经典论文翻译导读之《Dremel: Interactive Analysis of WebScale Datasets》

[译者注]**从头到尾读懂一篇国外经典技术论文！**相信这是很多技术爱好者一直以来想干的事情。本系列译文的目标是满足广大技术爱好者对原始论文一窥究竟的需求，尽量对原文全量翻译。原始论文中不乏较晦涩的学术性语句，也可能会有您不感兴趣的段落，所以译者会添加【译者预读】【译者总结】等环节帮助大家选择性的阅读，或者帮助读者总结。根据译者的翻译过程，论文中也难免缺少细节的推导过程（Google的天才们总是把我们想的跟他们一样聪明），遂添加特殊的【译者YY】环节，根据译者的理解对较为复杂的内容进行解释和分析，此部分主观性很大难免有误，望读者矫正。所有非原文内容皆以蓝色文字显示。

废话不多说，大家一览为快吧！

【译者预读】此篇是伴随着Dremel神话横空出世的原始论文（不知道Dremel的读者可以立刻搜索感受一下Dremel的强大）。文章深入分析了Dremel是如何利用巧妙的数据存储结构+分布式并行计算，实现了3秒查询1PB的神话。

论文的前几部分是“abstract”、“introduction”、“background”，介绍性的文字较多，其核心意思是：面对海量数据的分析处理，MR（MapReduce）的优势不需多言，其劣势在于时效性较差不满足交互式查询的需求，比如3秒内完成对万亿数据的一次查询等，Dremel应此需求而生，与MR成为有效互补。

摘要

Dremel是一个用于分析只读嵌套数据的可伸缩、交互式ad-hoc查询系统。通过结合多层级树状执行过程和列状数据结构，它能做到几秒内完成万亿行数据之上的聚合查询。此系统可伸缩至成千上万的CPU和PB级别的数据，而且在google已有几千用户。在本篇论文中，我们将描述Dremel的架构和实现，解释它为何是MapReduce计算的有力互补。我们提供一种嵌套记录的列状存储结构，并且讨论在拥有几千个节点的系统上进行的实验。

1. 介绍

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，尤其是因为目前已经可以用廉价的存储来收集和保存海量的关键业务数据。如何让分析师和工程师便捷的利用这些数据也变得越来越重要；在数据探测、监控、在线用户支持、快速原型、数据管道调试以及其他任务中，交互的响应时间一般都会造成本质的区别。
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，如果使用普通的硬盘，希望在1秒内读取1TB压缩的数据，则需要成千上万块硬盘。相似的，CPU密集的查询操作也需要运行在成千上万个核上。在Google，大量的并行计算是使用普通PC组成的共享集群完成的[5]。一个集群通常会部署大量共享资源的分布式应用，各自产生不同的负载，运行在不同硬件配置的机器上。一个分布式应用的单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取代而永远不能完成。因此，处理好异常、故障是实现快速执行和容错的重要因素[10]。
互联网和科学计算中的数据经常是独立的、互相没有关联的。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，都可以用嵌套式表达法来很自然的描述。规格化、重新组合这些互联网规模的数据通常是代价昂贵的。嵌套数据模型成为了大部分结构化数据在Google处理的基础[21]，据报道也在其他互联网公司被使用。

这篇论文描述了一个叫做Dremel的系统，它支持在普通PC组成的共享集群上对超大规模的数据集合执行交互式查询。不像传统的数据库，它能够操作原位嵌套数据。原位意味着在适当的位置访问数据的能力，比如，在一个分布式文件系统（比如GFS[14]）或者其他存储层（比如Bigtable[8]）。查询这些数据一般需要一系列的MapReduce（MR[12]）任务，而Dremel可以同时执行很多，而且执行时间比MR小得多。Dremel不是为了成为MR的替代品，而是经常与它协同使用来分析MR管道的输出或者创建大规模计算的原型系统。
Dremel自从2006就投入生产了并且在Google有几千用户。多种多样Dremel的实例被部署在公司里，排列着成千上万个节点。使用此系统的例子包括：

- 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
- 托管Bigtable实例中的Tablet迁移
- Google分布式构建系统中的测试结果分析
- 成百上千的硬盘的磁盘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

Dremel基于互联网搜索和并行DBMS的概念。首先，它的架构借鉴了用在分布式搜索引擎中的服务树概念[11]。就像一个web搜索请求一样，查询请求被推入此树、在每个步骤被重写。通过聚合从下层树节点中收到的回复，不断装配查询的最终结果。其次，Dremel提供了一个高级、类SQL的语言来表达ad-hoc查询。与Pig[18]和Hive[16]不同，它使用自己技术执行查询，而不是翻译为MR任务。

最后也是最重要的，Dremel使用了一个column-striped的存储结构，使得它能够从二级存储中读取较少数据并且通过更廉价的压缩减少CPU消耗。列存储曾被采用来分析关系型数据[1]，但是据我们了解还没有推广到嵌套数据模型上。我们所展现的列状存储格式在Google已经有很多数据处理工具支持，包括MR、Sawzall[20]、以及FlumeJava[7]。

在本论文中我们做了如下贡献：

- 我们描述了一个嵌套数据的列状存储格式。并且提供了算法，将嵌套记录解剖为列结构，在查询时重新装配它们（第4章节）。
- 我们描述了Dremel的查询语言和执行过程。两者都被定制化设计，能够在column-striped的嵌套数据上高效执行，不需要装载原始嵌套记录（章节5）。
- 我们展示了在web搜索系统中使用的树状执行过程如何被适用到数据库处理，并且解释他们的优劣，以及如何做到高效聚合查询（章节6）。
- 我们在万亿记录、TB级别的数据集合上进行实验，系统实例拥有1000-4000个节点[章节7]。

这篇论文结构如下。章节2中我们解释了Dremel如何结合其他数据管理工具进行数据分析。它的数据模型在章节3介绍。上述主要贡献覆盖在章节4-8。相关工作在章节9讨论。章节10是总结。

2. 背景

作为开始我们来看一个场景，这个场景说明了交互式查询处理的必要性，以及它在数据管理生态系统上怎么定位。假设一个Google的员工Alice，诞生了一个新奇的灵感，她想从web网页中提取新类型的signals。她运行一个MR任务，分析输入数据然后产生这种signals的数据集合，在分布式文件系统上存储这数十亿条记录。为了分析她实验的结果，她启动Dremel然后执行几个交互式命令：

```
DEFINE TABLE t AS /path/to/data/*
SELECT TOP(signal1, 100), COUNT(*) FROM t
```

她的命令只需几秒钟就执行完毕。她也运行了几个其他的查询来证实她的算法是正确的。她发现signal1中有非预期的情况于是写了个FlumeJava[7]程序执行了一个更加复杂的分析式计算。一旦这个问题解决了，她建立一个管道，持续的处理输入数据。然后她编写了一些SQL查询来跨维度的聚合管道的输出结果，然后将它们添加到一个交互式的dashboard，其他工程师能非常快速的定位和查询它。
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操作。第一个组成部分是一个通用存储层。Google File System（GFS[14]）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，即使出现掉队者（stragglers）也能达到快速响应时间。对原位数据管理来说，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[13]不太被使用。另外一个好处是，在文件系统中能使用标准工具便捷的操作数据，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。

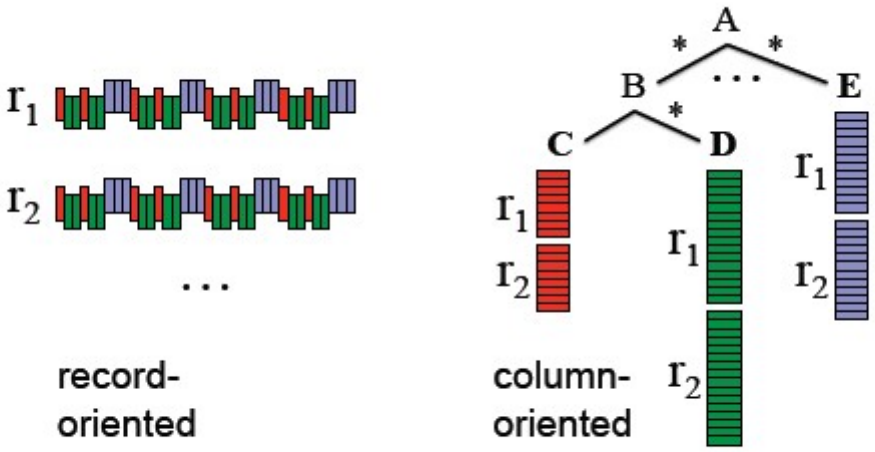


Figure 1: Record-wise vs. columnar representation of nested data

第二个构建互相协作的数据管理组件的要素，是一个共享的存储格式。列状存储已经证明了它适用于扁平的关系型数据，但是使它适用于Google则需要适配到一个嵌套数据模型。图1展示了主要的思想：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。我们面临的挑战是如何保护所有的结构化信息，并且能够按任意字段子集来重建记录。下一步我们讨论数据模型，然后是算法和查询处理。

【译者总结】前几部分最需要关注的其实是图1中的嵌套数据和列状存储格式（columnar representatin of nested data）。这是Dremel提升性能的核心理论，而作者没有对此图着重强调，其实对图1右边列状存储的理解是攻克此篇论文的关键。

3.数据模型

【译者预读】有经验的程序员都知道理解一个系统的第一步就是理解它的数据模型，所以此章节可称之为论文最核心的部分之一。其数学公式对于广大coder不很直观，但其实并不复杂，就如图2中描述的结构一样，本质上和JSON、XML描述的数据结构没有区别，就是一种嵌套的、定制化的数据结构。需要着重理解的是在下面章节会频繁使用的几个名词和基础知识。比如记录（record）、字段（field）、列（column）等。记录（record）就是指一条完整的嵌套数据，如果是在DB中一条记录就是一行（row）数据。字段和列在大部分情况下指的是同一个概念，比如图2中Name、Language等，它们是结构中的一个字段（field），将来存储时就是一个列（column）。比如在

Google里爬虫抓来的一个网页（Document）的数据就是一条记录，而将其结构化之后其中的Forward链接、Url就是字段（或列）。所谓的列状存储其实就是将原始记录按字段切分，各个字段的数据独立集中存储（比如将所有记录中Name.Url这一列的值放在一起存储）。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，0到1个值；repeated（*），表示重复，0到N个值，等。其中repeated和optional类型是非常重要的，作者会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的描述出原始的数据。最后还需要补充两个术语，一是column-stripe，表示图1右边按列存储的一堆列值（列“条”，某个列下顺序存储的一长条数据）；另一个是在论文中广泛使用的路径表达式，xxx.xxx.xxx，其作用类似于XML中的XPath，比如Name.Language.Code，就表示图2中的code字段，因为是在树状结构中，用这样的path能够准确的描述其位置。

在此章节中我们介绍Dremel的数据模型以及一些后续将会用到的术语。这个在分布式系统中经常面对的数据模型(‘Protocol Buffers’ [21]）在Google使用广泛，也提供了开源实现。这个数据模型是基于强类型嵌套记录的。它的抽象语法是：

$\llcorner \pi = dom \mid \langle A1 : \pi\llcorner\ [\llcorner *|?], \ldots, An : \pi\llcorner\ [*|?] \rangle$

π是一个原子类型（一个int、一个string…比如DocId）或者记录类型（指向一个子结构，比如Name）。在dom中原子类型包含整型、浮点数、字符串等等。记录则由一到多个字段组成。字段i在一个记录中命名为Ai，以及一个标签（比如(?)或(*),指明该字段是可选的或重复的…）。重复字段（*）表示在一个记录中可能出现多次，是多个值的列表，字段出现的顺序是非常重要的。可选字段(?)可能在记录中不出现。如果不是重复字段也不是可选字段，则该字段在记录中必须有值，有且仅有一个。

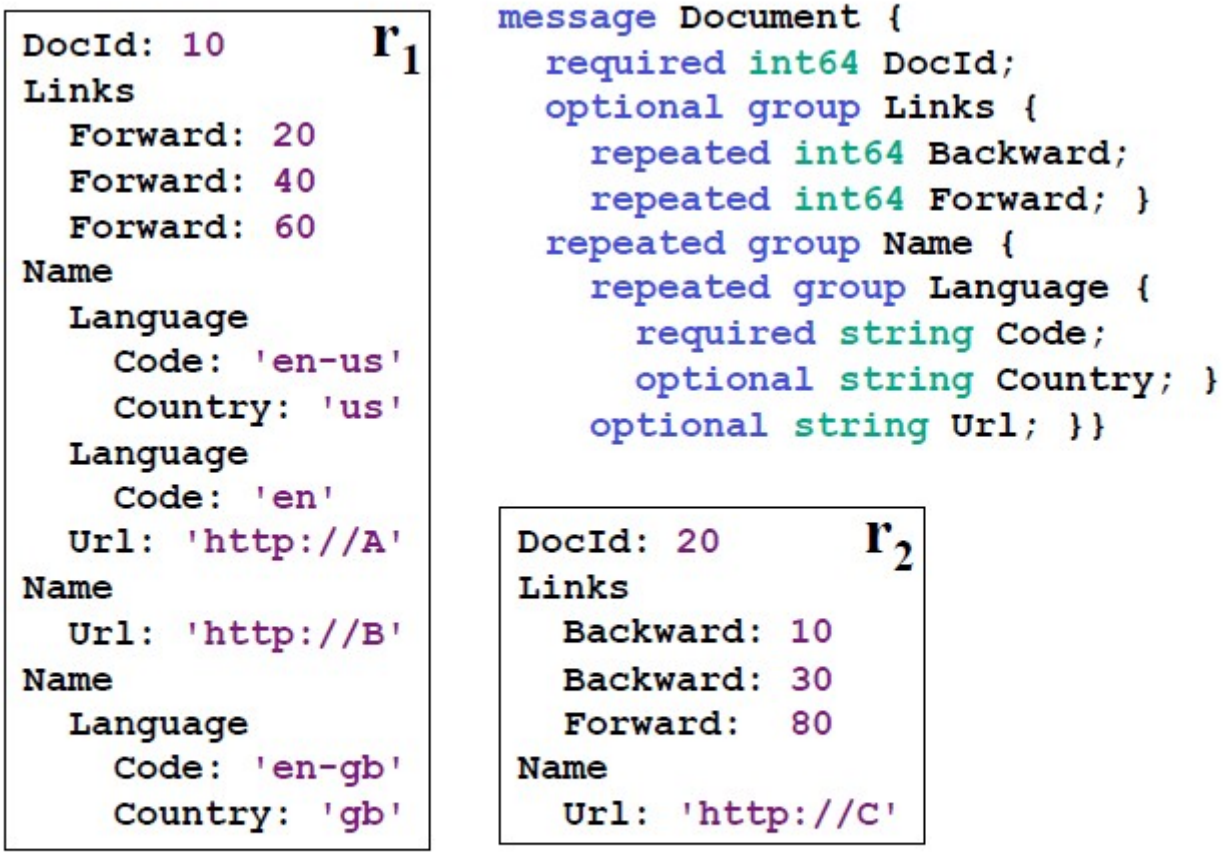


Figure 2: Two sample nested records and their schema

图2进行了举例说明。它描述了一个叫Document的schema，表示一个网页。schema定义使用了[21]中介绍的具体语法。一个网页文档必有整型DocId和可选的Links属性，包含Forward和Backword列表，列表中每一项代表其他网页的DocId。一个网页能有多个名字Name，表示不同的URL。名字包含一系列Code和（可选）Country的组合（也就是Language）。图2也展现了两个示例记录，r1和r2，遵循上述schema。我们将使用这些示例记录来解释下一章节涉及到的算法。
schema的字段定义按照树状层级。一个嵌套字段的完整路径使用简单的点缀符号表示，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。而且有为C++、Java等语言打造的代码生成工具。通过使用标准二进制on-the-wire结构，实现跨语言互操作性，字段值按它们在记录中出现的次序被顺序的陈列。这样一来，一个Java编写的MR程序能利用一个C++库暴露的数据源。因此，如果记录被存储在一个列状结构中，快速装配就成为了MR和其他数据处理工具之间互操作性的重要因素。

4. 嵌套列状存储

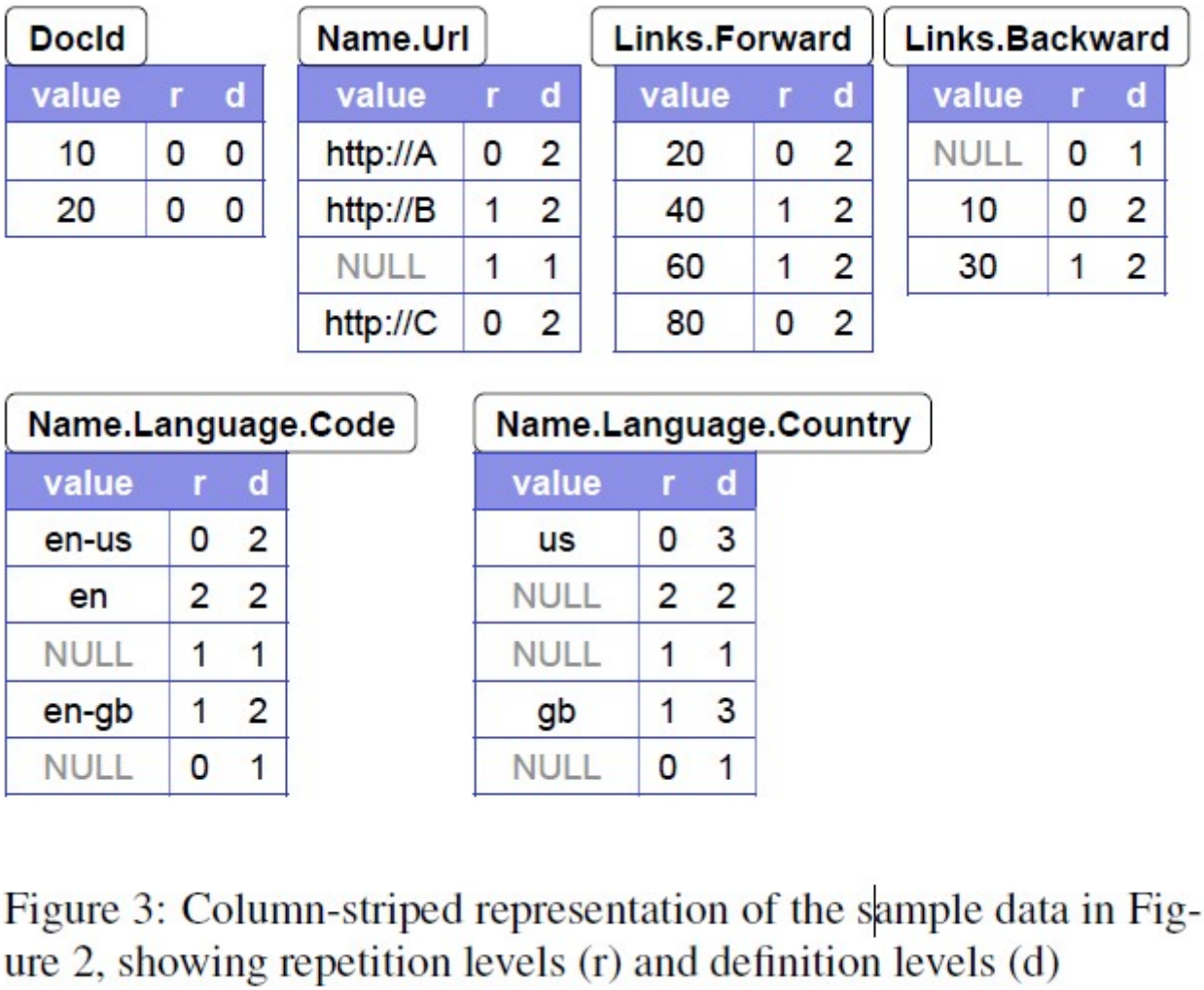
如图1所示，我们目标是连续的存储一个字段的所有值来改善检索效率。在本章节中，我们面对下列挑战：一个列状格式记录的无损表示（章节4.1），快速encoding（章节4.2），高效的记录装配（章节4.3）。

4.1 重复深度、定义深度

只有字段值不能表达清楚记录的结构。给出一个重复字段的两个值，我们不知道此值是按什么‘深度’被重复的（比如，这些值是来自两个不同的记录，还是相同的记录中两个重复的值）。同样的，给出一个缺失的可选字段，我们不知道整个路径有多少字段被显示定义了。因此我们将介绍重复深度和定义深度的概念。图3概述了所有原子字段的重复和定义深度以供参考。

【译者注】读者请重新审视一下图1右边的列状存储结构，这是Dremel的目标，它就是要将图2中Document那种嵌套结构转变为列状存储结构。要实现这个目标的方式多种多样，而此章节中Dremel信心满满的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，重复深度和定义深度。因为Dremel会将记录肢解、再按列各自集中存储，此举难免会导致数据失真，比如图2中，我们把r1和r2的URL列值放在一起得到[“http://A”,"http://B","http://C"]，那怎么知道它们各自属于哪条记录、属于记录中的哪个Name…这里提出的两个深度概念其实就是为了解决此失真问题，实现无损表达。

【译者YY】在翻译上面一段文字的译者其实感觉很突兀，原文作者试图摆出一个难题来引发读者的思考（只有一个赤裸裸的字段值如何弄清楚它所属的记录和结构），但是像我这样按部就班的人，读到这里脑子里思考的是一些更浅显的问题。上面论文中虽然提到“列状存储已经证明了它适用于扁平的关系型数据”、“Dremel希望按字段连续的存储所有值来提升检索效率”，但都是一笔带过，没有详述这么做为何能提升检索效率？提升检索效率的方法多种多样，这么做是不是唯一的、最好的方法？Dremel作者是怎么一步步想到这个方法的（不要告诉我就是灵光一现、一挥而就的）？作者之所以省略，应该是有其他论文早就证明、推导过列状存储的优势和诞生过程。但在此文中直接将其面临的细节问题搬上台面，引出两个陌生的深度概念，不禁略显突兀，让人困惑。这里会先保留这些困惑，直译原文，在此节结束的译者YY环节，译者将尝试与拥有同样困惑的读者一起，YY一下个中奥妙。



DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Figure 3: Column-striped representation of the sample data in Figure 2, showing repetition levels (r) and definition levels (d)

重复深度。注意在图2中的Code字段。可以看到它在r1出现了3次。‘en-us’、‘en’在第一个Name中，而‘en-gb’在第三个Name中。结合了图2你肯定能理解我上一句话并知道‘en-us’、‘en’、‘en-gb’出现在r1中的具体位置，但是不看图的话呢？怎么用文字，或者说是一种定义、一种属性、一个数值，诠释清楚它们出现的位置？这就是重复深度这个概念的作用，它能用一个数字告诉我们在路径中的什么重复字段，此值重复了，以此来确定此值的位置（注意，这里的重复，特指在某个repeated类型的字段下“重复”出现的“重复”）。我们用深度0表示一个纪录的开头（虚拟的根节点），深度的计算忽略非重复字段（标签不是repeated的字段都不算在深度里）。所以在Name.Language.Code这个路径中，包含两个重复字段，Name和Language，如果在Name处重复，重复深度为1（虚拟的根节点是0，下一级就是1），在Language处重复就是2，不可能在Code处重复，它是required类型，表示有且仅有一个；同样的，在路径Links.Forward中，Links是optional的，不参与深度计算（不可能重复），Forward是repeated的，因此只有在Forward处重复时重复深度为1。现在我们从上至下扫描纪录r1。当我们遇到’en-us’，我们没看到任何重复字段，也就是说，重复深度是0。当我们遇到‘en’，字段Language重复了（在‘en-us’的路径里已经出现过一个Language），所以重复深度是2.最终，当我们遇到’en-gb‘，Name重复了（Name在前面‘en-us’和‘en’的路径里已经出现过一次，而此Name后Language只出现过一次，没有重复），所以重复深度是1。因此，r1中Code的值的重复深度是0、2、1.

【译者注】树的深度很好理解，根节点是0，下一级就是1，再下一级就是2，依次类推。但重复深度有所不同，它skip掉了所有非repeated类型的字段，也就是说只有repeated类型才能算作一级深度。这么做的原因是在已知schema的情况下，对于重复深度这个值而言，只需要repeated类型的参与就够了（够下面的split和装配算法所需了），没必要按照完整的schema树来计算深度值。

要注意第二个Name在r1中没有包含任何Code值。为了确定‘en-gb’出现在第三个Name而不是第二个，我们添加一个NULL值在‘en’和‘en-gb’之间（如图3所示）。在Language字段中Code字段是必须值，所以它缺失意味着Language也没有定义。一般来说，确定一个路径中有哪些字段被明确定义需要一些额外的信息，也就是接下来介绍的定义深度。。

定义深度。路径p中一个字段的每个值，尤其是NULL，都有一个定义深度，说明了在p中有多少个可选字段实际上是有值的。例如，我们看到r1没有Backward链接，而link字段是定义了的（在深度1）。为了保护此信息，我们为Links.Backward列添加一个NULL值，并设置其定义深度为1。相似的，在r2中Name.Language.Country定义深度为1，而在r1中分别为2（‘en’处）和1（‘http://B’处）。

定义深度使用整型而不是简单的is-null二进制位，这样叶子节点的数据（比如，Name.Language.Country）才能包含足够的信息，指明它父节点出现的情况；在章节4.3给出了使用该信息的具体例子。

【译者注】定义深度从某种意义上是服务于重复深度的。在论文中其实有一个非常重要的理论介绍的不是很明显，只是简单的用sequentially、contiguously这样的单词带过。这个理论就是在图1右边的列状存储中，所有列都是先存储r1，后存储r2，也就是说对所有的列，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，那当某条记录的某一列的值为空时就不能简单的跳过，必须显式的为其存储一个NULL值，以保证记录顺序有效。而NULL值本身能诠释的信息不够，比如记录中某个Name.Language.Country列为空，那可能表示Country没有值（如‘en’），也可能表示Language没有值（如‘http://B’），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，能够准确描述出此信息。

上面大概提到的encoding保证了record的结构是无损的。这个比较好理解，此处就不过多介绍证明过程了。

Encoding（编码）。每一列被存储为块的集合。每个块包含重复深度和定义深度（下文统称为深度）并且包含字段值。NULLs没有明确存储因为他们根据定义深度可以确定：任何定义深度小于重复和可选字段数量之和就意味着一个NULL。必须字段的值不需要存储定义深度。相似的，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。

【译者总结】这里又提到了块（block）等概念，其实论文应该简而言之——图3中那多张类似“表”的结构（长得很像一张Table，暂称其为“表”，无伤大雅），一张“表”就是一个column-stripe，就是块（block）集合，“表”中的每一行就是一个block，就是图1右边所示的列状存储。物理上像一张张独立的“表”，而逻辑上可以做到图1右部所示的树状、列状结构。在后面装配状态机算法一节中读者能对此有较深理解

【译者YY】读完原文章节后，这里YY一下上面提到的种种困惑

大家都知道任何的技术方案都不是空想出来的，肯定是因为某些痛处催生优化而得来的。译者尝试YY一下Dremel的推导过程：

step1. 首先，不考虑任何性能、优化，也不考虑分布式环境，只想要实现功能，最直接的做法就是按记录存储，比如把一个爬虫抓来的一个Document（如图2中的r1、r2），直接存储到一个GFS的文件中。查询时读取出必需的文件，解析为结构化数据，查询出结果。这样做肯定是能实现功能的，但是我们不会这么做，因为它的劣势十分明显——我只需要读取r1中的Name.URL信息，这里却需要把整条记录都读出来，无用数据远超过有效数据，是性能的极大浪费。（其实也就是论文中一直强调的面向记录存储的劣势）

step2. 第一步中失败就失败在存储时数据是非结构化的（存储时非结构化就意味着需要读取整条数据然后在内存中解析为结构化数据），那当前的优化目标就是做到在存储介质里数据就是结构化的（这样就可以按结构只读取出必要的数据）。不用想的太远，最经典的结构化存储就是众人皆知的关系型数据库，它的表、列、行、关联等概念足以在存储时就按实现数据结构化，而且同样能做到无损。对于嵌套型数据，关系型数据库也早有设计表结构的定式了（其实就是一系列一对多的表结构），以Name.Language.Country这样的路径为例，就三张表，Name、Language、Country，三表包含自己内部的required字段，同时包含父表的外键体现一对多的关联关系（Country表包含Language_id，Language表包含Name_id）。这样一个老掉牙的设计其实就能实现Dremel的一个重要目标——只读取必需的列。query要统计Country，就只需要遍历Country表，如果还要统计Language字段，那就是Language+Country两表join查询，一点不浪费（要知道Dremel查询过程中对一个column-stripe的遍历也是逃不掉的，就相当于这里遍历一张表了）。

第二步的YY有点不靠谱了，但是并没有跑题，如果不考虑通用性、不考虑为嵌套结构建表多么恶心（事实上利用动态schema将一个嵌套结构翻译成关系表也不是难事），也不考虑酷不酷，为什么不能这么做呢？但是答案还是不能，原因有二。Language+Country这个示例太简单了，假如是Name+Country的统计（比如统计Country是‘xxx’的Name有多少个），问题就暴露的很明显了，除了遍历Name、Country表，还需要涉及到Language表（从Country表只能得到关联的Language，需要三表join查询Name+Language+Country才能得到结果），这就违背了Dremel的目标（只遍历必需的表）。改变表设计是可以解决该问题——在Country表里增加对Name的外键关联。那就继续往极端情况去发散，假如Name之上还有一层呢？Country下面还有一层呢？这些层都可能会join查询呢？最终你会发现按照这个方向，你需要在所有的表里加上它所有祖先表的外键。不仅如此，上面曾经提到过为避免失真Dremel采用顺序化存储，顺序就相当于一条记录的主键，所有列值都要包含它，那就意味着在这个方案里各张表还要再加上record_id这个外键。这样一来已经足够令人无法直视了（光是冗余的外键存储就浪费了很大的空间）。第二个原因其实很简单，即使能动态schema、动态控制表结构，也不够通用，较难扩展，不适合通用数据分析平台。

step3. 经过上面对第二步的纠结，我们发现摆在面前的难题其实就是2个，一是要解决每张表上可能无穷无尽的外键，二是这个方案要足够通用化。再回过头看看Dremel最终采用的方案，也许你会发现它其实就是在第二步上做了两个天才的改良：第一，用重复深度+定义深度+顺序这三剑客取代所有外键；第二，表设计时不区分required、repeated等类型，一视同仁，都设计为字段值+重复深度+定义深度这样三列。对于第一个改良，我只能说这三剑客确实是神器，它们足够为任意两张“表”的数据建立关联关系（具体是如何做到请看下面4.3中的状态机算法），足以取代繁杂的外键；对于第二个改良，其实也是为了支持通用的结构和算法而妥协的结果，论文中不止一次的提到那两个深度并不是对所有字段都是必须的，比如DocId字段的r和d永远都是0（如果在关系型数据库中设计表的话DocId只会作为某张表的一个列而不是独立成为一张表）、所有字段非NULL的定义深度永远都相等，这些造成的些许浪费是为了通用化所付出的代价，但是问题不大，只要在存储、计算时稍作手脚就可以尽量避免浪费（上面encoding一节提到如何做手脚）。

上面3个step的推导看似毫无章法，其实是逻辑紧密的，代表了译者这样一个普通coder为了实现一个目标而不断反省优化的过程，并没有任何跳跃性的思维，除了step3中那两个改良，不是译者的YY水平所能驾驭的。我这里也妄自揣测一下，Google的天才们想出这样的方案可能是基于两条路线：一是对数据分析计算过程进行了高等级的抽象，建立了数学模型，帮助了推导过程（数学题的好处就是它大部分情况下是有解的）；另一种就是为了避免存储record_id、避免处理复杂的外键关

联，得出按顺序存储、按顺序遍历的思路，通过在“顺序”二字上做足文章（各column-stripe中，记录间是按固定顺序的，那记录内也可以按由上而下的固定顺序，扫描时把“顺序”发挥到极致），推导出4.3中状态机算法的大概流程，剩下最后一道难题——面前只有一个字段值，没有任何外键（关联信息），仅仅知道它和其他字段值都是按严格顺序存储的，怎么能知道它属于哪条记录以及在记录内的确切位置？针对这一问题最终推导出重复深度和定义深度的概念（在4.1刚开头，作者就直接提出了摆在他面前的这最后一道难题去引读者入戏——“只有字段值不能表达清楚记录的结构……这些值是来自两个不同的记录，还是相同的记录中两个重复的值？……”）。但对于按部就班、接受不了跳跃性思维的译者来说，还是希望论文里能详细介绍这最后一道难题之前的推导过程的——为什么要按照列状结构、为什么把记录拆解的这么零散、无损表示的方法有很多种为何要选择这一种…… 所以才有如上YY，仅供读者参考和矫正。另外论文中曾经提到“列状存储已经证明了它适用于扁平的关系型数据”，这也是为什么译者会联想到基于关系型数据库遇到的问题进行推导。

4.2 分割记录为列状存储

上面我们展示了使用列状格式表达出记录结构并进行encoding。我们要面对的下一个挑战是如何高效率制造column-stripe以及重复和定义深度。计算重复和定义深度的基础的算法在Appendix A中给出。算法遍历记录结构然后计算每个列值的深度，为NULL时也不例外。在Google，经常会有一个schema包含了成千上万的字段，却只有几百个在记录中被使用。因此，我们需要尽可能廉价的处理缺失字段。为了制造column-stripe，我们创建一个树状结构，节点为字段的writer，它的结构与schema中的字段层级匹配。基础的想法是只在字段writer有自己的数据时执行更新，而不尝试往下传递父节点状态，除非绝对必要。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 记录装配

【译者预读】遍历column-stripe时，面前是赤裸裸的字段值（比如‘en’）和两个int（重复深度、定义深度），没有任何的关联信息，怎么知道它属于哪条记录？处于记录内的什么位置？这就是本章节状态机算法要解决的问题。译者认为此算法的核心在于“顺序”二字，在没有任何关联信息的情况下，记录存储顺序就是record的主键，record内由上而下的字段顺序就是位置，而两个int就是判断顺序的唯一线索。

从列状数据高效的装配记录是很重要的。拿到一个字段的子集，我们的目标是重组原始记录就好像他们只包含选择的字段，其他列就当不存在。核心想法是：我们为每个字段创建一个有限状态机（FSM），读取字段值和深度，然后顺序的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。重复深度驱动状态变迁。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。一个FSM状态变化的始终就是一条记录装配的全过程。

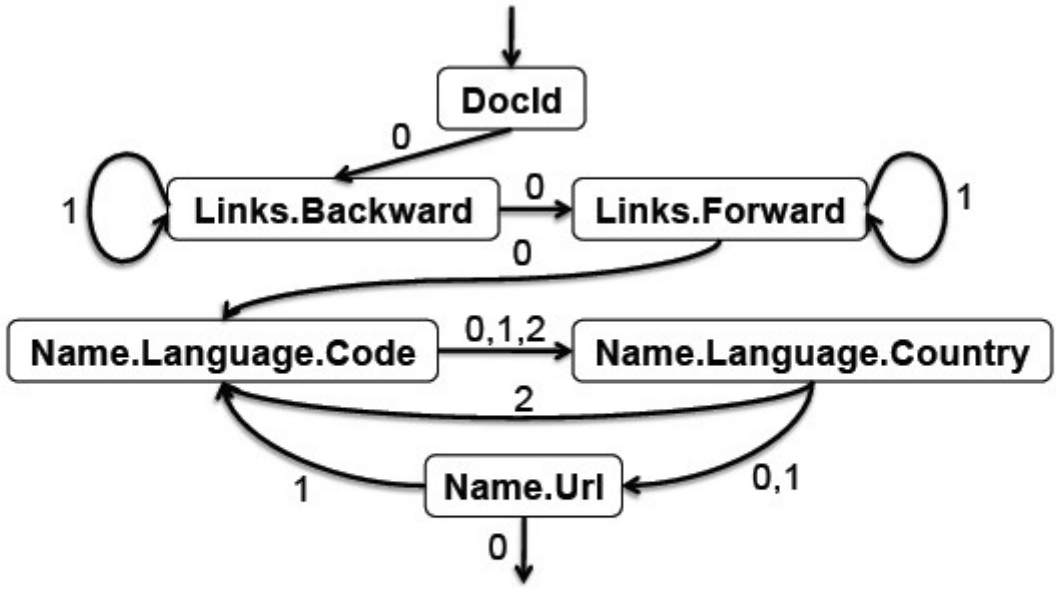


Figure 4: Complete record assembly automaton. Edges are labeled with repetition levels.

图4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转变到Links.Backward。获取完所有重复字段Backward的值，FSM跳向Links.Forward，依次类推。记录装配算法细节在Appendix B中。

【译者注】由于Appendix B的存在（原始论文中对核心算法都附带了源代码和解释，可在原文中查阅），这里对状态跳转的介绍过于简单，所以稍作补充。首先要确定3个思路：第一，所有数据都是按图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的读取某些“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… 那会进入第一个循环读取并装配出r1，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，可以想到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、并且是哪个层级的祖先。举个例子：

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。

示例还可以举更多，但重要的是从示例中抽象出状态变化的本质，下面一段是论文对该本质的简单描述

FSM的构造逻辑可以这么表示：设置r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r的祖先，然后选择该祖先节点的第一个叶子字段n。这给了我们一个FSM状态变化(f;r)->n.比如，让r=1作为f=Name.Language.Country读取的下一个重复深度。它的祖先重复深度1的是Name，它的第一个叶子字段是n=Name.Url。FSM组装算法细节在Appendix C中。

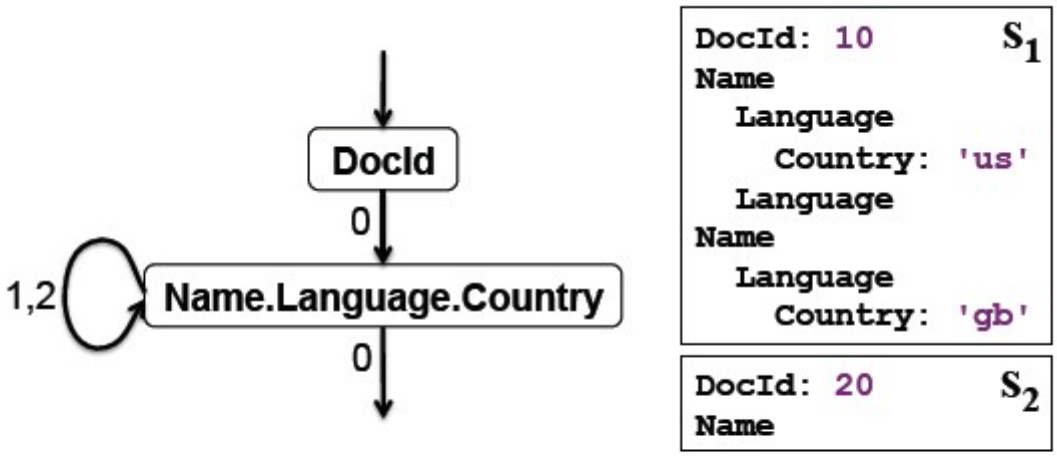


Figure 5: Automaton for assembling records from two fields, and the records it produces

如果只有一个字段子集需要被处理，FSM则更简单。图5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。注意我们的encoding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.

5. 查询语言

```
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; }}}
```

Figure 6: Sample query, its result, and output schema

Dremel的查询语言基于SQL，其实现是定制化设计的，可在列状嵌套存储上高效执行。定义语言严格上不是本文范畴，这里简介一下它的特点。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6描述了一个query例子，执行了投影、选择和记录内聚合等操作(投影-projection,选择-selection，是SQL中的概念，可参考这里，就是SQL select语句中的各个部分)。例子中的query执行在图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。

为了解释query做了什么，需要解释下选择和投影两个操作。在selection操作（where子句）中。可以将一个嵌套记录想象为一个树结构，树中每个节点的标签对应字段的名字。selection操作要做的，就是砍掉不满足指定条件的分支。因此，上述例子中，只有当Name.Url有值且满足正则’^http’才被保留。下一步，在投影操作中，select子句中的每个标量表达式都会投影为一个值，此值的嵌套深度和表达式中重复字段最多的保持一致。所以，Str值的嵌套深度与Name.Language.Code相同。COUNT表达式部分用到了记录内聚合。每个Name子记录都会执行此聚合，将Name.Language.Code出现的COUNT投影为每个Name下的Cnt值，它是一个非负数的64位的整型（uint64）。
此语言支持嵌套子查询，记录内聚合，top-k（排序），joins（多表关联），用户自定义函数等等；下面的实验章节会涉及到其中的一些特性。

6. QUERY的执行

【译者预读】分布式、并行计算是毋庸置疑的，此章节就是描述在数据分布式存储之后，如何尽可能并行的执行计算过程。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，每个层级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。Query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。此方案在论文中称为服务树（serving-tree）

简单起见，我们只讨论在只读系统中执行query的核心思路。很多Dremel查询其实是一次性的聚合，因此我们以这种类型的查询作为重点，并且在下一个章节中用它们进行试验。我们暂不讨论joins、索引、updates（更新操作）等，留在将来详述。

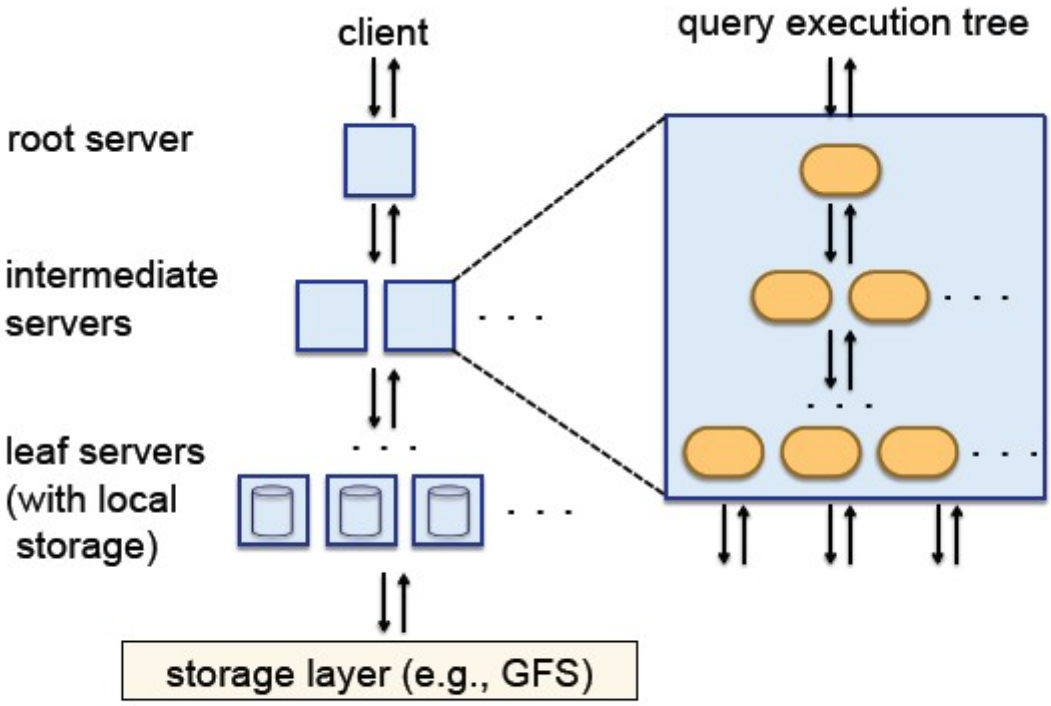


Figure 7: System architecture and execution inside a server node

树结构。Dremel使用一个多层级服务树来执行查询（见图7）。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（可以把一个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T进行了水平拆分，tablet就相当于一个分区），重写查询为如下：

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

【译者注】

R_n^1

不方便编辑，这里使用R(1/n)来表示

R(1/1)到R(1/n)是树中第1层的（1到n）节点返回的子查询结果（根节点是第0层，下一层就是第1层，依次类推）：

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

T(1/i)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。查询任务被一级级的分解成更小的子任务（分区粒度也越来越小），最终落实到叶子节点，并行的对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，也就是说，多个查询通常会被同时执行。一个查询分发器会基于table的分区和负载均衡对query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拷贝不可访问时可以重新调度。
每个query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。
叶子服务器在列状结构中读取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。当一个叶子服务器失效时，请求会通过故障恢复被调度到其他的拷贝上。

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，我们最近证明了，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。
每个服务器有一个内部的执行树，就像图7右边部分。内部树对应到一个物理的query执行过程，包括标量表达式求值。通过优化，绝大部分标量方法会被生成为特定类型代码。在一个聚合查询的执行过程中，首先会有一组迭代器对输入列进行扫描，然后投影出聚合和标量函数的结果，结果上标注了正确的重复和定义深度，不断填充并最终装配出查询结果。详细算法请看Appendix D。一些Dremel查询，比如top-k（排序出前多少个）和count-distinct（去重计数），使用一些大家熟知的算法返回近似的结果（比如[4]）。

7. 实验

【译者预读】论文少不了实验证明。但是这里的实验不仅是简单的证明Dremel多么厉害，而是在Dremel内也采用了不同的方案进行对比，让读者加深对Dremel内部机制的了解。

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×

Figure 8: Datasets used in the experimental study

在这个章节里我们利用几个在Google使用的数据集合对Dremel的性能进行评估，以检验嵌套数据的列状存储到底效率如何。图8中描述了实验使用的数据集合的基本信息。数据集合如果不压缩、不复制，大概占据一个PB的空间。所有table是三向复制的，除了一个双向复制的table，tablets大小不一，从100K到800K。我们开始是在单台机器上检验基础数据访问特征，然后展现列状存储如何优于MR执行过程，最后重点分析一下Dremel的性能。实验运行在两个数据中心的系统节点上，和其

他常规应用一起运行。除非另有说明，执行耗时一般会由5次结果求平均。下面使用匿名的Table和字段名。

本地磁盘。在第一个实验中，我们检查列状和面向记录两种存储的性能，表T1中有1GB的碎片包含大约300K行数据（见图9），分别用两种技术进行扫描。数据存储在一个本地磁盘，在压缩的列状存储中占用375MB，面向记录存储中使用了更重的压缩，但在磁盘占用空间上是相同的。实验在一个双核Intel机器上完成，磁盘提供70MB/S的查询带宽。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。

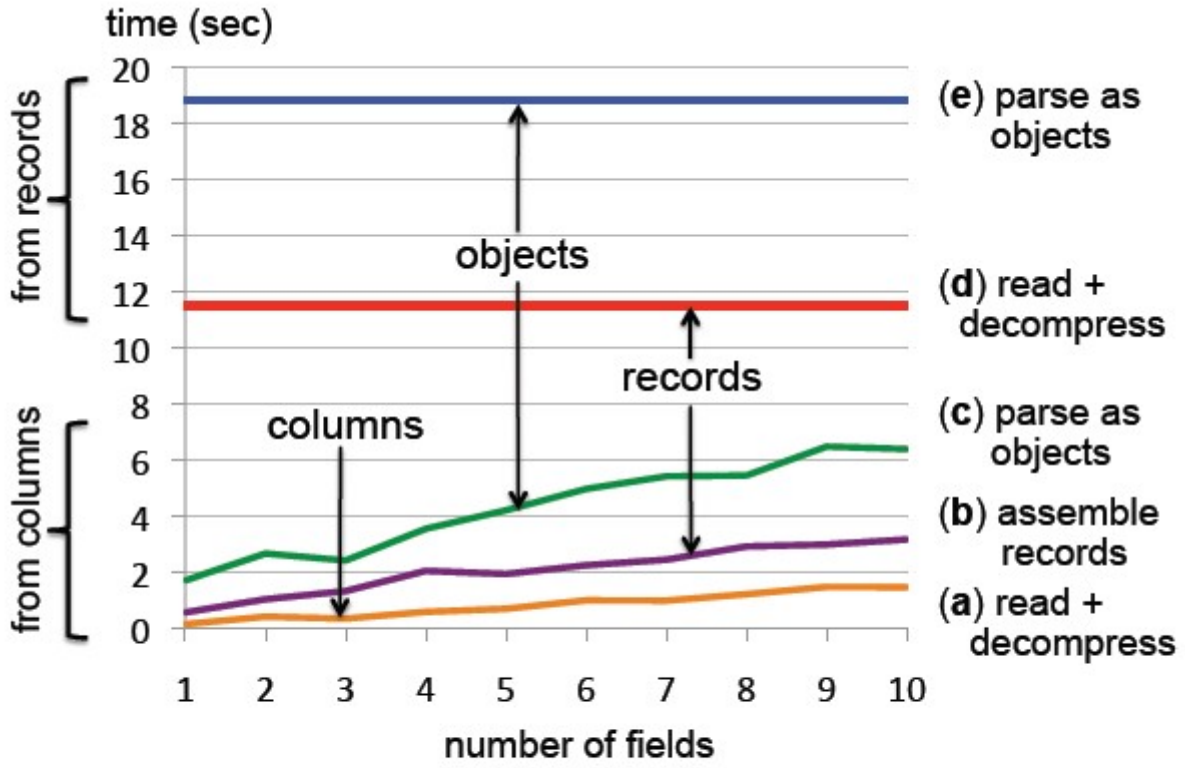


Figure 9: Performance breakdown when reading from a local disk (300K-record fragment of Table T_1)

图中有5个曲线，说明了读取、解压数据、装配、解析记录的耗时。曲线(a)-(c)描述了列状存储的结果。这些图中每个数据点都平均通过了30次测量，每一次都随机选择一定数量的列。曲线（a）展现了读取和解压缩耗时。曲线（b）添加了装配嵌套记录的耗时。曲线（c）展现了将记录解析为强类型的c++数据结构的耗时。

曲线（d）到（e）描述了面向记录存储的各个耗时。曲线(d)展示了读取和解压缩时间。大量时间耗费在解压缩上；不过数据压缩确实减少了大约一半的磁盘读取时间。如曲线（e）指示，解析过程在读取和解压缩时间之上又增加了50%。这些耗时消耗在所有字段上，包括那些并不需要的。
这个实验主要的结论是：当只需读取少量列时，列状存储的性能提升了一个数量级，它的耗时与列的数量成正比。记录装配和解析是昂贵的，每个都可能导致执行耗时翻倍。在其他数据集合的实验中我们看到的趋势大致相同。观察曲线的趋势，很自然会想到一个问题：上下曲线何时交叉？交叉之时，也就意味着面向记录存储开始胜过列状存储。根据我们的经验，交叉点通常在几十个fields时出现，不过在不同数据集合中有所不同，跟是否执行记录装配也有关系。

MR和Dremel。下面的实验是在三者之间进行对比：MR+面向记录存储、MR+列状存储、Dremel+列状存储。这个场景只有一个field被访问，让列状存储的性能收益最明显（列的影响已经

在图9和上面的实验中讨论过了，本实验就排除列数量的影响）。在这个实验中，我们利用表T1的txtField字段，计算每条记录的平均单词数量。MR使用下面的Sawzall[20]程序：

```
numRecs: table sum of int;
numWords: table sum of int;
emit numRecsemit numWords
```

每个记录中，会对input.txtField执行CountWords函数得到单词数量，不断累加得到numWords。在程序运行之后，使用numWords/numRecs得到平均单词数量。在SQL中，这个执行过程可被表示为：

```
Q1: SELECT SUM(CountWords(txtField)) / COUNT(*) FROM T1
```

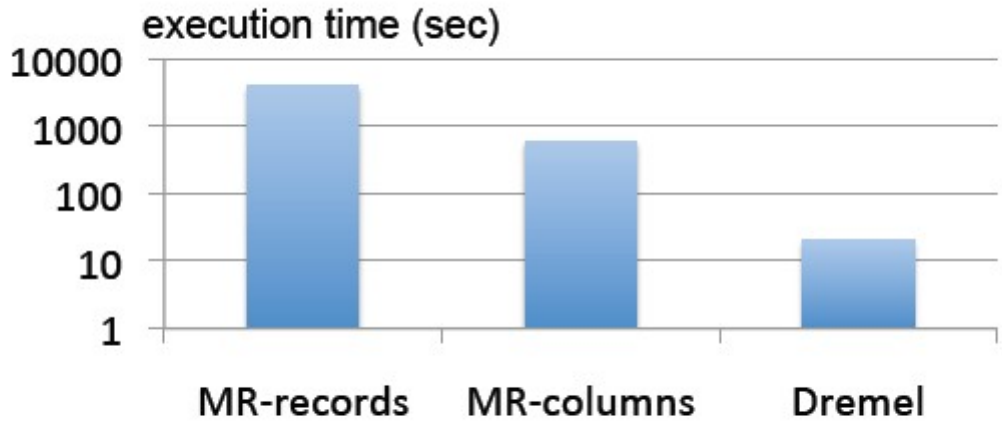


Figure 10: MR and Dremel execution on columnar vs. record-oriented storage (3000 nodes, 85 billion records)

图10展现了两个MR任务和Dremel的执行耗时。两个MR任务被运行在3000个工作节点上。相似的，执行查询Q1的Dremel实例也拥有3000个节点。Dremel和MR+列状存储，读取大约0.5TB的压缩列状数据，MR+面向记录存储读取87TB数据。如图所示，MR从面向记录切换到列状存储后性能提升了一个量级（从小时到分钟）。使用Dremel则又提升了一个量级（从分钟到秒）。

服务树拓扑。在下一个实验里，我们展现了服务树深度对查询执行耗时的影响。对表T2执行两个GROUP BY查询，每个都会对数据进行一次扫描。表T2包含24 billion条嵌套记录。每个记录有一个重复字段item，包含一个数值amount。字段item.amount重复大概40 billion次。第一个查询按country来统计item.amount。

```
Q2: SELECT country, SUM(item.amount) FROM T2
GROUP BY country
```

它返回几百条记录，在磁盘上读取了大约60GB压缩数据。第二个查询在一个文本字段domain上执行GROUP BY，并且执行了一个选择条件。它读取大约180GB，产出大约1.1 million条去重的domain：

```
Q3: SELECT domain, SUM(item.amount) FROM T2
WHERE domain CONTAINS ' .net'
GROUP BY domain
```

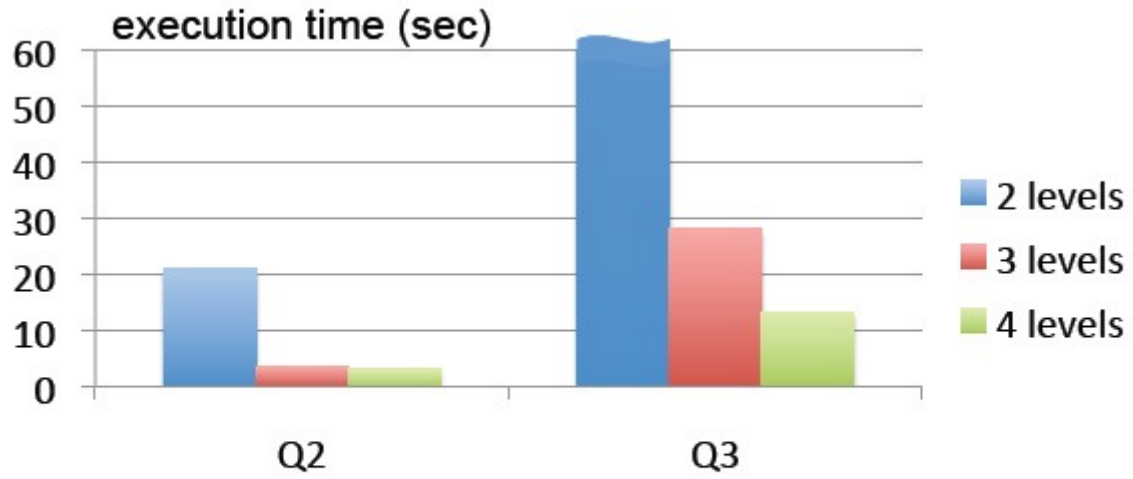


Figure 11: Execution time as a function of serving tree levels for two aggregation queries on T_2

图11展现了两个查询在不同服务器拓扑上的执行耗时。在每个拓扑里，叶子服务器的数量保持在2900，这样我们能假定累积扫描速度是相同的。在2级拓扑（1:2900）中,用一个单一的根节点服务器直接与叶子服务器通讯。在3级拓扑中，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level拓扑是1:10:100：2900.
使用3级拓扑时，查询Q2可以在3秒内完成，增加到4级拓扑时收益就不大了。Q3则不同，从3级到4级耗时减少了一半，因为对于Q3来说4级拓扑可以有效增强它的并行能力。在2级拓扑中，Q3的耗时都超过了图的上限，因为根节点服务器几乎需要顺序的聚合从几千个叶子节点收到的结果（不能并行）。实验证明了当聚合（group by）返回的group越多时（domain有1.1 million条而country只有几百条）多层级服务树的收益就越明显。

tablet视角的曲线图。为了更深的探寻在查询执行过程中发生了什么，我们统计出了图12。此图展示了叶子节点的服务器执行Q2和Q3时处理tablet的速度。当一个tablet被调度到可用slot上执行时开始计时，也就是说耗时不包括等待任务队列的时间，并且消除了其他并发执行的查询造成的影响。以百分比为值的曲线覆盖的总面积就是100%。如图所示，Q2中99%的tablets在1秒内处理完成，Q3中99%的tablet在2秒内完成。

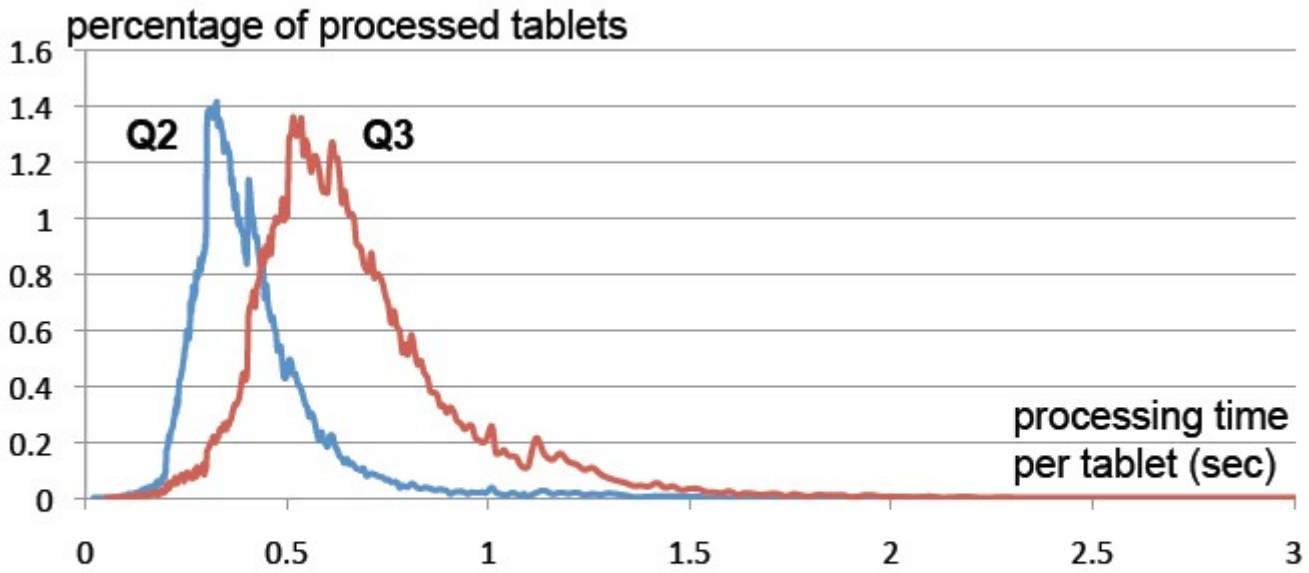


Figure 12: Histograms of processing times

记录内聚合。这个实验我们将在表T3上执行一个带记录内聚合的查询(Q4)：它对a.b.c.d和a.b.p.q.r进行计数分别得到c1和c2，只查询c1大于c2的记录，统计其数量得到最终结果。磁盘中的数据占据70TB，由于column-striping的优势，我们只需读取出13GB，在15秒内就可以完成。没有列状、嵌套等技术的支持，在这样的数据量上运行Q4简直就是噩梦。

Q4 : 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)

可伸缩性。下面实验证明了Dremel如何在一个万亿级记录的表上实现可伸缩、可扩展。这次执行的查询Q5，将从T4中查询出top-20的aid和它们的计数。查询将扫描4.2TB的压缩数据。

Q5: SELECT TOP(aid, 20), COUNT(*) FROM T4
WHERE bid = fvalue1g AND cid = fvalue2g

我们使用4种配置的系统来执行此查询，节点数量范围从1000扩展到4000。图13显示了执行耗时。在每次运行中，总计CPU消耗时间近似相同，大约300K秒，但用户感知的耗时随着系统节点数量增长而几乎线性的减少。这个结果表明Dremel系统规模增大时并不会降低资源利用率（一般分布式系统规模扩展的越大就会消耗更多的资源在业务处理之外的开销上），而执行过程可以更快。

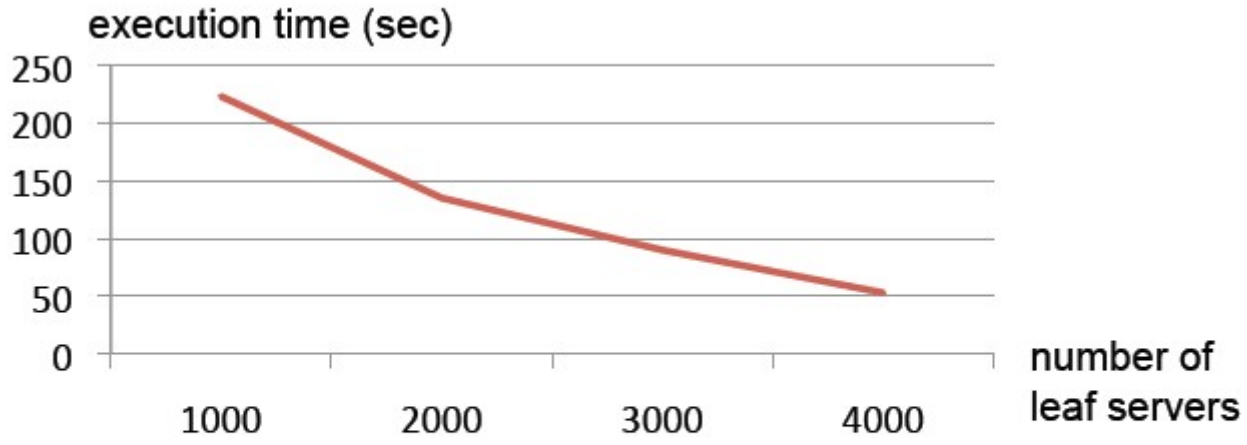


Figure 13: Scaling the system from 1000 to 4000 nodes using a top-k query Q_5 on a trillion-row table T_4

掉队者(Stragglers)。我们最后的实验展示了掉队者的影响。查询Q6将被运行在一个万亿级行数的表T5上。对比其他数据集合，T5只是双向复制的。因此，掉队者减慢执行过程的可能性更高，因为它意味着任务的重新调度。
Q6: SELECT COUNT(DISTINCT a) FROM T5

查询Q6运行在1TB的压缩数据上。待查询字段的压缩比大约是10。如图14所示，每个slot对每个tablet的处理耗时中，99%是低于5秒的。然而，一小部分的tablet花费了非常长的时间，减慢了查询响应时间（从少于一分钟到好几分钟），系统节点规模为2500个。下一章节将对我们的实验结果和教训进行总结。

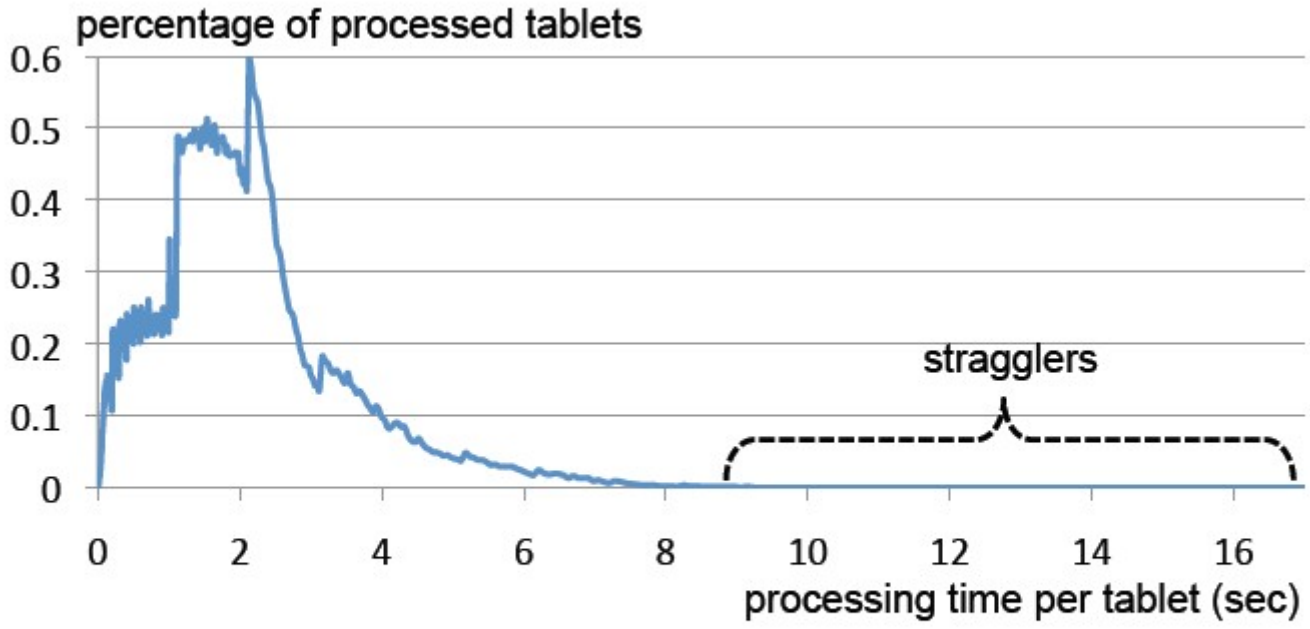


Figure 14: Query Q_5 on T_5 illustrating stragglers at 2× replication

8. 观察和结论

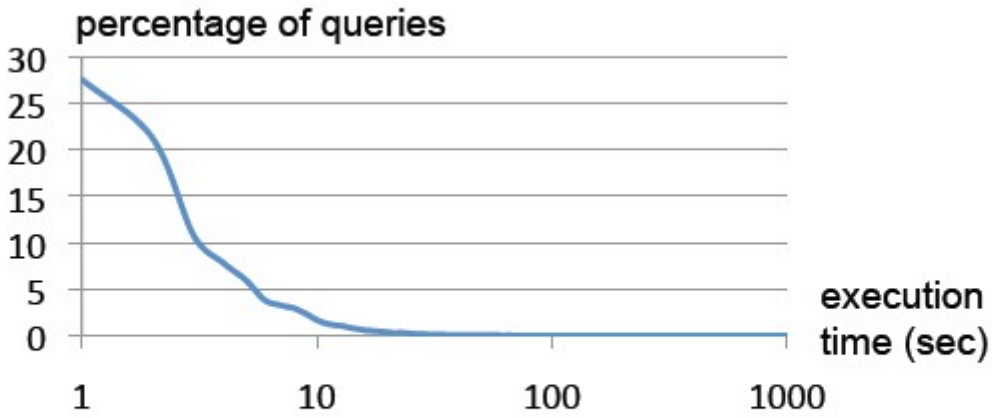


Figure 15: Query response time distribution in a monthly workload

Dremel每个月扫描千之五次方条记录。我们采样了某个月的查询记录，统计出耗时分布曲线。如图15所示，大部分查询低于10秒，在交互型查询的耗时容忍范围内。一些查询会在共享集群上执行接近于100 billion条记录每秒的全量扫描，在专用机器上这个值还要更高。通过对上述实验数据进行观察，我们可以得到如下结论：

- 我们可以在磁盘常驻的数据集合上对万亿级记录执行基于扫描的查询，并达到交互式速度。
- 在几千个节点范围内，列数量和服务器数量的可伸缩性、可扩展性是接近线性的。
- MR也可以从列状存储中得益，就像一个DBMS。
- 记录装配和解析是昂贵的。软件层（在查询处理层之上）最好被优化，能够直接消费面向列的数据
- MR和查询处理可以互为补充；一个层的输出能作为另一个的输入。
- 在一个多用户环境，规模较大的系统能得益于高性价比的可伸缩能力，而且本质上改善用户体验。
- 如果能接受细微的精度损失，查询速度可以更快。
- 互联网级别的海量数据集合可以做到很快速的扫描，但想要花费更少的时间则很困难。
- Dremel的代码库包含少于100K行的C++ Java和 Python 代码

【译者注】后续是“相关工作”和“引用文献”部分，不涉及核心技术内容，这里译者不再赘述了。读者可直接阅读原文。

英文原文：**googleusercontent**，编译：ImportNew - 储晓颖

译文链接：http://www.importnew.com/2617.html

关于作者： 储晓颖