# 树状大数据管理系统Steed

陈世敏
中国科学院计算技术研究所

**关键词：**树状结构数据　JSON　Protocol Buffers　Steed

## 什么是树状结构数据?

树状结构数据类型 (Tree-Structured Data Type) 可直观地表达高级程序设计语言中类 (class)、结构体 (struct) 等丰富的结构，能够简洁地支持嵌套、多值和缺值，已被广泛应用于社交网络数据服务、Web 服务、数据交换格式、分布式系统协议、物联网等，是一种重要的大数据类型。实践中常见的树状结构数据类型有 JSON[1](JavaScript Object Notation)、Protocol Buffers[2] 等。JSON 是 JavaScript 语言标准的一个子集，常常作为数据输出和数据交换的类型。Protocol Buffers 是由谷歌推出的一种数据类型，用于实现分布式系统内部通信协议的数据格式，也是谷歌的 Dremel[3] 和 BigQuery[4] 等大数据系统的数据类型。

下面列出了一个 JSON 类型的简化推特 (Twitter) 数据记录：

```
{“geo”:{“coordinates”:[-7.1,13.4]},
 “retweet_count”:0,
 “user”:{“statuses_count”:28156,
         “favourites_count”:0,
         “followers_count”:5740,
         “id”:32
        }
}
```

JSON 用花括号表示一个记录对象，用逗号隔开对象的多个属性。上述记录在最高层，包含 geo, retweet_count 和 user 三个属性。每个属性由属性名和属性值组成，由冒号隔开。属性值可以是原子类型的字符串、数值、布尔值等，也可以是嵌套的记录或数组。例如，retweet_count 的属性值是一个原子类型，而 geo 和 user 的属性值是一个嵌套的记录对象。数组用方括号表示，例如 geo.coordinates 的属性值。

确切地说，树状数据类型是指如下递归定义的类型 $T_{tree}$：

- $T_{tree}=T_{object}$
- $T_{object}=\{key_1:T_{value_1},...,key_n:T_{value_n}\}$
- $T_{array}=[T_{value},...,T_{value}]$
- $T_{primitive}=string|number|boolean|null$
- $T_{value}=T_{primitive}|T_{object}|T_{array}$

一个树状数据记录可以递归地表达为一棵树，$T_{tree}$ 是树根。树根必须是 $T_{object}$，每个 $T_{object}$ 由属性名 key 和属性值 value 对组成。除了 $T_{object}$，还定义了数组 $T_{array}$ 和原子类型 $T_{primitive}$。而 value 类型则可以递归地定义为原子类型、对象类型或数组类型。树状结构数据类型可以表达多层复杂的嵌套，每层嵌套表现为树的一个内部结点，而树的叶子结点是原子类型。

## 树状结构数据的重要性

在实践中，树状结构数据类型已经被广泛应用。社交网络数据服务推特等输出的数据类型就是 JSON。Web 2.0 RESTFUL 架构中推荐的数据交换格式也是 JSON。许多提供公共数据下载的网站都可以

使用 JSON 来下载数据。Apache Hadoop、HBase 等开源大数据系统中分布式通信协议采用了 Protocol Buffers 来实现。此外，许多物联网单片机芯片 (Arduino, DragonBoard, BeagleBone) 都支持 JSON 格式的数据输出。大量的原始数据是树状结构数据类型。

理论上，树状结构数据类型有潜力替代关系模型成为新的通用数据模型。在 2016 年国际数据工程大会 (ICDE) 上，IEEE TCDE 影响力奖 (Impact Award) 获得者迈克尔·凯里 (Michael J. Carey) 教授作了题为“Beyond Rows and Columns: Is the Fourth Time the Charm?”的大会报告 [5]。报告指出，在关系模型出现之后，数据库研究领域已经认识到了关系模型表达能力的局限性，曾经三次试图推动新的数据模型：面向对象的数据库 (Object Oriented Database)、关系对象数据库 (Object Relational Database) 和可扩展标记语言 (XML)。但由于种种原因，这三次尝试都未竟全功，没能得到推广应用。树状结构数据类型可以看作是对关系模型的第四次变革。凯里教授认为这次变革能够成功的可能性很大，因为 JSON 等类型已经被产业界广泛接受，形成了数十亿美元的市场。产业界的推动结合科研领域的多次尝试和经验积累，可能引发质变，将产生新的得到广泛认可的数据模型。

树状结构数据类型与 XML 相比，具有不同的特点。XML 也可以表达丰富的嵌套、多值结构，但是 XML 的表达引入了许多成本（例如文档类型定义 (DTD) 和标签）。JSON 等树状结构数据类型更加简洁轻量，因此在实践中已经逐渐取代 XML，成为了事实上的标准。此外，单个 XML 文档常常包含很多数据，具有复杂的结构，许多 XML 的相关研究工作正是专注于单个 XML 文档的处理。与此不同的是，单个树状结构数据记录相对简单，而一个数据集常常包涵大量的记录，因此研究的关注点是对大量小记录的处理。

## 现有的树状结构数据处理系统

现有的树状结构数据处理系统主要有下述三种。

**1. 扩展关系型数据库系统。**主流的关系数据库系统 Oracle、Microsoft SQL Server、IBM DB2 和开源数据库系统 PostgreSQL 等都扩展了对 JSON 的支持。基本思路是将整个 JSON 记录以文本或二进制格式存放在关系表的单个属性中，提供内置函数支持 JSON 的解析和访问，从而可以在 SQL 语句中动态地解析 JSON 记录、提取 JSON 属性值，并用于 SQL 查询 [6]。这也是 SQL/JSON 工作组的基本解决方案。但是，经研究发现这种解决方案对数据分析的支持较差。数据分析操作通常只关心 JSON 记录的少量属性，存储和读取整条 JSON 记录会导致大量不必要的 I/O 访问。而且，每次执行 SQL 查询语句，都要动态地解析 JSON 记录，将引入很大的性能开销。

**2. 行式树状结构数据处理系统。**以 MongoDB 为代表的文档存储系统 (Document Store) 支持 JSON 的行式存储和处理 [7]。MongoDB 是 C/C++ 实现的，采用二进制的 BSON 格式存储 JSON 记录。对于 JSON 的属性名，BSON 仍然存储其字符串；而对于 JSON 的原子属性值，BSON 采用二进制存储。MongoDB 提供一组 JavaScript 编程界面，可以执行与 SQL 查询功能相当的操作。和第一种系统相似，因为采用行式存储，数据分析操作会导致大量的 I/O 访问。此外，在访问 JSON 嵌套结构时，MongoDB 需要在每个嵌套层次进行字符串比较，搜索对应的属性名，有较大的性能代价。

**3. 列式树状结构数据处理系统。**谷歌的 Dremel 提出了 Protocol Buffers 数据的列式存储编码方式 [3]。Apache Parquet 是 Dremel 的开源实现，支持 Parquet 格式的文件存取和访问。与 Apache Hive 相结合，就可以将数据存放在 Parquet 列式文件中，并利用 Hive 实现基于 MapReduce 的 SQL 查询，对大规模的树状数据进行分析。由于采用了列式存储，Parquet 可以有效地避免读取不相关属性的 I/O 操作。但其基于 MapReduce 和 Java 的实现影响了查询的效率。

上述三种系统都采用完全通用的设计，为了支持树状结构数据类型可能出现的丰富复杂的嵌套和

多值结构，引入了复杂的算法。例如，为了把多个分别存储的数据列组装还原成原始记录，Dremel 的组装算法要建立一个有限状态自动机，根据自动机和列式文件中的特殊编码完成组装。除了上述讨论的每种系统各自的性能问题，这种完全通用的解决方案本身也存在相对高昂的代价。

## Steed

我们设计并实现了一个树状结构数据管理系统 Steed(System for TrEE structured Data)。Steed 是用 C/C++ 实现的，支持通用的树状结构数据存储和类似 SQL 的查询处理。Steed 同时支持行式和列式的树状结构数据存储，以适应不同类型应用的需要。系统能够自动提取 JSON 的语法树，从而有效地压缩了对属性名的存储。通过分析现实数据，我们发现语法树中 90% 以上从根到叶子的路径是简单路径，于是，Steed 针对简单路径进行了优化，简化和加速了对简单路径的存储和查询处理。图 1 展示了 Steed 的系统结构，主要包括下述三个模块。

**1. 数据解析模块。**该模块读取并解析文本的 JSON 或 Protocol Buffers 数据，将其转化为行式或列式的二进制格式，存储在数据存储模块中。同时，系统建立语法树存储元数据。JSON 记录的数据格式是以标点符号表达的，没有单独的数据类型定义，所以系统将动态地生成语法树，对新出现的属性在语法树中创建新的节点。而 Protocol Buffers 本身提供了数据类型定义，于是系统读取其定义并产生语法树。通过建立语法树，可以把字符串属性名映射为整数 ID，有效地压缩了对属性名的存储。

**2. 数据存储模块。**该模块存储了经过数据解析模块生成的行式或列式二进制数据，支持两种格式数据的相互转换。行式存储采用二进制编码存储属性名和属性值，列式存储将每个属性列分别存放，并基于 Dremel 的编码设计，存放了特殊的结构信息，以支持通用的列组装。

**3. 查询执行模块。**该模块支持类 SQL 的选择（过滤）、投影（提取列）、连接（关联两个数据集的匹配记录）、分组统计、排序等功能，支持行式和列式数据的查询处理。选择和投影的实现对于行式或列式是不同的。行式的选择和投影是对于每条记录依次处理的，而列式的选择操作是分别对相关的列进行过滤，投影操作是选取相关的列，然后进行组装。选择和投影的结果是在内存中的行式中间结果。在此基础上，Steed 实现了连接、分组统计、排序等功能。整个查询执行采用传统关系型数据库

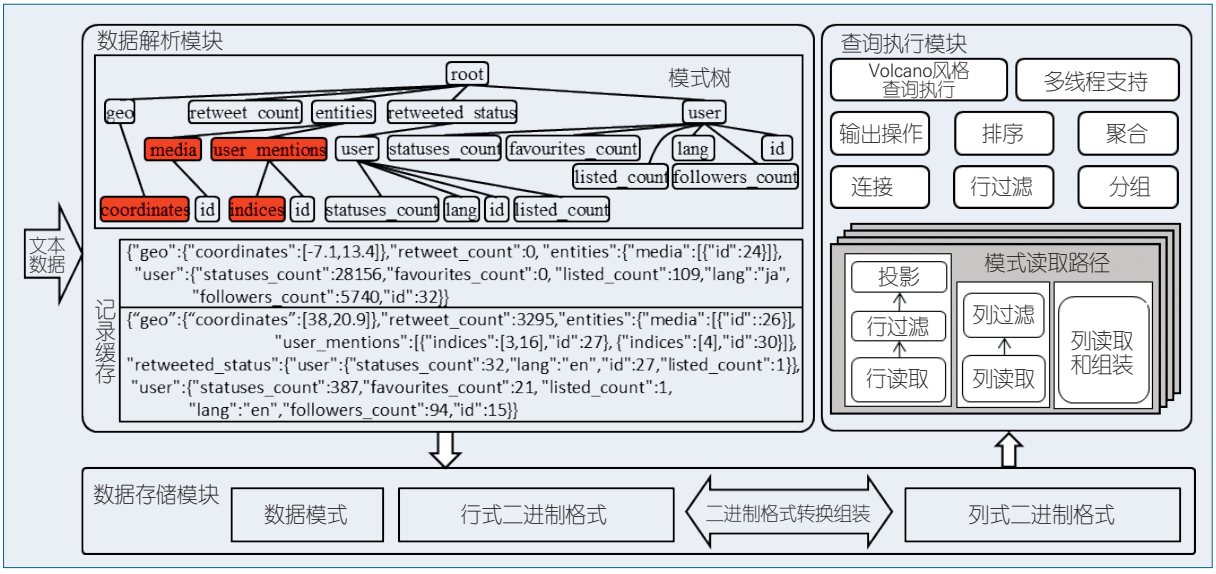



图1　Steed系统结构

中查询树的方式实现。

## 简单路径及其优化

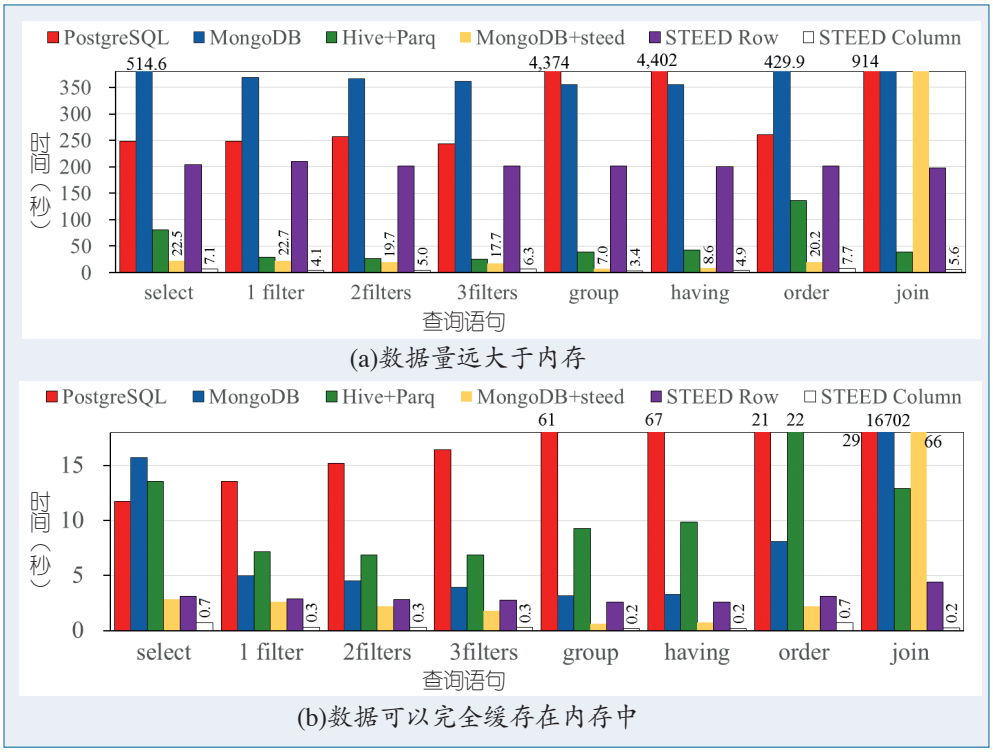
在树状结构数据的语法树中，存在许多从树根到叶子的路径。如果路径上存在多值节点，那么实际的记录中就可能出现多个叶子对应的原子属性值。当路径上的多值节点是两个或多个时，叶子出现的位置必须通过复杂编码来区分。例如，假设路径 a.b.c 中 a 和 b 是多值属性，而 c 可能缺值，那么记录中可能出现 a[0].b[0].c、a[1].b[0].c 和 a[1].b[1].c，却没有 a[0].b[1].c。于是，在 c 的列式存储中，必须特殊记录多值组合的位置，导致了树状结构数据在存储和处理时的复杂性。相反，当路径中不存在多值节点或仅存在一个多值节点时，就有可能进行简化的处理。我们称一条包含最多 1 个多值节点的从树根到叶子的路径为**简单路径**。

通过对现实的树状结构数据的分析，我们发现简单路径大量存在。例如，Twitter 数据的语法树中包含 203 个叶子节点，其中 195 个叶子节点对应的路径是简单的，即 96% 的路径是简单路径。我们分析了雅虎、互联网电影资料库(IMDB)、新浪微博等多种 REST 服务数据，发现超过 99% 的路径是简单的。在其他多类现实数据集中，也发现绝大多数路径是简单路径。

针对简单路径，Steed 优化了列式存储、列式组装和内存行式格式。首先，对于列式存储，可以简化对于多值父节点位置信息的编码。其次，对于列式组装，Steed 对于简单路径的列，可以避免基于有限状态自动机的复杂组装算法，而是依次读取各列相同记录的数据，简化地拼接完成组装。最后，列式组装产生的行式内存结构是连接、分组统计、排序等操作的核心数据结构。Steed 中通用的行式内存结构忠实地保留了原记录的嵌套结构，所以访问深层次的叶子节点属性值时，必须经过多个嵌套层次的查找和跳转。对于简单路径，Steed 采用扁平的内存行式结构，去除了嵌套层次，从而加速了属性值的访问。

## 性能比较

我们通过实验比较了 Steed 和现有的树状结构数据处理系统 PostgresQL, MongoDB, Hive+Parquet 的性能，如图 2 所示。Steed Row 和 Steed Column 分别采用行式和列式数据存储。MongoDB+Steed 是把




(a)数据量远大于内存

(b)数据可以完全缓存在内存中

图2　Steed和现有的树状结构数据处理系统的性能比较

MongoDB 的后端存储引擎替换为 Steed Column，使用列式存储读取数据，转化为 BSON，并使用 MongoDB 现有的内存处理。实验是在单台联想 ThinkCentre M8500t 工作站上运行的，工作站配有一个 3.4GHz 4 核的 Intel Core i7-4770 处理器、16GB 内存和一个 7200rpm 的 SATA 硬盘。在 Twitter 数据上，我们运行多种查询操作，图中从左到右依次是使用 1 至 3 个条件过滤数据集 (1filter, 2filters, 3filters)、分组统计 (group)、在分组统计的结果上进一步过滤 (having)、排序 (order)、连接 (join)。实验包括两种场景：数据量远大于内存和数据可以完全缓存在内存中。

**数据量远大于内存**：在这种情况下，I/O 对查询性能至关重要。因为行式和列式存储的 I/O 量明显不同，所以我们先分开进行比较。考虑采用行式存储的 MongoDB 和 Steed Row，可以看到 Steed 取得了 1.8~2.5 倍的性能提升[1]。这是因为 MongoDB 的 BSON 格式在每条记录中都需要存储属性名的字符串，而 Steed 采用语法树，在每条记录中只需要存储属性名的整数 ID，从而节省了空间，降低了查询需要读取的 I/O 数据量。比较采用列式存储的 Hive+Parquet 和 Steed Column，可以看到 Steed Column 取得了 4.1~17.8 倍的性能提升。这是因为 Hive 使用 Java 和 MapReduce 处理 Parquet 格式的文件，引起了较大的性能代价。MongoDB+Steed 采用了列式存储，比采用行式存储的原始 MongoDB 性能提升 16~51 倍。进一步地，如果把 BSON 格式改变为 Steed 的内存结构，节省了对字符串属性名的比较，采用了更高效的查询处理实现，Steed Column 比 MongoDB+Steed 性能提升 1.8~5.5 倍。总体而言，Steed Column 性能在所有系统中最优，比 Hive+Parquet 提高 4.1~17.8 倍，比 MongoDB 提高 55.9~105.2 倍，比 PostgreSQL 提高 33.8~1294 倍。

**数据可以完全缓存在内存中**：PostgreSQL 常常是最慢的系统，这说明把整条 JSON 记录存放在关系型属性中并在查询处理时动态地解析和访问 JSON 记录，会引起显著的性能代价。Hive 的基于 Java 和 MapReduce 的处理，有一定的性能代价。此外，Steed Column 比 Steed Row 性能好，这是因为 Steed Column 减少了数据的拷贝操作。总之，Steed Column 的性能比 Hive+Parquet 提高 10.5~59.3 倍，比 MongoDB 提高 11.9~22.6 倍，比 PostgreSQL 提高 16.9~392 倍。

## 结语和展望

以 JSON 和 Protocol Buffers 为代表的树状结构数据类型是一种新兴的重要的大数据类型，在实践中被广泛应用，在理论上有潜力替代关系模型成为新的通用数据模型。但是，现有的树状结构数据处理系统存在多种问题，无法高效地支持树状结构数据的分析处理。我们设计实现了一个树状结构数据管理系统 Steed，支持通用的树状结构数据存储和类似 SQL 的查询处理。通过分析现实数据，我们发现实际数据中从树根到树叶的路径有 90% 以上是简单的。针对简单路径，Steed 优化了外存存储、列组装算法和内存行式结构。与现有系统 PostgreSQL、MongoDB、Hive+Parquet 相对比，Steed 对于数据分析类操作普遍有 10~1000 倍的性能提升。 ■

**陈世敏**

CCF专业会员，CCF大数据专家委员会委员、数据库专业委员会委员。中科院计算所研究员。主要研究方向为数据库系统、大数据处理、计算机体系结构。

chensm@ict.ac.cn

## 参考文献

[1] JSON[OL]. http://www.json.org/.

[2] Protocol Buffers. https://code.google.com/p/protobuf/.

[3] Melnik S, Gubarev A, Long J J, et al. Dremel: Interactive analysis of web-scale datasets[C]//*Proceedings of the VLDB Endowment*, 2010, 3(1):330-339.

[1] 不包括连接操作，MongoDB的连接操作实现比Steed row慢1010倍。

[4] Google. An inside look at google bigquery[OL]. https://cloud.google.com/files/BigQueryTechnicalWP.pdf.

[5] Carey M J. Beyond rows and columns: Is the fourth time the charm? [R]. IEEE International Conference on Data Engineering (ICDE), Keynote Speech, 2016.

[6] Liu Z H, Hammerschmidt B, McMahon D. JSON data management: Supporting schema-less development in RDBMS[C]//*Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. New York: ACM Press, 2014: 1247-1258.

[7] MongoDB[OL]. https://www.mongodb.com/.