STEED: An Analytical Database System for TrEE-structured Data

Zhiyi Wang Dongyan Zhou Shimin Chen

State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
{wangzhiyi,zhoudongyan,chensm}@ict.ac.cn

ABSTRACT

Tree-structured data formats, such as JSON and Protocol Buffers, are capable of expressing sophisticated data types, including nested, repeated, and missing values. While such expressing power contributes to their popularity in real-world applications, it presents a significant challenge for systems supporting tree-structured data. Existing systems have focused on general-purpose solutions either extending RDBMSs or designing native systems. However, the general-purpose approach often results in sophisticated data structures and algorithms, which may not reflect and optimize for the actual structure patterns in the real world.

In this demonstration, we showcase Steed, an analytical database System for tree-structured data. We use the insights gained by analyzing representative real-world tree structured data as guidelines in the design of Steed. Steed learns and extracts a schema tree for a data set and uses the schema tree to reduce the storage space and improve the efficiency of data field accesses. We observe that sub-structures in real world data are often simple, while the tree-structured data types can support very sophisticated structures. We optimize the storage structure, the column assembling algorithm, and the in-memory layout for the simple sub-structures (a.k.a. simple paths). Compared to representative state-of-the-art systems (i.e. PostgreSQL/JSON, MongoDB, and Hive+Parquet), Steed achieves orders of magnitude better performance for data analysis queries.

1. INTRODUCTION

Tree-structured data formats, such as JSON and Protocol Buffers, have numerous applications in a wide variety of real scenarios, including social network data feeds, online data services, communication protocols, publicly available data sets, and sensor data. A tree-structured data model can be recursively defined as follows:

$$
\begin{array}{lcl}
T_{value} & = & T_{object} \mid T_{array} \mid T_{primitive} \\
T_{object} & = & \{key_1 : T_{value_1}, ..., key_n : T_{value_n}\} \\
T_{array} & = & [T_{value}, ..., T_{value}] \\
T_{primitive} & = & string \mid number \mid boolean \mid null \\
key & = & string \\
T_{tree} & = & T_{object}
\end{array}
$$

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

A tree-structured data type can be viewed as a schema tree. The top level type is the root of the tree. An object contains a list of key-value pairs, while an array consists of a list of values. A value type is recursively defined as an object, an array, or a primitive type. An object is a non-leaf node. A primitive value is a leaf node. An array is combined with its child node to make the child node a repeated node. (An example tree is shown in Figure 1.)

Compared to the traditional relational data model, tree-structured data formats are capable of expressing much more sophisticated data types, including nested, repeated, and missing values. Data structures (e.g., `struct`, `class`, arrays) in high-level programming languages (e.g., C/C++ and Java) can be easily presented as tree-structured data. Such expressing power is one of the main reasons for the popularity of tree-structured data formats. However, the complexity presents a significant challenge in efficiently supporting tree-structured data formats.

Note that there is a large body of work in the literature on storing and querying XML documents [2, 6]. However, JSON and Protocol Buffers have significantly different characteristics from XML. A single XML document often contains a great many repeated tags. DTDs can allow recursions and cycles [5]. In contrast, records in JSON-like formats are often much lighter weight. The record size is often comparable to that of a relational record. Therefore, while we build on learnings from research on XML, the design focus for JSON-like tree-structured data is on processing a large number of relatively small records, rather than processing complex structures inside a single document as in the case of XML.

We have designed and implemented an analytical database system, called *Steed* (**S**ystem for **tree**-structured **d**ata) (which is descried in detail in our SIGMOD'17 [12] paper). The baseline Steed design supports general-purpose data storage and query processing of tree-structured data in both row and column formats. Based on our analysis of representative real-world use cases of tree structured data, we observe that a majority of the sub-structures in the real-world trees are quite simple: most root-to-leaf paths contain no repeated node or only one repeated node. We call such paths *simple* paths. We propose and implement optimized column storage, column assembling process, and in-memory data layouts for simple paths. Moreover, we compare Steed with State-of-the-art systems that support either JSON or Protocol Buffers, including PostgreSQL with JSON support, MongoDB (which supports JSON natively), and Hive using Parquet file formats for storing columnar Protocol Buffers data. Experimental results show that Steed achieves 4.1–1294x speedups over the state-of-the-art systems running SQL-like data analysis queries.

In this demonstration, we would like to explain the inner working of Steed, compare the performance of Steed with State-of-the-art systems, and invite audience to try out Steed.

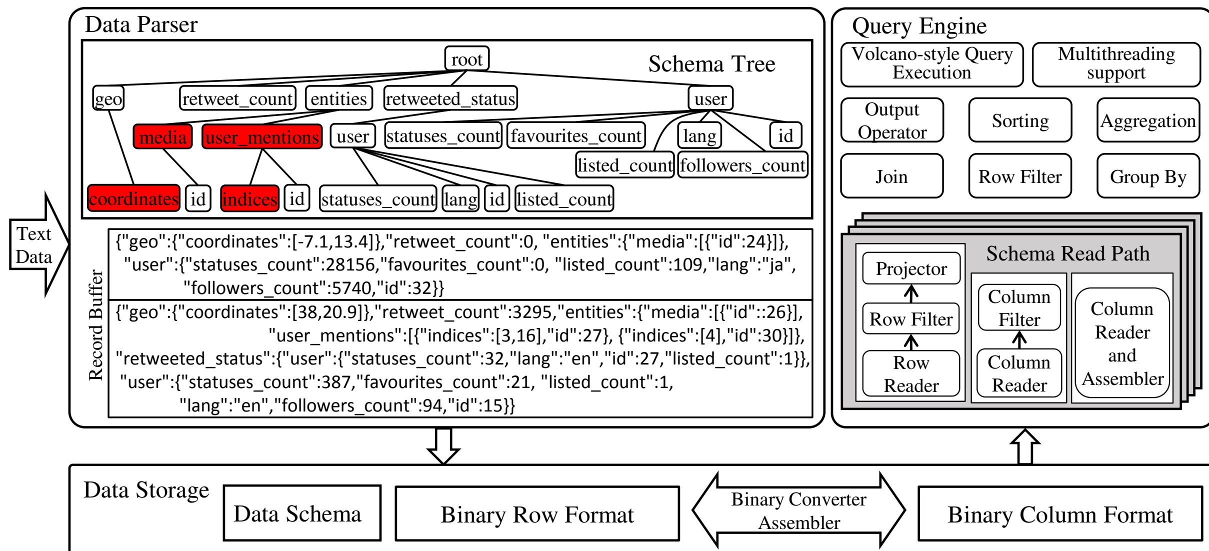

Figure 1: Architecture of Steed. (Two JSON records are used as the running example. The highlighted red nodes are repeated.)

2. RELATED WORK

Previous work has focused on *general-purpose* support for tree-structured formats either (i) by extending RDBMSs or by (ii) designing native systems.

Extending RDBMSs. A tree-structured data record is stored either as shredded fields, or as a whole in a single attribute, or a combination of the two in RDBMSs. Chasseur et al. proposed Argo, a mapping layer that splits a JSON record into a set of (object ID, key, value) tuples corresponding to the leaf nodes in the tree, and stores the shredded fields in the RDBMS [4]. In contrast, Liu et al. proposed to store a JSON record in a varchar/BLOB/CLOB, and use various functions (e.g., extracting scalar values) to work with JSON records in SQL queries [8]. This is the approach taken in the SQL/JSON standard effort and by Oracle [8]. PostgreSQL also stores an entire JSON record into a text or a binary field, but designs a different syntax to use JSON in SQL queries [1]. On the other hand, Tahara et al. proposed a hybrid approach, where a subset of the attributes are materialized as columns and the remainder is serialized into a single binary column in the RDBMS [11]. Recently, Liu et al. proposes to enhance the JSON support in Oracle with an OSON binary format [9], where every OSON document contains a mapping from string field keys to field IDs.

Designing Native Systems. NoSQL document stores (e.g., MongoDB and CouchDB) provide native storage and query support for JSON data. JSON data are stored in row formats. MongoDB defines a binary JSON row format, called BSON, and supports Javascript based query APIs. Moreover, Melnik et al. [10] proposed Dremel, a system that supports general-purpose column data layouts and SQL-like queries for Protocol Buffers data. Column layouts can significantly improve the performance of data analysis. Apache Parquet is an open-source Java-based implementation of Dremel's columnar design. It can be integrated into the Hadoop ecosystem (e.g., Hive) as an input/output format. Furthermore, AsterixDB [3] supports a tree-structured data model, called ADM, and a query language, called AQL, on ADM.

Challenge of General-Purpose Designs. Previous work provides general-purpose designs for various tree-structured formats. However, such designs must consider arbitrarily sophisticated types. Conceptually, trees can range from shallow trees with primitive leaf nodes, to very deep trees with many nested levels and repeated fields. Therefore, a general-purpose solution has to support any kind of tree-structured data, no matter how simple or or complex it is. For example, functions that work with JSON records in RDBMSs must be able to parse arbitrarily complex JSON records. The column design in Dremel must be able to encode and assemble Protocol Buffers data with arbitrarily large number of nested levels and repeated nodes. As a result, these solutions require sophisticated algorithms and data structures, which may not reflect the actual use patterns in the real world and thus may be less efficient.

3. OUR SOLUTION: STEED

Real-World Data Patterns. We have performed an in-depth study of the tree-structured data patterns in the real world by analyzing the tree structures in representative use cases in social network data feeds, online data services, communication protocols in distributed systems, web sites providing downloading services for publicly available data sets, and sensor data [12]. While the numbers of leaf nodes in the tree structures vary greatly, from less than 10 to a few hundred, we find interesting common patterns across the cases. First, the records in a data set typically have similar tree structures. In other words, it makes sense to extract and use schemas even in schema-less data types, such as JSON. Second, the heights of the trees are often not very large, varying from 1 to 8 levels. Third, more importantly, a majority of the root-to-leaf paths are simple paths, containing no repeated node or only one repeated node. Steed aims to optimize for the common patterns.

Steed Architecture. We have designed and implemented Steed, an analytical database System for tree-structured data. Figure 1 shows the architecture of Steed. It consists of mainly three parts: (i) the data parser, (ii) the data storage, and (iii) the query engine.

The data parser takes tree-structured data in text formats as input, such as JSON in the example, or Protocol Buffers. It parses the input and constructs a schema tree. In some cases, such as Protocol Buffers, a data schema is provided along with the data. Thus, Steed constructs the schema tree from the provided data schema. In other cases, such as JSON, there is no explicit data schema. Therefore, Steed learns the schema while reading the data records, and constructs the schema tree on the fly.

The data storage stores binary data and schemas as files in the underlying file system. It supports both binary row format and bi-

nary column format for tree-structured data. The binary row layout faithfully stores the nested structure of a record. It uses the node IDs in the schema tree to replace the string keys in the records in order to save space and improve access efficiency. In contrast, BSON stores string keys in the records. On the other hand, OSON also tries to reduce space by storing field IDs [9]. However, OSON's field IDs are local within a record, thus OSON has to store a mapping table (a.k.a. field-ID-name dictionary) in every record. In comparison, the node IDs in the schema tree in Steed are global for all JSON records in the same collection. As a result, Steed can save more space than both BSON and OSON.

In the column data layout, every leaf node in the schema tree is stored as a column data file. During query processing, one has to assemble relevant columns to reconstruct the trees (or sub-trees). However, this task is non-trivial, for root-to-leaf paths may contain multiple repeated nodes, and there may be missing branches as well. Therefore, it is important to store not only the values in the column but also the tree structure for correctly assembling the columns. Our baseline scheme is based on Dremel [10]. The encoding scheme in Dremel stores two additional values, a repetition level(rep) and a definition level (def), for every column value (as well as every missing branch). The column assembling algorithm is sophisticated. It constructs a finite state machine, where state transitions are based on rep and def values. The assembling algorithm essentially traverses the schema tree from left to right, and may jump back to repeatedly visit repeated nodes.

The query engine supports SQL-select-like queries with select, from, where, group-by, having, and order-by clauses. We have implemented a Volcano-style query execution engine [7], and multi-threading support for common relational operators. Steed can process both row data and column data. The main difference is in the implementation of the filter operators. In the case of the binary row format, the row filter operator processes the records one by one, applying the filtering predicates. Then, the projector preserves only the attributes that are relevant to the query. In the case of the binary column format, Steed instantiates a column reader and a column filter for every column in the query. Then the columns are assembled for the records that satisfy all the filter predicates. As a result, after the selection and the projection, the records are in the baseline binary row layout (regardless of the original storage formats). The remaining operators, including join, group-by, aggregation, and sorting, all process data in the row layout. We assume that main memory is large enough to hold all the data after filtering. Therefore, we employ main memory algorithms for the operators, including a hash-based join operator, a hash-based group-by operator, and a quick-sort based sorting operator.

Optimizations for Simple Paths. We leverage the knowledge of the simple path to simplify the column layout, accelerate the column assembling process, and design more compact and more efficient in-memory record structures. Note that the complexities of the baseline implementation are mainly required to handle the potential sophistication of the tree structures. For the simple paths, such complexities are often not necessary. First, for the column data layout, we can omit or reduce the size of ref and/or def if the column corresponds to a simple path. Second, for the column assembling process, we propose a flat column assemble algorithm that does not require a finite state machine. Third, for the in-memory row layout, we propose a flat structure to reduce the number of nested levels. Note that every nested level introduces a level of indirection. A (binary) search is performed at every level to determine the child node to follow next. Therefore, reducing the number of levels can significantly reduce the cost of accessing individual fields in a record during query processing in memory.

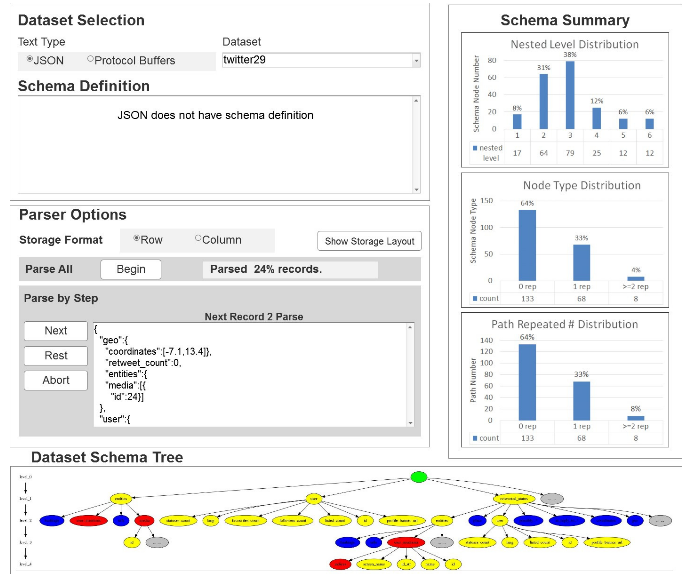

Figure 2: Data parsing.

4. DEMONSTRATION PLAN

The demonstration is structured in three parts to showcase the functionality, the inner working, and performance of Steed highlighted in Section 3.

Part 1: Data Parsing. We offer a gentle introduction to tree-structured data formats and demonstrate the data parsing functionality of Steed. As shown in Figure 2, a user can select a data set and specify its type, i.e. JSON or Protocol Buffers. Several example data sets, including Twitter data sets (downloaded using Twitter API in 2012), will be available. For Protocol Buffers, the user can specify an input schema file, while JSON is schema-less. Then, the user can choose either the row or the column storage format, and press the "Begin" button to start parsing all records. Alternatively, the user can parse record by record. After parsing, we will show the schema tree of the parsed data set and summary statistics about the schema. In this part of the demonstration, audience can examine tree-structured text records in the selected data set, compare them with the schema tree of the data set to get a better understanding of tree-structured data, and study the storage layout of the generated row or column storage format.

Part 2: Query Composition and Processing. We invite the audience to try out the query processing of Steed. As shown in Figure 3, a user can select a dataset that has already been parsed and stored in binary formats. Then, the schema tree of the selected data set will be shown in the UI. The user can compose a SQL-like query statement, where the attributes in the SQL-like statement can be path expressions. A click on a node in the schema tree will add the corresponding path expression to the end of the query statement being composed. The "Add More Dataset" button is used in the composition of join queries. The "Explain" button shows the execution plan for the query, explaining how the query is supported. The "Run" button runs the query. Query results can be shown by pressing the "Show Result" button.

Part 3: Performance Comparison. We showcase the performance of Steed by comparing Steed with row layouts, baseline Steed with column layouts, optimized Steed with column layouts, with the following State-of-the-art systems:

- *PostgreSQL*: PostgreSQL has been extended with JSON support recently. A JSON record is stored either as text (in the `json` type)

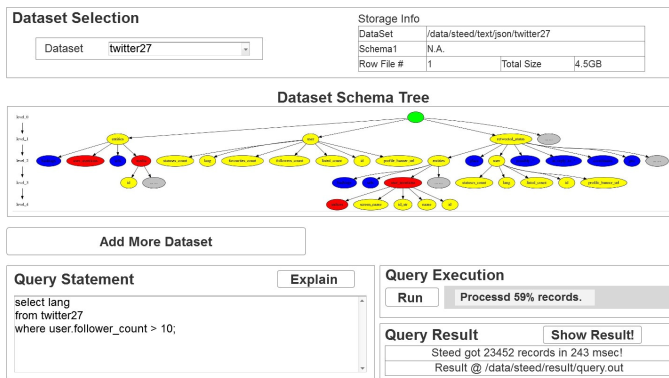

Figure 3: Query composition and processing.

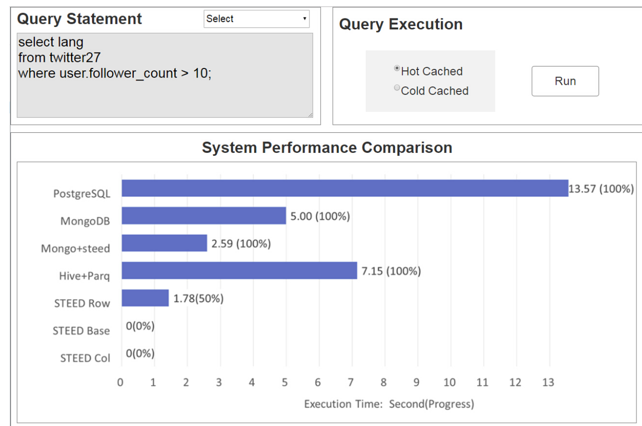

Figure 4: Comparison with state-of-the-art systems.

or as binary data (in the `jsonb` type). We use jsonb because it is more efficient for query processing. PostgreSQL provides a set of operators and functions to access JSON data in SQL queries. In our experiments, we use PostgreSQL 9.5 and issue SQL queries to process JSON data.

- *MongoDB*: MongoDB is an open-source database system that provides native support for JSON documents. It is written in C/C++. MongoDB users include Adobe, Craigslist, eBay, LinkedIn, Foursquare, and so on[1]. MongoDB stores JSON records in a binary format called BSON. We use MongoDB 3.2.8. We load JSON data sets using mongoimport, and use MongoDB's Javascript interface to submit queries.
- *Hive+Parquet*: Apache Parquet implements the Dremel design of storing and assembling columns for tree-structured data. Apache Hive is a popular analytical big data processing system that supports SQL-like queries on top of MapReduce. In this demonstration, we use Hive 1.2.2 on Hadoop 2.6.0 with Parquet as the underlying storage format. Hadoop, Hive, and Parquet are all implemented in Java. Note that Parquet requires the declaration of the data schema. We manually create the required schema and load the data into Hive using a tool called Kite. Then we submit SQL queries using Hive's query interface.
- *MongoDB+Steed*: In addition to the three systems, we also implemented a hybrid system that runs MongoDB on top of Steed. MongoDB stores binary JSON records (a.k.a. BSON) in an underlying storage management system, called wired tiger. We have modified the calling interface between MongoDB and wired tiger to redirect the retrieval of records to Steed. To take advantage of the column layout in Steed, we parse the MongoDB commands to extract all the fields used in a query. The fields are then communicated to Steed for reading only the relevant columns. After obtaining the records, MongoDB performs the actual query processing operations.

As shown in Figure 4, we specify a query in the text box at the top left corner, and press the "Run" button to start the experiments. Then, the query is automatically run on each system one by one. For systems that have finished the query, the figure shows the execution times. For the system that is currently running the query, the figure shows its progress. In this way, audience can easily try different query types and compare the performance of Steed and the state-of-the-art systems.

Acknowledgments. This work is partially supported by the CAS Hundred Talents program, by NSFC project No. 61572468, and by NSFC Innovation Research Group No. 61521092. Shimin Chen is the corresponding author.

[1]According to db-engines.com, in February 2017, PostgreSQL and MongoDB were ranked the 4th and the 5th most popular databases.

5. REFERENCES

[1] PostgreSQL's JSON Support. https://www.postgresql.org/docs/9.4/static/datatype-json.html.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[3] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[4] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In *WebDB*, 2013. Extented version: http://pages.cs.wisc.edu/∼chasseur/argo-long.pdf.

[5] B. Choi. What are real dtds like? In *WebDB*, pages 43–48, 2002.

[6] A. Eisenberg and J. Melton. Advancements in SQL/XML. *SIGMOD Record*, 33(3):79–86, 2004.

[7] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[8] Z. H. Liu, B. Hammerschmidt, and D. McMahon. Json data management: Supporting schema-less development in rdbms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[9] Z. H. Liu, B. C. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang. Closing the functional and performance gap between SQL and nosql. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 227–238, 2016.

[10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[11] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: A sql system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 815–826.

[12] Z. Wang and S. Chen. Exploiting common patterns for tree-structured data. In *ACM SIGMOD international conference on Management of data*, 2017 (to appear).