Exploiting Common Patterns for Tree-Structured Data

Zhiyi Wang Shimin Chen*
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
{wangzhiyi,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 paper, we aim to better understand tree-structured data types in real uses and optimize for the common patterns. We present an in-depth study of five types of real-world use cases of tree-structured data. We find that a majority of the root-to-leaf paths in the tree structures are simple, containing up to one repeated node. Given this insight, we design and implement Steed, a native analytical database system for tree-structured data. Steed implements the baseline general-purpose support for storing and querying data in both row and column layouts. Then we enhance the baseline design with a set of optimizations to simplify and improve the processing of simple paths. Experimental evaluation shows that our optimization improves the baseline by a factor of up to 1.74x. Compared to three representative state-of-the-art systems (i.e. PostgreSQL, MongoDB, and Hive+Parquet), Steed achieves orders of magnitude better performance in both cold cache and hot cache scenarios.

1. INTRODUCTION

Tree-structured data formats, such as JSON [11] and Protocol Buffers [17], have numerous applications in a wide variety of real scenarios, including social network data feeds [19], online data services [20, 10, 18], communication protocols [1, 2], publicly available data sets [8, 7], and sensor data [4, 9, 5]. 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 types (e.g., C struct, C++ class, Java class) used in high-level programming languages can be easily represented. Such expressing power is one of the main reasons for the popularity of tree-structured data formats. However, the sophisticated data types present a significant challenge for systems supporting tree-structured data. In this paper, we aim to better understand tree-structured data types in real uses and optimize algorithms and data structures for processing tree-structured data.

1.1 Tree-Structured Data Model

In general, a tree-structured data model[1] can be recursively defined as follows:

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

A value type is defined as an object, an array, or a primitive type. An object contains a list of key-value pairs, while an array consists of a list of values. The key-value pairs in an object may be missing or present in specific data records. A primitive type is an atomic data type. The top-level type is an object. Depending on the actual tree-structured data formats, an object is also called a message, a document, or a record, and an array type is also known as a repeated type. Some formats (e.g., Protocol Buffers) require schema declarations for data, while others (e.g., JSON) are schema-less, where the schema information is implied by the data.

A tree-structured data type can be viewed as a schema tree. The top level type is the root of the tree. 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 3.)

The tree-structured data model is flexible enough to express sophisticated data types used in high-level programming languages, such as C/C++ and Java. An object can easily represent a struct, a class, or a map. An array is capable of expressing an array or a list. In fact, one can describe arbitrarily sophisticated data types with deep nesting levels and a lot of repeated sub-types. However, such complexity presents a significant challenge in efficiently supporting tree-structured data formats.

1.2 Existing Support for Sophisticated Types

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

[1]We follow previous work [22] to call light-weight nested data formats, such as JSON and Protocol Buffers, tree-structured data formats.

*Corresponding author

SIGMOD'17, May 14-19, 2017, Raleigh, NC, USA

DOI: http://dx.doi.org/10.1145/3035918.3035956

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 [25]. 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 [31]. This is the approach taken in the SQL/JSON standard effort and by Oracle [31]. 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 [16]. 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 [36].

Designing Native Systems. NoSQL document stores (e.g., MongoDB [15] and CouchDB [6]) 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. [33] 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 [3] is an open-source Java-based implementation of Dremel's columnar design. It can be integrated into the Hadoop ecosystem (e.g., Hive [37]) as an input/output format. Furthermore, AsterixDB [23] 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.

1.3 Our Solution: Steed

We take a different approach in this paper. We perform 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. While the number of leaf nodes in the tree structures varies greatly, from less than 10 to a few hundred, we find interesting common patterns across the cases. The heights of the trees are not very large. The highest tree consists of 8 levels. Trees of 2–3 levels are popular. More importantly, a majority of the root-to-leaf paths are quite simple, containing no repeated node or only one repeated node. We call such paths *simple* paths. Therefore, we would like to optimize for the common patterns.

We have designed and implemented an analytical database system, called *Steed* (**S**ystem for **tree**-structured **d**ata), that provides native support for tree-structured data. The baseline Steed design supports general-purpose data storage and query processing of tree-structured data in both row and column formats. Then, we optimize the column storage, the column assembling process, and in-memory data layouts for the frequent structure patterns observed in our use case study. For the column storage, we propose a simplified encoding scheme. For the process to assemble columns, we propose a flat assemble algorithm that avoids the cost of traversing state machines and tree structures. For the data layout in memory, we propose a flat data layout that reduces the overhead for accessing nested fields in query processing.

We perform an extensive experimental study to evaluate the performance of our proposed optimizations and compare the overall performance of Steed with state-of-the-art systems. Experimental results show that our optimizations can improve the performance of SQL-like queries by a factor of up to 1.74x compared to the baseline Steed implementation. We compare Steed with three state-of-the-art systems: an RDBMS with JSON extension (PostgreSQL), a native system with a binary row layout (MongoDB), and a native system that stores data in a binary column format (Hive+Parquet). In cold cache experiments where data are read from disks, the best Steed design achieves 4.1–17.8x speedups over Hive+Parquet, 55.9–105.2x improvements over MongoDB, and 33.8–1294x improvements over PostgreSQL. When data fits into memory, the best Steed design achieves 11.9–22.6x speedups over MongoDB, 19.5–59.3x improvements over Hive+Parquet, and 16.9–392x speedups over PostgreSQL.

Related Work on XML. There is a large body of work in the literature on storing and querying XML documents [21, 28]. Column storage of tree-structured data has already been explored for XML. In particular, MonetDB/XQuery is an XML database system implemented on top of a columnar relational database system, MonetDB [24]. The properties of real-world DTDs (i.e. XML schemas) have been studied [26]. However, XML documents are often large with many entities. A single XML document often contains a great many repeated tags. Moreover, DTDs can allow recursions and cycles [26]. In contrast, records in JSON-like formats are often much lighter weight. The size of a JSON-like record is often comparable to that of a record in a relational table. As a result, processing structures inside a single document often plays a significant role in XML, while we focus on the case with a large number of relatively small JSON-like records. As shown in our survey of real-world data patterns, JSON-like tree-structured records often have simple structures, which we exploit in the design of Steed.

1.4 Contributions

The contributions of this paper are as follows. First, we present the first in-depth study to understand real-world data patterns for tree-structured data formats (Section 2). Using five types of representative use cases, we find that simple paths dominate the tree structures. Second, we propose a set of data structures and algorithms to optimize for the common patterns (Section 4). Third, we describe the design and implementation of Steed, a native analytical database system for tree-structured data (Section 3 and 4). Finally, we perform an extensive experimental study to evaluate the proposed techniques and compare the overall performance of Steed with three representative state-of-the-art systems (Section 5).

2. ANALYZING TREE-STRUCTURED DATA IN THE REAL WORLD

In this section, we collect and analyze representative tree-structured data in the real world. We would like to understand: How are tree-structured data structures used in the real world? How complex are the structures in use? Are there any common structure patterns? Such understanding can guide our design choice for optimizing the efficiency of tree-structured data processing.

We have observed the following interesting facts in the study:

- Tree-structured data are used widely in practice.

Table 1: Root-to-leaf path analysis for Tweets.

Leaf level	No Repeated	1 Repeated	≥2 Repeated	Total
Level 1	16	0	0	16
Level 2	61	2	0	63
Level 3	51	21	4	76
Level 4	1	19	4	24
Level 5	0	12	0	12
Level 6	0	12	0	12
Total	129	66	8	203

- The number of leaf nodes in the tree structures varies greatly, from less than 10 to a few hundred.
- The heights of the trees are not very large. The highest tree consists of 8 levels. Trees of 2–3 levels are popular.
- A large fraction of the leaf nodes are quite simple. Their paths to the tree root contain at most one repeated field. We call a path that contains at most one repeated field a *simple* path.

In the following, Section 2.1 overviews the representative use cases in the study. Then, Section 2.2–2.6 each studies a specific type of real world use case of tree-structured data.

2.1 Representative Use Cases

We analyze five types of real-world use cases that employ tree-structured data, such as JSON and Protocol Buffers. First, data feeds in social network services, such as Twitter [19], contain data in JSON. They are used as an important type of raw data in many data analysis applications. Second, online data services (e.g., with RESTful APIs [12]), such as Yahoo web service [20], often provide query answers in JSON. Third, a large number of popular open source distributed computing platforms, including Hadoop [1] and HBase [2], use Protocol Buffers to implement their communication protocols. Fourth, a number of publicly available data sets, including semantic web data sets [8], can be obtained in JSON. Finally, the latest sensor platforms (e.g., Arduino [4], DragonBoard [9]) provide the capability of generating sensor data in JSON.

2.2 Data Feeds

The tweets in the Twitter data feed [19] are in JSON. Figure 15 depicts the schema tree structure of tweets. Since the full figure is too large to be clearly presented, we cut the tree into two. The cutting point is the node highlighted with the blue color. The green node is the root node. We use red to denote nodes that are repeated (i.e. arrays in JSON).

Analyzing the tree structure, we see that there are 238 nodes in the tree, among which 203 nodes are leaf nodes and 35 nodes are non-leaf nodes. The deepest leaf node appears at the 6th level of the tree (root is at level 0). Among the 238 nodes, 22 nodes (or less than 10%) are repeated nodes, i.e. arrays, which may contain multiple values in instances of tweets.

Table 1 lists the statistics of root-to-leaf paths for the tree structure. If we consider every distinct path from the root to a leaf node, then 129 paths of the total 203 paths contain no repeated nodes. 66 paths contain a single repeated node. Only 8 paths contain more than one repeated nodes. We consider paths with no repeated nodes or with one repeated node as simple paths. Overall, more than 96% of the root-to-leaf paths are simple.

Note that the tree structure of tweets is the most complex among the real-world data sets that we see. However, even in this case, there are common simple patterns that can be exploited in query processing. Our experiments in Section 5 will use Twitter data.

2.3 Online Data Services

We study three representative online data services: Yahoo [20], IMDB [10], and Sina Weibo [18]. JSON is often the preferred format of query results in online services that conform to the REST architecture. Many (mobile) applications are built on top of online data services to provide useful information and services to end users. For example, Sina Weibo provides a set of online service APIs that can be used by mobile apps to integrate the capability of accessing and managing (tweet-like) Weibo contents.

Table 2: Analysis of representative online data services.

Online service	Yahoo Geo	Yahoo Finance	IMDB Movie Star	Sina Weibo avg (max)
#Levels	2	5	2	2.2 (4)
#Nodes	11	25	6	29.9 (111)
#Leaves	8	19	4	27.5 (105)
No Repeated	0%	16%	0%	67%
1 Repeated	100%	84%	100%	32%
≥2 Repeated	0%	0%	0%	1%

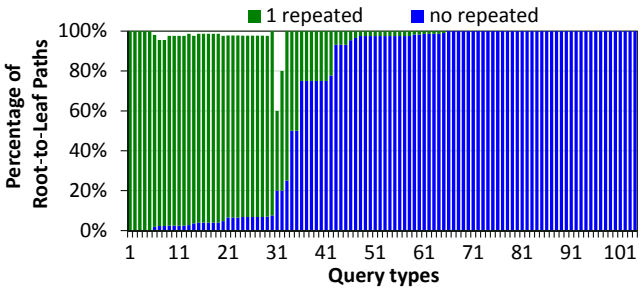


Figure 1: Analysis of root-to-leaf paths of JSON results for 104 different Weibo query types.

Table 2 shows the statistics of two types of Yahoo services, one type of IMDB service, and 104 types of Weibo services. For each type, we run example queries as specified by the respective tutorials or manuals, then obtain and analyze the returned JSON results. We spend most of our effort in understanding the query results of the 104 types of Weibo services.

From the table, we see that the trees contain 2–5 levels (excluding the root). The most complex trees contain 111 nodes and 105 leaf nodes. Among all the root-to-leaf paths, 100% are simple paths in Yahoo and IMDB query result trees, and an average of 99% of the paths are simple in the query results trees of Weibo.

Figure 1 further shows the breakdown of the root-to-leaf paths for all the 104 types of query results in Weibo. The results are sorted from left to right in ascending order of percentage of paths without any repeated nodes. As seen from the figure, simple paths dominate the majority of query results. Interestingly, in a large fraction of cases, almost all paths contain no repeated nodes, while there are other cases where almost all paths contain 1 repeated nodes. Therefore, both of these patterns are important.

2.4 Communication Protocols

Protocol Buffers simplifies the implementation of communication protocols and is employed in a number of well-known open source platforms, include Apache Hadoop [1] and Apache HBase [2]. Protocol buffers data can be recorded and analyzed for understanding system behaviors, detecting anomalies, and debugging.

We take Hadoop as an example, and analyze the tree structures of its communication protocols. In Hadoop 2.6.0, there are 449 types of Protocol Buffers messages that are not empty[2].

We construct syntax trees for the protocol buffers messages. On average, there are 14.1 nodes, including 9.5 leaf nodes, in the trees. The most complex tree contains 146 nodes, of which 89 are leaf nodes. The average height of the trees is 2.4 levels. The highest tree, which is also the highest seen in all studied cases, has 8 levels.

[2]There are some messages that do not contain any internal elements.

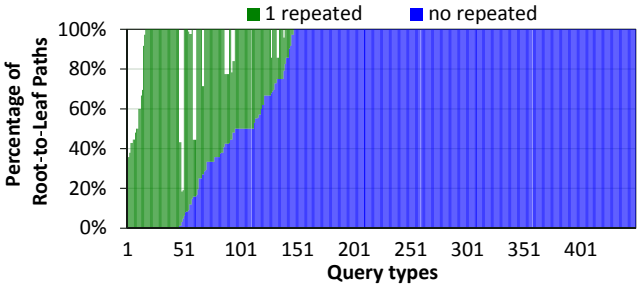


Figure 2: Analysis of root-to-leaf paths of Protocol Buffers messages for Apache Hadoop.

Figure 2 considers the breakdown of root-to-leaf paths for the trees. The trees are sorted in ascending order of the percentage of paths that contain no repeated nodes. From the figure, we see that simple paths, i.e. paths with up to 1 repeated nodes, dominate almost all cases. On average, 78% of the root-to-leaf paths in a tree contain no repeated nodes, 19% contain 1 repeated nodes, and only 3% in a tree contain 2 or more repeated nodes. In 416 trees out of the 449 trees, the percentage of simple paths is greater than 90%.

2.5 Publicly Available Data Sets

Many publicly available online data sets provide JSON as one of their data download formats. For example, DBpedia [8] extracts structured data from Wikipedia. It is part of the decentralized linked data effort [13]. The DBpedia dataset contains about 1 billion triples, describing over 3 million concepts in 11 different languages. The full DBpedia data set and portions of the data set can be downloaded in JSON[3]. Another representative example is data.gov [7], which provides public access to open datasets from the US federal government. In July 2016, there are over 180 thousand datasets available on data.gov, many of which can be downloaded in JSON.

We have downloaded a number of data sets from DBpedia and data.gov. Interestingly, we find that a whole downloaded data set is a single JSON document, as shown below. It consists of two large key-value elements. The first element (i.e. `properties` in the case of DBpedia and `meta` in the case of data.gov) contains an array of metadata descriptions, while the second element (i.e. `instances` in the case of DBpedia and `data` in the case of data.gov) contains an array of the actual data records.

DBpedia:

```
{"properties": [{...}, {...}, ...],
 "instances": [{...}, {...}, ...]
}
```

data.gov:

```
{"meta": [{...}, {...}, ...],
 "data": [{...}, {...}, ...]
}
```

Note that this organization cannot be efficiently processed by most of existing systems with JSON support. This is because existing systems expect a data set to contain (a large number of) records of relatively small sizes rather than a single large record that encapsulates all the data. However, it is easy to develop a tool to preprocess the downloaded data set into a set of metadata records and a set of data records. In this way, both the metadata and the data meet the systems' expectation.

Then, we analyze the tree structures of the metadata and the data. In the case of DBpedia, the tree structure of the metadata consists of a single level with four leaf nodes, `propertyType`, `propertyTypeLabel`, `propertyLabel`, and `propertyURI`. Every key (i.e. DBpedia property) that appears in the data is described by a metadata record. In the schema tree of the data, the root node has a single child node (level 1). The level-1 node has many child nodes (level 2), each of which is a property described in the metadata. A property node may either be a leaf node or contain an array of leaf nodes. (The two varieties appear 78% and 22% in the level-2 nodes of the Game data set, respectively.) Therefore, there are 3 levels in the tree (excluding the root level). Since the root-to-leaf paths contain at most 1 repeated node, the paths are all simple.

In the case of data.gov, the structure of a metadata record is more complex than that of DBpedia's metadata record. For example, in the data set titled "Infant and neonatal mortality rates: United States, 1915-2013", the structure of the metadata record is a tree of 5 levels. There are 75 root-to-leaf paths, among which 46 have no repeated nodes, 21 have 1 repeated nodes, and 8 have 2 or more repeated nodes. Therefore, 89%of the root-to-leaf paths are simple. On the other hand, a data record of data.gov is simpler than that of DBpedia. The schema tree is a flat single-level tree.

In summary, the majority of root-to-leaf paths are simple in the tree structures of both DBpedia and data.gov.

2.6 Sensor Data

The latest sensor platforms (e.g., Arduino [4], DragonBoard [9], BeagleBone [5]) provide the capability of generating and processing data in JSON. Since IoT (Internet of Things) is expected to be one of the largest raw data sources, it is important to handle sensor data well.

The following shows two examples of JSON outputs from a temperature sensor and a sensor that reports WaveLan signal strength:

```
{"arduino": [
   {"location":"indoor","celsius":22.77},
   {"location":"outdoor","celsius":15.55}
]}
```

```
{"pk": 52728371, "model": "wifimon.wifimondata",
 "fields": {
   "typeid": 7, "srcmac": "fc:4d:d4:d1:e2:67",
   "timestamp": "2014-08-31T23:00:01Z",
   "interval": 1, "dstmac": "ff:ff:ff:ff:ff:ff",
   "timestamp_ms": 45163, "frequency": 1,
   "deviceid": 33, "rssi": -72
}}
```

We see that all root-to-leaf paths in the two structures are simple, either containing no repeated nodes or containing at most 1 repeated nodes. A sensor typically manages only a small number of measurements, and has limited power and processing capability. As a result, JSON structures similar to the above would be common. Therefore, we expect sensor JSON outputs to contain simple structures, where most root-to-leaf paths are simple.

3. STEED DESIGN

We have designed and implemented an analytical database system, called *Steed* (**S**ystem for **tree**-structured **d**ata), that provides native support for tree-structured data. We describe our baseline design in this section and propose a set of optimization techniques for the common patterns in the next section.

3.1 Steed Architecture

Figure 3 shows the architecture of Steed. Presently, it consists of mainly three parts: (i) the data parser, (ii) the data storage, and (iii) the query engine. Snippets of two tweet records, shown in Figure 3, will be used as our running example. For simplicity, we have preserved only a subset of the attributes sufficient for the description

[3]For example, the download link for Thing/Activity/Game is http://web.informatik.uni-mannheim.de/DBpediaAsTables/json/Game.json.gz.

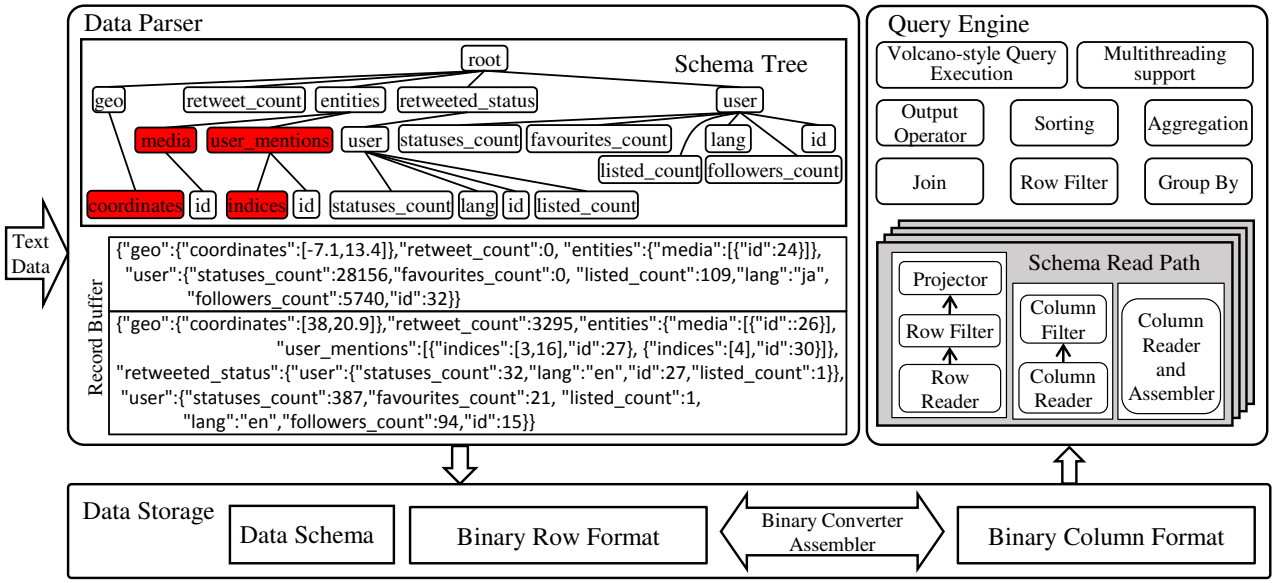


Figure 3: Architecture of Steed. (Two JSON records are used as the running example. The highlighted red nodes are repeated.)

in this section. We randomly modify the id fields to avoid leaking user information.

3.2 Data Parser

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. Figure 3 shows the schema tree for the example tweet records[4]. There are three levels (excluding the root) in the tree. We use the dot notation to express a full path, e.g., `retweeted_status.user.lang`. The highlighted red nodes correspond to repeated nodes (i.e. arrays). Every node in the schema tree is assigned a unique ID, and contains information about the corresponding type (e.g., object, array, or primitive types) and its children. Note that attributes that have the same path but different types are considered as different attributes and created as different nodes in the tree[5]. The schema tree is stored as a file in the underlying storage.

3.3 Data Storage and Data Layouts

Steed stores binary data and schema as files in the underlying file system. Steed supports both a binary row format and a binary column format. Presently, the choice of the storage formats is determined manually by setting an input parameter of the data parser.

The row data layout is shown in Figure 4. An object instance consists of a size field for the object's total size in bytes, a number field recording the number of children, a list of IDs and value offsets of child node instances that are present, and a list of values. Note that the offset fields are used to support variable sized values. An array instance has a similar structure except that node IDs are

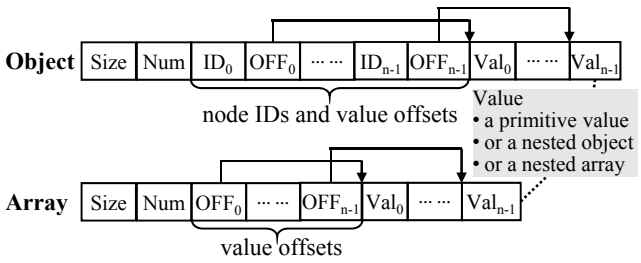


Figure 4: Row data layout.

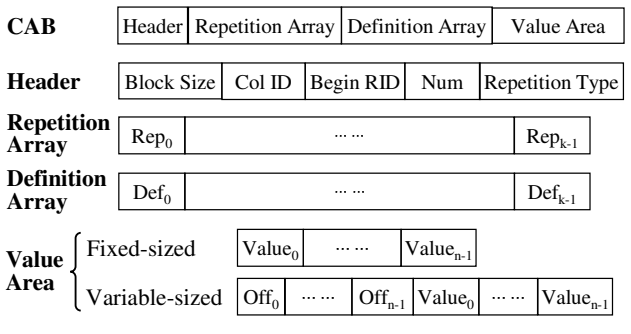
omitted because all children have the same type. A value field can contain a primitive value or a nested value. A nested value contains an object instance or an array instance in the value field. MongoDB provides a binary JSON layout called BSON. However, BSON encodes detailed type information in each record. Steed stores only node IDs in each record, which reference the type information in the schema tree, thereby avoiding repeatedly storing the type information, and reducing space and I/O costs for row data layout.

The column data layout is illustrated in Figure 5. Every leaf node in the schema tree is stored as a column data file. To support efficient I/O, the files are divided into multiple CABs (Column Aligned Blocks). The CABs across different columns are aligned at the same record boundaries to simplify the column assembling procedure. As shown in Figure 5, our baseline scheme is based on Dremel [33]. In general, a root-to-leaf path may contain multiple repeated nodes. Therefore, it is necessary to distinguish at what level the repetition occurs and whether a leaf node actually appears in a particular repeated branch. The two pieces of information are encoded as a repetition level (rep) and a definition level (def), respectively. rep shows at what level the repetition occurs, while def shows how many nodes on the path appear. The CAB stores a repetition array, a definition array, and a value area. Note that the number of rep and def entries can be larger than the number of values. This is because rep and def are also used to encode missing instances as evidenced by the following example.

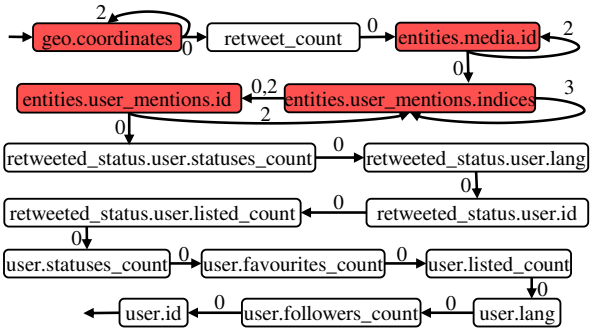
For example, consider `entities.user_mentions.id` in Figure 5(b). The first row is (0, 1, null). $rep = 0$ (i.e. the repetition occurs at the root) means that a new record starts. Since the first record does not contain `entities.user_mentions.id`, the

[4]Note that there is a data attribute "indices". This should not be confused with an index structure in the system. Presently, we have not yet implemented index support, but it is on our to-do list.

[5]We assume that elements in an array are of the same type. If this is not the case, then the array can actually be regarded as a simplified object, where the (implicit) keys are array indices, "0", "1", "2", etc.



(a) Column Aligned Block (CAB) layout



(c) Finite state machine for assembling columns

geo. coordinates

rep	def	value
0	2	-7.1
2	2	13.4
0	2	38
2	2	20.9

retweet_count

rep	def	value
0	1	0
0	1	3295

entities.media.id

rep	def	value
0	3	24
0	3	26

entities.user_mentions.indices

rep	def	value
0	1	null
0	3	3
3	3	16
2	3	4

entities.user_mentions.id

rep	def	value
0	1	null
0	3	27
2	3	30

retweeted_status.user.statuses_count

rep	def	value
0	0	null
0	3	17952

retweeted_status.user.lang

rep	def	value
0	0	null
0	3	en

retweeted_status.user.id

rep	def	value
0	0	null
0	3	27

retweeted_status.user.listed_count

rep	def	value
0	0	null
0	3	1

user.lang

rep	def	value
0	2	ja
0	2	en

user.followers_count

rep	def	value
0	2	5740
0	2	94

user.listed_count

rep	def	value
0	2	109
0	2	1

user.statuses_count

rep	def	value
0	2	28156
0	2	387

user.favourites_count

rep	def	value
0	2	0
0	2	21

user.id

rep	def	value
0	2	32
0	2	15

(b) Each leaf node is stored as a column file

Figure 5: Storing and assembling column data. (The baseline scheme is based on Dremel.)

value is null[6]. $def = 1$ means that only one node on the path appears (i.e. `entities`). The second row is (0, 3, 27), corresponding to the first occurrence in the second record. $def = 3$ means that all three nodes on the path appear. The third row is (2, 3, 30), which is the second occurrence in the second record. Here, $rep = 2$ because the repetition occurs at `entities.user_mentions`.

When reading multiple columns, Steed assembles columns into row data layout in memory. The baseline assembling algorithm constructs a finite state machine, as shown in Figure 5(c). The nodes in the FSM are the leaf nodes in the schema tree, and the numbers on the transition edges are rep. In essence, the FSM traverses the schema tree from left to right. For a repeated node, it may jump back and repeatedly visit the node depending on the rep and def. For a transition in the FSM, Steed moves up from the source leaf node in the Schema tree, then moves down to the destination leaf node. During such transitions, Steed sets the IDs, offsets, and primitive values in the row data layout.

3.4 Query Engine

Presently, Steed 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 [29], 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 reader reads a block of data at a time, and 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. The column reader reads a CAB at a time. If there is a filter predicate, the column filter applies it to the column values. Then the columns are assembled according to the description in Section 3.3. As a result, after the selection and the projection, the records are in the row layout (regardless of the 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 and data structures for the operators, including a hash-based join operator, a hash-based group-by operator, and a quick-sort based sorting operator.

Steed can exploit multiple cores to run a single query. Given a desired thread count t, Steed divides the underlying data files into t subsets of data blocks and assigns them to t threads for the selection and projection. Then, Steed implements multithreaded versions of the remaining operators.

3.5 Addressing Semantic Differences

We consider only relational operations on tree-structured records in our current implementation of Steed.[7] Even in this case, there exist significant semantic differences between tree-structured data and relational data. Afrati et al. has studied the semantic difference in filtering predicates [22]. The work shows that repeated nodes must be used with care otherwise predicates may have ambiguous meanings. Therefore, our goal in the design is to avoid ambiguity by defining default behaviors and sometimes introducing new keywords.

Our solution is summarized in Table 3. First of all, a leaf node with no repeated nodes on the root-to-leaf path represents an atomic primitive value (e.g., `user.lang`). Thus, it has the same semantics as a relational attribute, and therefore can appear in any clauses in a SQL select query.

Second, a leaf node with repeated nodes on the path corresponds to a list of values in a record. To avoid ambiguity, we convert the list into a single value depending on where it is used, as shown in Ta-

[6] null values can be identified by rep and def and are not stored.

[7] It would be interesting to investigate operations that are specific to tree-structured records, which may not be relational. However, this is beyond the scope of this paper.

Table 3: Addressing semantic differences.

	Leaf node (not repeated)	Leaf node (repeated)	Nonleaf node
Filtering predicate	Correct semantics	Any, All	A string representation of the subtree rooted at the node
Group-by key		Concatenation	
Aggregation value		All	
Order-by key		Concatenation	
Join key		Concatenation	

ble 3. For filtering predicates, we introduce two keywords: *any* and *all*[8]. One of the keywords must be specified. For the predicate to be evaluated to be true, *any* requires at least one of the value in the list to satisfy the predicate, while *all* requires all values in the list to satisfy the predicate. For example, `any:entities.media.id < 25` is true, but `all:entities.media.id < 25` is false in the running example. In the case of group-by keys, order-by keys, and join keys, Steed will concatenate all the values in the list and use the concatenated string as the key. If the list is used as an aggregation value, Steed assumes the all behavior, i.e. all the values in the list are aggregated.

Third, if a nonleaf node is used, then we assume that it represents its subtree as a whole (e.g., `retweeted_status.user`). Steed converts the nonleaf node to a string representation of its subtree. This string is then used as an atomic relational attribute in the query.

Finally, we use the SQL null semantics to deal with values that are missing in a record.

4. OPTIMIZING FOR SIMPLE PATHS

While the baseline design supports generic tree-structured data that may have arbitrarily complex structures, our study of the real-world tree-structured data in Section 2 shows that a majority of the root-to-leaf paths are simple. In this section, we optimize the processing for simple paths. In particular, we can leverage the knowledge of the simple path to simplify the CAB layout, accelerate the column assembling process, and design more compact and more efficient in-memory record structures.

4.1 Optimizing Column Storage

Consider the example in Figure 5(b). Clearly, if there are no repeated nodes on the root-to-leaf path (the cases without highlighting), then rep will always be 0. Therefore, we can omit the repetition array entirely. If there is only a single repeated node on the root-to-leaf path, then repetition can occur either at this node or at the root node (to start a new record). Thus, rep has at most two values. In this case, we can use a single bit for each rep. In both cases, we can reduce def to a single bit to specify whether the leaf node value is missing or not.

Our optimization reduces the size of rep and/or def for simple paths. Suppose there are L levels in the tree. Then a rep and a def can take up to $log(L)$ bits. For simple paths without repeated nodes, we remove the rep and use a single bit for def. Therefore, for each record, we save up to $2log(L) - 1$ bits. For simple paths with a single repeated node, we use a single bit for both rep and def. Therefore, for each occurrence of the record and the repeated branch, the saving is up to $2log(L) - 2$ bits. Suppose the path is repeated an average r times in a record. For each record, the saving is up to $r(2log(L) - 2)$ bits.

In the real world use case study, L is up to 8. In this case, the savings are up to 5 bits and $4r$ bits per column per record for simple paths without repeated nodes and simple paths with a single repeated node, respectively. Depending on the value size and the path length, this may lead to significant savings. For example, if the value is a 2-byte string representing languages and the path has 6 levels, then the space saving can be up to 20%.

[8]Argo [25] introduces the *any* keyword to deal with JSON arrays. In addition to *any*, we find that *all* is also meaningful for handling repeated nodes in predicates.

Algorithm 1 Flat Assemble Algorithm

```
Require: All column readers in rds are sorted by their leaf id
1: Procedure FlatAssemble (ColumnReader [] rds)
2: flt_recd = create new flat record
3: for all rd in rds do
4:     while rd does not reach record boundary do
5:         read column item and append it to flt_recd
6:     end while
7: end for
8: fill header of flt_recd
9: End Procedure
```

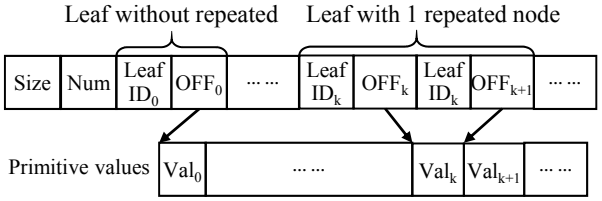
4.2 Optimizing Column Assembling Process

In the baseline assembling algorithm, the state transformation closely follows the tree structure. Every non-leaf node instance in the tree is visited twice, i.e. for initializing fields (e.g., Num, IDs, Offsets) when entering the subtree rooted at the node, and for finalizing the field values when leaving the subtree. Every leaf node instance in the tree is visited once. A node visit is costly; it visits the state transformation array then the schema tree, potentially requiring interpretation of repetition values and definition values.

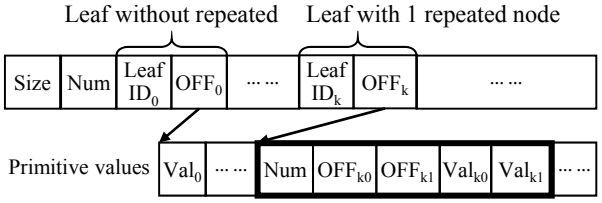
If all or most of the paths in a query are simple, we can avoid the complex baseline algorithm that visits a state machine to traverse the schema tree. We propose an optimization, called a flat assemble algorithm, as shown in Algorithm 1. Basically, it reads the value of the simple columns in a record one by one to construct a flat row data layout for the simple paths. The flat data layout will be discussed in detail in the next subsection. The non-simple nodes in the query are processed with the baseline algorithm. As a result, Algorithm 1 directly visits the leaf nodes without incurring the cost of visiting non-leaf nodes as in the baseline algorithm.

We consider a node visit as a basic operation, and analyze the cost savings. Suppose k simple columns C_1, ..., C_k are to be assembled. The leaf node of C_i appears at level L_i. C_i may share common ancestors with C_1, ..., C_{i-1}. Suppose the node at level LD_i is the first ancestor that does not appear in the previous paths. If there is a repeated node, suppose the repeated node is at level LR_i, and C_i repeats an average r_i times per record. We consider the following three cases:

- *Case 1: C_i does not contain repeated nodes.* Thus, the baseline algorithm will visit $L_i - LD_i$ non-leaf nodes that have not been covered by the computation for C_1, ..., C_{i-1}. Since every non-leaf node is visited twice, the number of nonleaf node visits is given by: $N_1 = 2(L_i - LD_i)$.
- *Case 2: C_i contains a repeated node and $LR_i \geq LD_i$.* In this case, the path fragment from level LD_i to $LR_i - 1$ is not repeated, while the path fragment from level LR_i to L_i is repeated r_i times. Therefore, the number of nonleaf node visits is given by:
$$N_2 = 2(LR_i - LD_i) + 2r_i(L_i - LR_i)$$
- *Case 3: C_i contains a repeated node and $LR_i < LD_i$.* In this case, the repeated node is shared between C_i and some previous nodes. We only need to consider Level LD_i to L_i, which occur r_i times. Therefore, the number of nonleaf node visits is given by: $N_3 = 2r_i(L_i - LD_i)$.



(a) One level of values



(b) Repeated values are nested

Figure 6: Flat data layout in memory.

Let $r_i = 1$ and $LR_i = L_i$ for case 1. Then we can combine the three cases to compute the total number of node visits:

$$\sum_{i=1}^{k}[2 \cdot max(LR_i - LD_i, 0) + 2r_i(L_i - max(LR_i, LD_i))]$$

Since the total number of leaf node visits is given by $\sum_{i=1}^{k} r_i$, the relative improvement depends on the leaf node levels and the tree structures. For example, if two leaf nodes at level 5 with non-repeated simple paths are assembled, then the number of leaf node visits is 2, and the number of non-leaf visits is up to 8. Therefore, this optimization saves 80% of the node visits in this case.

4.3 Optimizing In-memory Structure

The baseline design employs the generic row data layout (as shown in Figure 4) in memory. The number of nesting levels in the data layout is greater than or equal to the number of nonleaf nodes in the root-to-leaf paths. A nonleaf node corresponds to an object, while a repeated (nonleaf or leaf) node corresponds to an array. At every level, there is an array of Offsets and/or IDs. The IDs are sorted to allow binary searches. However, this design incurs significant overhead. it is necessary to perform a binary search at every level and follow offset pointers in a root-to-leaf path in order to retrieve a value. For example, the value of `retweeted_status.user.lang` is stored on the 3rd level in the generic layout. Thus, one has to perform three binary searches and follow offset pointers to move to `retweeted_status` at level 1, `user` at level 2, then `lang` at level 3 to retrieve the value.

We aim to avoid the above overhead. Figure 6 depicts two alternative flat data layouts. In the first design, as shown in Figure 6(a), the flat data layout contains only a single level of IDs, Offsets, and values. The values in the figure are primitive values. Leaf node IDs are sorted to allow efficient binary search. If a root-to-leaf path does not contain any repeated nodes, then there is a single instance of (ID, Offset, and value). If a root-to-leaf path contains a repeated node, then there can be multiple instances of (ID, Offset, and value)s. The ID is the same, while each pair of Offset and value corresponds to an instance of the leaf node in the record. Note that this design keeps the structure simple to access. The drawback is that the ID field is potentially repeated multiple times.

The second design employs a nested structure for the repeated values, as shown in Figure 6(b). Paths with no repeated nodes are stored in the same way as the first design. For a path with a repeated node, there is only a single entry in the top-level (ID, Offset) array. The multiple values and their offsets are stored contiguously in the value area. This design saves memory space because it does not store the ID field multiple times. However, it may incur more cost for going through one more nesting level. We choose this design only when the average number of the repeated values per record is above a pre-defined threshold. The average number of repeated values can be estimated by using the number of values and the number of records as recorded in the CAB header.

When a query involves both simple paths and non-simple paths, we use the flat layout and the generic layout for the simple paths and the non-simple paths, respectively. Then, we store a pointer in the flat layout to point to the generic layout of the same record.

The in-memory layout is accessed by the join, group-by, aggregation, and sorting operators. Therefore, the in-memory layout may significantly impact the performance of these operators. The actual savings depend on the number of levels of the root-to-leaf paths. The larger the number of the levels, the more savings that the flat data layout brings. Suppose a simple path without repeated nodes contains 3 nonleaf nodes and a leaf node. Therefore, the original generic row layout has 4 nesting levels. The flat data layout removes 3 of the 4 levels, reducing 75% of nesting level accesses for retrieving a value.

5. PERFORMANCE EVALUATION

We evaluate the performance of our proposed optimizations in this section. We would like to understand the following questions:

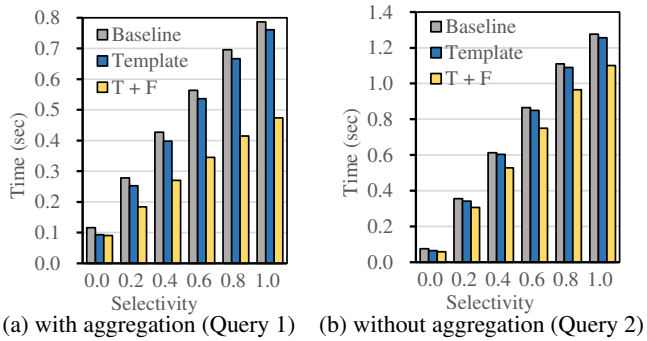
- How effective are our proposed optimizations for simple paths?
- What is the impact of the variation in path lengths and types on the proposed optimization?
- What is the relative performance of Steed compared to state-of-the-art systems (e.g., PostgreSQL, MongoDB, and Parquet)?

5.1 Experimental Setup

Machine Configuration. The experiments are performed on a Lenovo ThinkCentre M8500t equipped with an Intel(R) Core(TM) i7-4770 @3.40GHz CPU (4 cores, 2 threads/core), 16GB DRAM, and a 7200rpm SATA hard drive. The machine runs Ubuntu 14.04 with a 3.13.0-24-generic Linux Kernel. We have implemented Steed in C/C++, and compiled it using gcc version 4.8.2 with optimization level -O3.

Measurement Methodology. In most settings, we are interested in understanding the benefits of the optimizations in memory. Note that as main memory capacity increases exponentially, the (hot) data set of many important applications can be processed in main memory, as evidenced by the recent popularity of main-memory database engines in mainstream database systems (e.g., Microsoft Hekaton [27], IBM DB2 [35], Oracle [30], SAP HANA [34]) and main-memory based big data processing systems (e.g., Spark [38], Pregel [32], Memcached [14]). In such cases, we warm up the OS file cache by running a query once before taking the measurements. We call such measurements *hot cache*. When comparing Steed to state-of-the-art systems, we measure *cold cache* performance in addition to hot cache performance. In cold cache experiments, we manually clear the OS file cache before each run to ensure the data are retrieved from the hard drive. In both cases, every reported result is the average of 5 runs.

Datasets. We mainly use three tweet datasets collected by running the Twitter API in 2012. The first dataset contains over 2.3 million JSON text records, and is about 5.6 GB large. The second dataset contains over 19.1 million records, and is over 45 GB large. The third dataset is a small data set, which is about 44 MB large. We use the first data set in the hot cache experiments and the second data set in the code cache experiments. The third data set is used only



(a) with aggregation (Query 1) (b) without aggregation (Query 2)

Figure 7: Query performance varying selectivity. (Leaf nodes are at level 2. All paths are simple without repeated nodes.)

in the join experiments. The characteristics of the tweet datasets are discussed in Section 2.2. A majority of the root-to-leaf paths in the schema tree are simple. If we consider the sizes of column data files, the files corresponding to the simple paths occupy about 96% of the total size. Therefore, simple path optimizations may lead to significant improvement.

Steed Variants. We evaluate the following three variants of Steed. (i) *baseline*: this is the baseline design as described in Section 3. (ii) *template*: we optimize the column file layout to reduce the ref and def sizes for simple paths, as described in Section 4.1. In our implementation, each different layout is called a template. (iii) *T+F*: we exploit the flat assemble algorithm and the flat data layout in addition to (ii), as described in Section 4.2-4.3. Here, T stands for template, and F stands for flat.

State-of-the-art Systems to Compare. We consider three state-of-the-art systems that are relevant to the processing of tree-structured data: an RDBMS with JSON extension (PostgreSQL), a native system with a binary row layout (MongoDB), and a native system that stores data in a binary column format (Hive+Parquet).

- *PostgreSQL*: PostgreSQL has been extended with JSON support recently. A JSON record is stored either as text (in the `json` type) or as binary data (in the `jsonb` type). We use jsonb in our experiments 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++. In January 2017, MongoDB was ranked the 4th most popular database[9]. MongoDB users include Adobe, Craigslist, eBay, LinkedIn, Foursquare, and so on. MongoDB stores JSON records in a binary format called BSON. We use MongoDB 3.2.8 in our experiments. We load tweet data sets using mongoimport, and use MongoDB's javascript interface to submit queries.
- *Apache Hive with Apache 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 our experiments, 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. To simplify the experimental setup, we extract a subset of the tweet attributes that are to be used in the experimental queries. This subset contains 9 attributes. 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. Note that this simplification is favorable to Parquet because a smaller amount of data is stored, which may lead to more efficient data accesses.

[9] http://db-engines.com/en/ranking

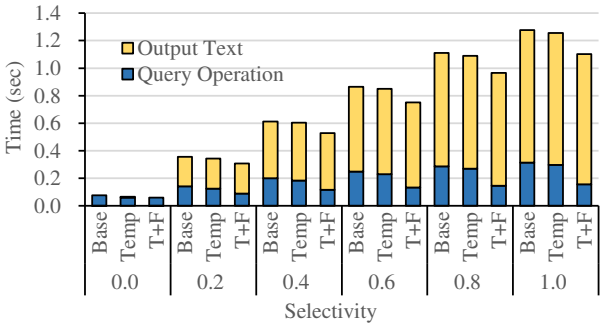


Figure 8: Breakdown of query execution times for Figure 7(b).

5.2 Query Performance Varying Selectivity

We compare the *baseline*, *template* and *T+F* schemes using the following two queries while varying selectivity:

- *Query with Aggregation (Query 1)*:

```
select max(user.followers_count)
from twitter
where user.favourites_count CMP_OP CONSTANT
group by user.lang
```

- *Query without Aggregation (Query 2)*:

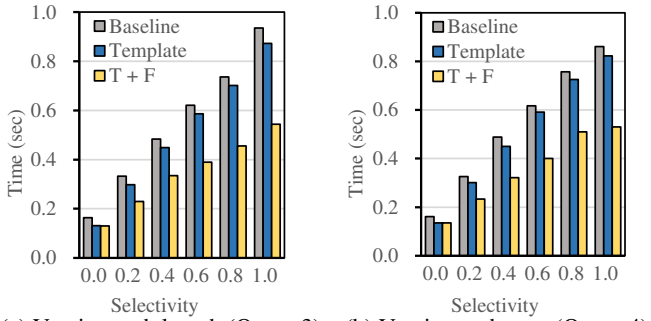
```
select user.followers_count
from twitter
where user.favourites_count CMP_OP CONSTANT
```

The queries perform selection and projection on the twitter data set. Note that the dot notations are used to specify attributes in tree-structured records. The where clause compares an attribute (i.e. `user.favourites_count`) against a constant threshold. We vary the constant threshold to select about 0%, 20%, 40%, 60%, 80%, and 100% of the total records. The two queries are the same except that the first query performs additional group-by and aggregation operations, which significantly reduce the amount of output. All the attributes in the queries are leaf nodes at level 2 in the schema tree. Their root-to-leaf paths are simple without repeated nodes. We will vary the leaf path levels and types in the Section 5.3.

Figure 7 shows the query execution times for the three Steed schemes. Overall, we see that (i) both optimized schemes achieve better performance than the baseline in all the experiments; (ii) *Template* slightly improves the baseline by a factor of 1.03–1.24x in the case with aggregation, and 1.02–1.17x in the case without aggregation; and (iii) *T+F* achieves significant improvements, obtaining 1.29–1.68x speedup in the case with aggregation, and 1.15–1.30x speedup in the case without aggregation.

The template optimization removes the space for rep and the cost of accessing rep for simple paths without repeated nodes. However, the leaf nodes in the queries are at level 2. Therefore, the original size of rep is log(2)=1 bit. As a result, the savings are quite low. On the other hand, the flat assemble algorithm significantly reduces the complexity in assembling columns, and the flat data layout improves data accesses in memory. As a result, *T+F* achieves significant better performance.

As more records are selected, the execution times of all schemes increase, as evidenced by the upward trends in the figure. The speedup of *T+F* over the baseline also increases. This is because the more records are selected, the more records need to be assembled and processed by the group-by and the aggregation operators. The benefits of the flat assemble algorithm and the flat data layout will be higher.



(a) Varying path length (Query 3) (level 3)

(b) Varying path type (Query 4) (path w/ 1 repeated node)

Figure 9: Adding a second predicate to Query 1.

Comparing Queries with Aggregation and without Aggregation. Comparing Figure 7(a) and (b), we see that the optimized schemes achieve significantly higher improvements in the case with aggregation. The main difference between the two cases is that the amount of output is drastically reduced when the group-by aggregation is used. As a result, the queries spend a much smaller fraction of their execution times producing text output.

To verify the above insight, in Figure 8, we break down the query execution time into two parts, the time for producing output and the remaining query computation time. We comment out the text output code and rerun experiments in Figure 7(b). This gives the time of the query operation. The output time is computed as the difference between the execution times with output and without output. From Figure 8, it is clear that the time to produce output is roughly the same across the three schemes. The output time occupies up to 86% of the total execution time (at 100% selectivity). Profiling shows that Steed spends a lot of time in the libc `snprintf` function. Therefore, the large (roughly fixed) output time is indeed the main cause for lower improvements of the optimized schemes. If we remove the output time and consider only the query operation, then *T+F* achieves 1.31–2.02x speedups over the baseline.

5.3 Varying Path Lengths and Types

In this section, we study how different root-to-leaf path lengths and types affect query performance.

Adding an Additional Predicate to Query 1. We add a second predicate to Query 1, and choose attributes of the predicate to introduce a different path length and a different path type, respectively.

```
select max(user.followers_count)
from twitter
where user.favourites_count CMP_OP CONSTANT
      and ATTR CMP_OP2 CONSTANT2
group by user.lang
```

- *Path length (Query 3)*: In Query 3, the `ATTR` is a level 3 leaf node, `retweeted_status.user.listed_count`.
- *Path type (Query 4)*: We choose `any:geo.coordinates` as `ATTR`. The attribute corresponds to a simple path with a single repeated node. Note that the keyword *any* is used. It means that the predicate is true if any instance of `geo.coordinates` satisfies the predicate.

Figure 9 shows the query performance for the two queries while varying selectivity. The two figures show similar trends as Figure 7(a). We see that (i) *Template* is slightly better than the baseline; and (ii) *T+F* achieves significant improvements, obtaining 1.21–1.68x speedup for Query 3, and 1.23–1.74x speedup for Query 4. The *T+F* scheme is effective when a different root-to-leaf path length or type is introduced.

Varying Both Path Lengths and Types. We use a variant of the above select-from-where query with three predicates. Then, we choose the attributes in the predicates to test various combinations of path lengths and types. We use *n*, *r*, and *m* to denote a root-to-leaf path with no repeated nodes, 1 repeated node, and 2 or more repeated nodes, respectively. Note that our optimization supports both *n* and *r*. We consider the following cases: 3n, 2n+1r, 1n+2r, 3r, 1m+2n, 1m+1n+1r, and 1m+2r. Most leaf nodes in the twitter schema tree appear at level 1 to 3. Thus, we consider level 1 to 3.

Figure 10 and Figure 11 show the performance of queries with various combinations of path lengths and types. In Figure 10, we vary path lengths for 3n and 2n+1r. All the predicate attributes are at the same level. In Figure 11, we fix the path length, then test different combinations of path types. Across all the configurations, we see that compared to the baseline, *Template* achieves a factor of 1.02–1.15 improvements, and *T+F* achieves a factor of 1.16–1.63 improvements. The *m* paths cannot be optimized. They are processed using the baseline algorithm. However, *T+F* can accelerate the processing of the simple paths when other *m* paths are present. We see that *T+F* achieves 1.16–1.31x speedups for configurations including *m* paths.

As the queries have different selectivity, it is difficult to compare the speedups of different configurations. However, it is clear that the speedups are similar to those achieved in Figure 7 and Figure 9. This shows the the effectiveness of the proposed scheme under various combinations of path lengths and types.

5.4 Random Query Performance

Next, we run queries generated randomly according to the following query template:

```
select $output_field
from twitter
where $pred1 and $pred2 and $pred3
group by user.lang
```

We use 7 patterns to generate random queries, as listed in Table 4. All the attributes in the output field and in the predicates are leaf nodes at level 3 in the schema tree. We divide the leaf nodes at level 3 into three disjoint subsets. The *n*-set, *r*-set, and *m*-set contain leaf nodes whose paths have no repeated nodes, 1 repeated node, and 2 or more repeated nodes, respectively. The output field is always randomly generated from the *n*-set. Each pattern in Table 4 specifies a combination of path types for the attributes in the three predicates. Given a pattern, the attributes in the predicates are randomly selected from the subsets of attributes as specified by the pattern. For example, for Pattern 2, two predicate attributes are randomly selected from the *n*-set, and one from the *r*-set.

Table 4: Random query patterns.

Path Type	n	r	m	Figure
Pattern 1	3	0	0	Patterns 1–4 are shown in Figure 12(a). In these cases, all paths are simple.
Pattern 2	2	1	0	
Pattern 3	1	2	0	
Pattern 4	0	3	0	
Pattern 5	2	0	1	Patterns 5–7 are shown in Figure 12(b). In these cases, one path contains 2 or more repeated nodes.
Pattern 6	1	1	1	
Pattern 7	0	2	1	

For every pattern, we randomly generate 1000 queries and measure their performance. Figure 12 shows the histograms on *T+F*'s speedups over the baseline. Patterns 1–4 are reported in Figure 12(a), and Patterns 5–7 are reported in Figure 12(b). The X-axis shows the speedup ranges, and the Y-axis shows the number of queries that fall in a particular histogram bucket.

From Figure 12(a), we see that *T+F* achieves 1.5–1.7x speedups over the baseline for a large number of random queries where all paths are simple. On the other hand, when there are mixed simple

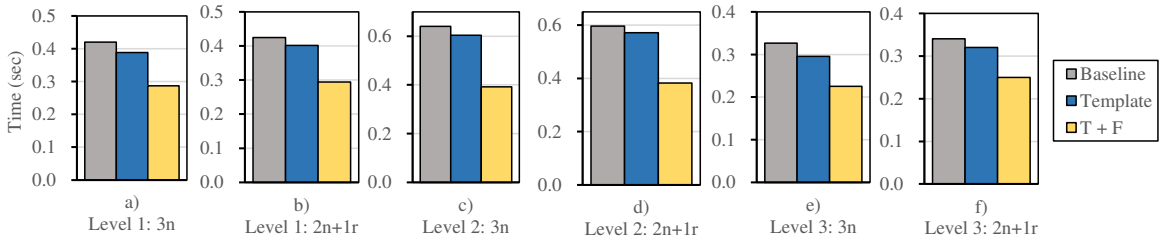


Figure 10: Varying path length for 3n and 2n+1r.

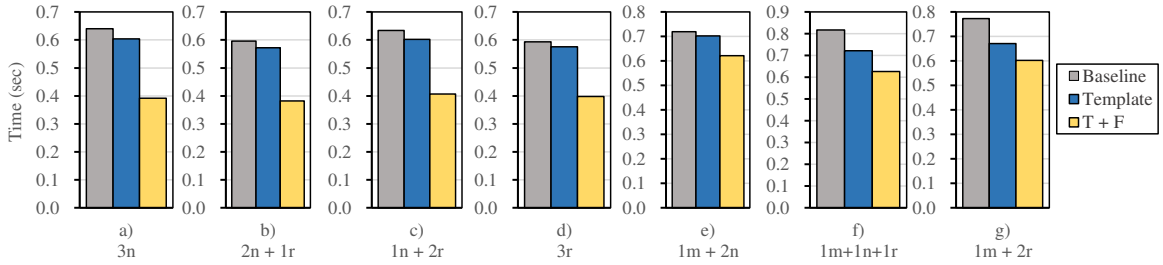
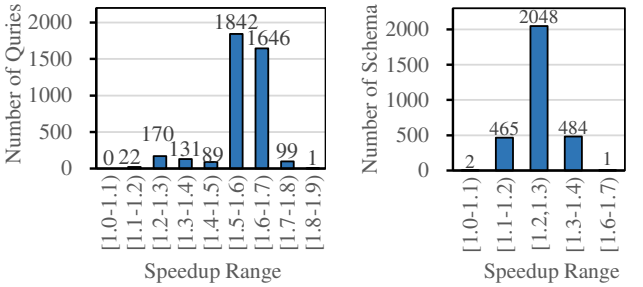
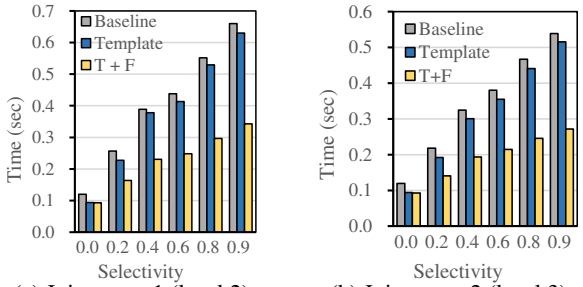


Figure 11: Varying path types for level 2. (n: no repeated nodes, r: 1 repeated node, m: 2 or more repeated nodes.)



(a) All simple (Patterns 1–4) (b) Some simple (Patterns 5–7)

Figure 12: Speedups of T+F for random queries.



(a) Join query 1 (level 2) (b) Join query 2 (level 3)

Figure 13: Join performance varying selectivity.

and non-simple paths, the speedups achieved by *T+F* are lower, as shown in Figure 12(b). In this case, a large number of queries see speedups in the range of 1.2-1.3x. Overall, the random query experiments confirm that the *T+F* optimizations can significantly improve the performance of the baseline.

5.5 Join Performance

While previous experiments run queries on a single tree-structured data set, we focus on the join performance of Steed in this subsection. In particular, we join the 5.6 GB twitter data set (denoted `twitter`) with the 44 MB twitter data set (denoted `twitter.small`). We run the following two join queries:

- *Join query 1*: All the nodes in the query are at level 2.

```
select ts.user.lang, max(t.user.statuses_count)
from twitter t, twitter.small ts
where t.user.listed_count CMP CONSTANT
      and t.user.id = ts.user.id
group by ts.user.lang
```

- *Join query 2*: All the nodes except `t.user.listed_count` are at level 3. The filtering predicate is kept the same as join query 1 so that both queries select the same sets of records as the inputs to the join operator at a given selectivity.

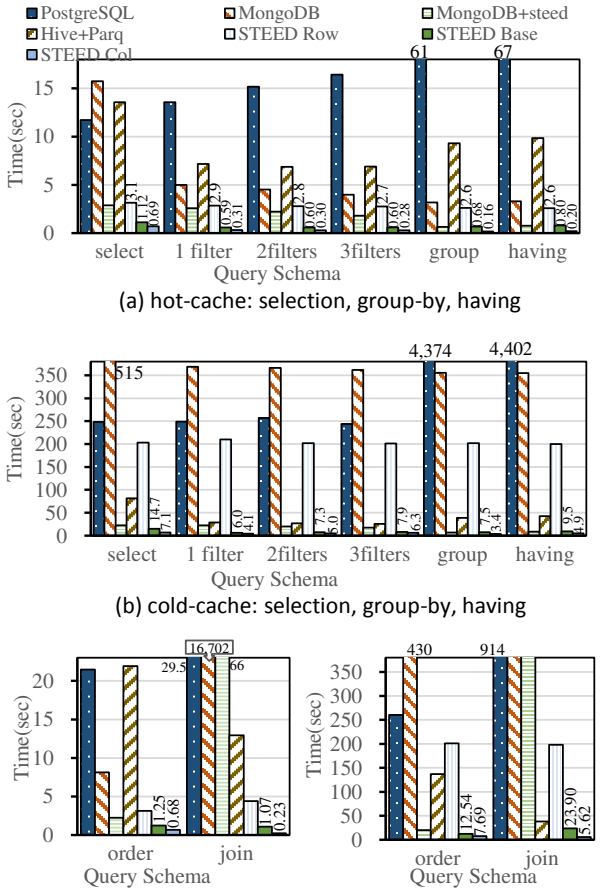
```
select ts.retweeted_status.user.lang,
   max(t.retweeted_status.user.statuses_count)
from twitter t, twitter.small ts
where t.user.listed_count CMP CONSTANT
  and t.retweeted_status.user.id
      = ts.retweeted_status.user.id
group by ts.retweeted_status.user.lang
```

Figure 13 shows the performance of the two queries varying selectivity. We see that *T+F* improves the baseline by a factor of 1.28–1.92x for join query 1, and 1.28–1.98x for join query 2.

Compared to the improvements for queries on a single data set, the improvements for join queries are higher. In a join, columns from both input data sets need to be assembled. The join operator needs to access join keys and payload attributes to perform the join operation in main memory. The *T+F* scheme employs the flat assembler to avoid the complex state machine and tree traversal in the baseline scheme for assembling columns. Moreover, it keeps only a single flat structure rather than the nested row layout in the baseline. Every flat attribute requires less space and fewer operations to access. Compared to queries on a single data set, join queries need to access and merge records from both inputs. The flat structure makes this process simpler and more efficient. As a result, the optimized *T+F* scheme achieves higher speedups for join queries.

5.6 Comparison with State-of-the-art Systems

Finally, we compare Steed with state-of-the-art systems, PostgreSQL, MongoDB and Hive+Parquet. In the case of PostgreSQL, we write SQL queries using the JSON operators and functions defined by PostgreSQL. In the case of MongoDB, we write our queries in javascript and run them via MongoDB's shell. MongoDB provides several ways to deal with query output. We choose 'toArray()' to return all query results as array. In the case of Hive, we directly run SQL queries via Hive's shell, which can return the JSON results. In addition to the three systems, we also implemented a hybrid system that runs MongoDB on top of Steed (MongoDB+Steed). MongoDB stores binary JSON records (a.k.a. BSON) in an underlying storage management system, called wired



(a) hot-cache: selection, group-by, having

(b) cold-cache: selection, group-by, having

(c) hot-cache: order-by, join (d) cold-cache: order-by, join

Figure 14: Comparison with state-of-the-art systems.

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. We perform hot cache experiments using the 5.6 GB data set, and cold cache experiments using the 45 GB data set. All the execution times are measured using the Linux time command.

Figure 14 shows the performance comparison. *Steed row* processes binary row data. *Steed base* and *Steed column* are the baseline and the T+F optimized versions of Steed that processes binary column data, respectively. The figures report the cold cache and hot cache performance of different types of queries: selecting all records (*select*), selection with 1–3 filtering predicates (*1filter*, *2filters*, and *3filters*), selection with group-by and aggregation operations (*group*), adding a having clause to the group-by query (*having*), selection with an order-by clause (*order*), and a join query (*join*). Appendix A lists the queries used in the comparison. In the figure, the range on the Y-axis is chosen to clarify the comparison. In some cases, the bars are too high to be cut off. In such cases, we label the bars with the execution times.

Cold Cache Comparison. Figure 14(b) and (d) show the cold cache results. The row and column layouts incur significantly different amount of I/Os. Comparing the two native systems that use row data layouts, i.e. MongoDB and Steed Row, we see that Steed Row achieves 1.8–2.5x speedups (excluding the join query[10], for which MongoDB runs for a very long time and the speedup is over 1010x). This is because Steed stores attribute type information in the schema tree, while MongoDB's BSON format stores type information in every binary record. Therefore, Steed saves space for storing data, and reduces the amount of I/Os for reading the data.

Comparing the two native systems with column layouts, i.e. Hive +Parquet and Steed column, we see that Steed column achieves 4.1–17.8x speedups over Hive+Parquet. Hive runs MapReduce jobs on top of Parquet, leading to the slower performance. Steed column achieves 1.2–2.2x speedups over Steed base, which shows the effectiveness of our optimizations.

The two native systems with column layouts are faster than both PostgreSQL and the two native systems with row layouts because the queries access only a small subset of attributes. Compared to MongoDB, Steed column achieves 55.9–105.2x improvements (excluding the join query). Compared to PostgreSQL, Steed column achieves 33.8–1294x speedups, where the 1294x speedup occurs for the group-by aggregation query.

MongoDB+Steed uses column layout instead of the original row layout. This leads to 16–51x speedups. On the other hand, the query processing in MongoDB uses BSON formats extensively. BSON embeds field names as strings. Field accesses need to perform string comparisons, which are significantly slower than the binary in-memory layout in Steed. Steed column is 1.8–5.5x faster than MongoDB+Steed.

Hot Cache Comparison. Figure 14(a) and (c) show the hot cache results. PostgreSQL is often the slowest system. This means that storing and processing entire JSON records in relational columns may incur significant overhead. Hive runs Java-based MapReduce jobs for queries. This leads to longer execution times when data fits in memory. Compared to MongoDB, Steed row achieves a factor of 1.2–5.0x improvements (excluding the join query, where the speedup is 3794x). On the other hand, Steed column reduces the amount of data copying operation compared to systems that process row data. As a result, it sees higher performance benefits. Steed column achieves 11.9–22.6x speedups over MongoDB (excluding the join query), 19.5–59.3x speedups over Hive+Parquet, and 16.9–392x speedups over PostgreSQL in the hot cache scenario. Moreover, Steed column achieves at least 1.6x speedups over the baseline Steed. MongoDB+Steed is a solution in between Steed and MongoDB, which is 3.8–8.4x slower than Steed, but 1.9–5.5x faster than MongoDB.

6. CONCLUSION

In conclusion, we have performed an in-depth study of real-world data patterns for tree-structured data. We find that simple root-to-leaf paths dominate. Therefore, we optimize for the common patterns in our Steed design. Experimental study confirms that this is indeed a good idea. Our optimizations achieve significant improvements over the baseline. Compared to three state-of-the-art systems (i.e. PostgreSQL, MongoDB, and Hive with Parquet), Steed achieves orders of magnitude better performance in both cold cache and hot cache scenarios.

7. 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.

[10]We implemented the join query using MongoDB's new lookup API. There is apparently significant overhead for this API.

8. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Apache HBase. http://hbase.apache.org/.

[3] Apache Parquet. http://parquet.apache.org/.

[4] Arduino. https://www.arduino.cc/.

[5] BeagleBoard. http://beagleboard.org/.

[6] CouchDB. http://couchdb.apache.org/.

[7] data.gov. https://www.data.gov/.

[8] DBpedia. http://wiki.dbpedia.org/.

[9] DragonBoard. https://developer.qualcomm.com/hardware/dragonboard-410c.

[10] IMDB. http://www.imdb.com/.

[11] JSON. http://www.json.org/.

[12] JSON Hypermedia API Language. https://tools.ietf.org/id/draft-kelly-json-hal-03.txt.

[13] Linked Data. http://linkeddata.org/.

[14] Memcached. http://memcached.org/.

[15] MongoDB. https://www.mongodb.com/.

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

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

[18] Sina Weibo API. http://open.weibo.com/wiki/.

[19] Twitter Feed. https://dev.twitter.com/streaming/public.

[20] Yahoo Developer Network. https://developer.yahoo.com/.

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

[22] F. N. Afrati, D. Delorey, M. Pasumansky, and J. D. Ullman. Storing and querying tree-structured records in dremel. *PVLDB*, 7(12):1131–1142, 2014.

[23] 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.

[24] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD*, pages 479–490, 2006.

[25] 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.

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

[27] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*, 2013.

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

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

[30] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. MacNaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zaït. Oracle database in-memory: A dual format in-memory database. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1253–1258, 2015.

[31] 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.

[32] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD international conference on Management of data*, pages 135–146, 2010.

[33] 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.

[34] H. Plattner. The impact of columnar in-memory databases on enterprise systems (keynote). In *VLDB*, 2014.

[35] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11), 2013.

[36] 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.

[37] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

APPENDIX

A. QUERIES FOR COMPARISON WITH STATE-OF-THE-ART SYSTEMS

Select

```
select retweeted_status.user.id
from twitter
```

1 filter

```
select retweeted_status.user.id
from twitter
where retweeted_status.user.favourites_count > 1
```

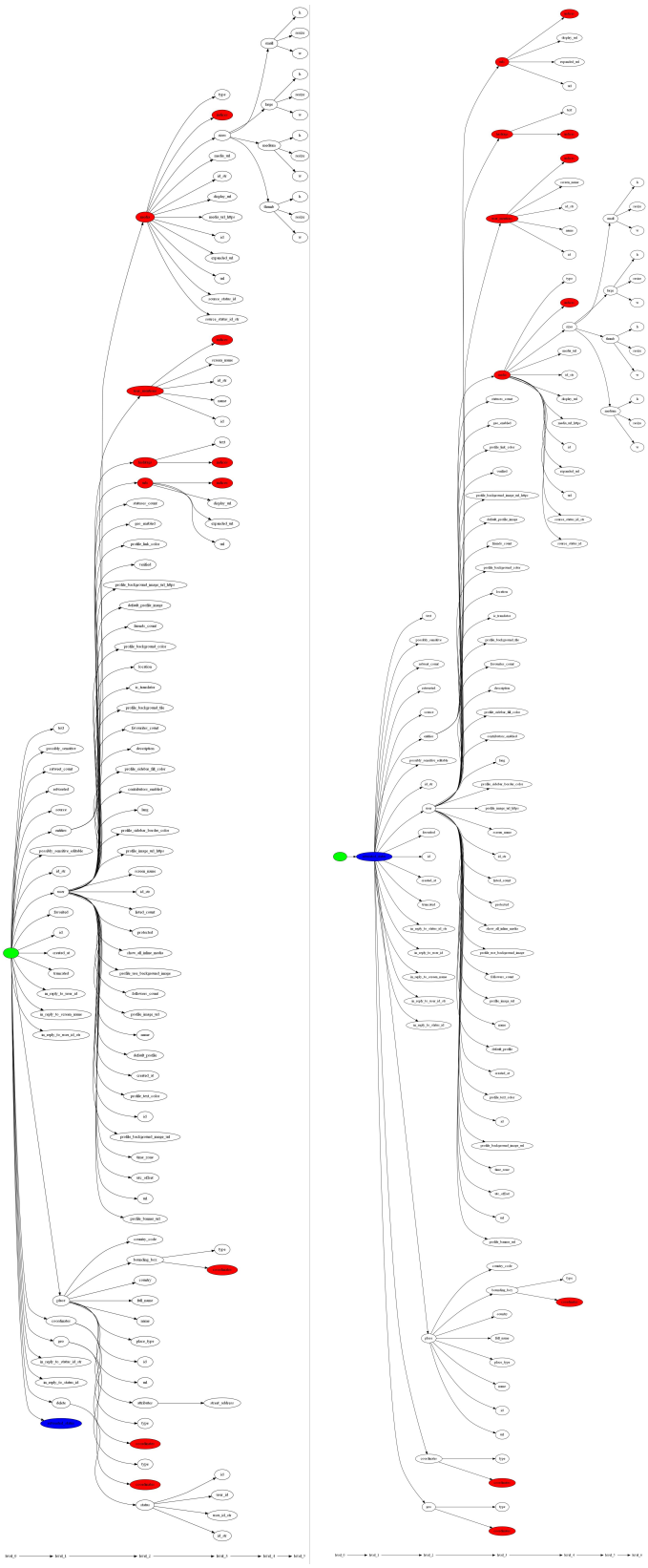
2 filters

```
select retweeted_status.user.id
from twitter
where retweeted_status.user.favourites_count > 1
    and retweeted_status.user.friends_count > 110
```

3 filters

```
select retweeted_status.user.id
from twitter
where retweeted_status.user.favourites_count > 1
    and retweeted_status.user.friends_count > 110
    and retweeted_status.user.followers_count > 500
```

Group by



```
select retweeted_status.user.utc_offset,
    max(retweeted_status.user.followers_count)
from twitter
group by retweeted_status.user.utc_offset
```

Having

```
select retweeted_status.user.utc_offset,
    max(retweeted_status.user.followers_count)
from twitter
group by retweeted_status.user.utc_offset
having
  max(retweeted_status.user.favourites_count)>10000
```

Order by

```
select retweeted_status.user.id,
    retweeted_status.user.followers_count
from twitter
order by retweeted_status.user.followers_count
```

Join

```
select ts.user.lang, max(t.user.statuses_count)
from twitter.small as ts , twitter as t
where t.user.listed_count >= 1
    and ts.user.id == t.user.id
group by ts.user.lang
```

Figure 15: The schema tree of Twitter tweets. (The green node is the root of the tree. The tree is too large to fit into a single graph. Therefore, the subtree rooted at the blue node is shown on the right. Red nodes are repeated.)