果值不是NULL，这是通过查看其定义深度确定的，我们 将方法MoveToLevel中正在汇编的记录 同步到当前 reader 的记录结构 ，并将字段值附加到记录。否则，我们只是调整记录结构而不附加任何值 - 如果存在空记录则需要这样做。在第12行，我们使用'完整定义深度'。回想一下，定义深度会将所需字段计算出来（仅计算重复和可选字段）。完整定义深度考虑所有字段。

步骤 MoveToLevel 过渡记录从状态 lastReader 到 nextReader （见第22行）。例如，假设 lastReader 对应于 Links.Backward 于图2和 nextReader 是 Name.Language.Code 。该方法结束嵌套记录 Links 并按 顺序启动新记录 Name 和 Language 。过程 ReturnsToLevel （第30行）是 MoveToLevel的副本 ，它只结束当前记录而不启动任何新记录。

# C. FSM CONSTRUCTION ALGORITHM FSM构造算法

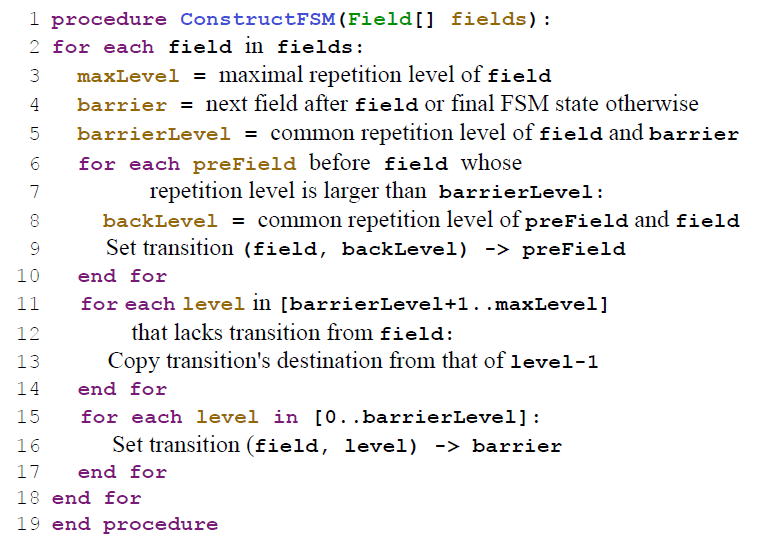


图18：构造记录组件自动机的算法

图18显示了用于构造执行记录组装的有限状态机的算法。该算法将应该在记录中填充的字段作为输入，按照它们在模式中出现的顺序。该算法使用两个字段的“共同重复深度”的概念，这是其最低共同祖先的重复深度。例如， Links.Backward 和 Links.Forward 的公共重复深度等于1.第二个概念是“屏障”，它是当前序列之后的序列中的下一个字段。直觉是我们尝试逐个处理每个字段，直到屏障被击中并需要跳转到之前看到的字段。

该算法包括三个步骤。在步骤1（第6-10行）中，我们向后查看常见的重复深度。这些保证不会增加。对于我们遇到的每个重复深度，我们选择序列中最左边的字段 - 这是我们在 FieldReader 返回重复深度时需要转换到的字段。在第2步中，我们填补了空白（第11-14行）。产生间隙是因为并非所有重复深度都存在于第8行计算的公共重复深度中。在步骤3（第15-17行）中，我们设置了等于或低于屏障深度的所有深度的转换以跳转到屏障领域。如果 FieldReader 产生这样的深度，我们需要继续构建嵌套记录，而不需要从屏障中弹回。

# D. SELECT-PROJECT-AGGREGATE EVALUATION ALGORITHM评估算法

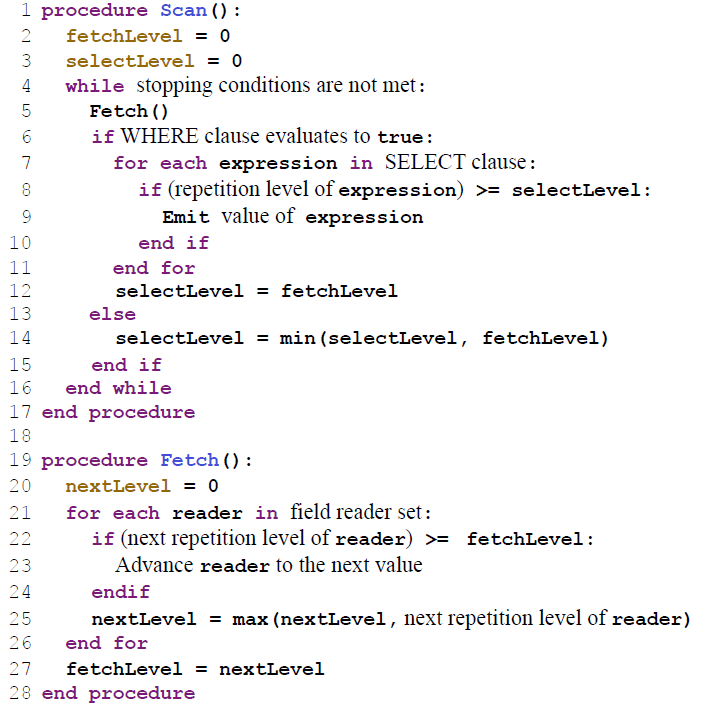


图19：用于评估列式输入上的select-project-aggregate查询的算法，绕过记录程序集

图19显示了用于在Dremel中评估select-projectaggregate查询的算法。该算法解决了查询可能引用重复字段的一般情况; 更简单的优化版本用于平面关系查询，即仅引用必需和可选字段的查询。该算法有两个隐式输入：一组 FieldReader ，一个用于查询中出现的每个字段，一组标量表达式，包括聚合 表达式，存在于查询中。标量表达式的重复深度（在第8行中使用）被确定为该表达式中使用的字段的最大重复深度。

本质上，该算法使读取器以锁步方式推进到下一组值，并且如果满足选择条件，则发出预测值。选择和投影由两个变量 fetchLevel 和 selectLevel控制 。在执行期间，只有下一个重复深度不低于 fetchLevel的 readers才会被提升（参见 第19行的 Fetch 方法）。类似地，仅 发出当前重复深度不小于selectLevel的表达式 （第7-10行）。该算法确保较高深度的嵌套表达式（即具有较小重复深度的表达式）仅针对每个较深层嵌套表达式进行评估和发出一次。

1. [] Dremel是一种电动工具品牌，主要依靠速度而不是扭矩。 我们仅将此名称用于内部项目。 [↑](#footnote-ref-1)