Hybrid.JSON: High-velocity Parallel In-Memory Polystore JSON Ingest

Steven Ortiz, Caner Enbatan, Maksim Podkorytov, Dylan Soderman, Michael Gubanov
Department of Computer Science
University of Texas at San Antonio

***Abstract*—Hybrid.poly is an in-memory polystore data management system, able to ingest various kinds of data and run complex analytical workloads on the ingested data [Gubanov, 2017], [Podkorytov et al., 2017]. Hybrid.JSON, a part of [Gubanov, 2017] focuses on ingesting and querying JSON documents in the polystore.**

***Keywords*-Large-scale Data Management; Cloud Computing; Summarization; Human-Computer Interaction.**

I. Problem Statement

Variety of Big Data is one of the key characteristics that make it difficult to process and analyze. This property is defined by the large number of types of data that exist in the wild. The examples of such data types are relational tables, web pages, XML documents, JSON documents, graph data, binary data retrieved from sensors, streams of messages in social networks, audio and video files. The classic data management engines are usually best optimized for relational or semi-structured data.

The JSON data model organizes data into a lightweight format[json.org, 2017] having a tree-like structure where internal nodes are either javascript objects or arrays of objects, and leaves are atomic values, such as strings and numbers. Some large-scale storage engines, such as MongoDB[MongoDB, 2017], support this model. However, all of the existing the data is not limited to this format, so many popular databases miss the boat when it comes to accommodating a variety of different data formats.

Hybrid.poly [Gubanov, 2017], [Podkorytov et al., 2017] aims to support ingestion and querying data of different data formates from a variety of data sources. Hybrid.poly focuses on the facet of scalable JSON document ingestion and querying.

II. Solution

A. Ingestion

The underlying storage of the polystore is a high-performance distributed in-memory key-value store written in about 1 million lines of Java. Before our work, the storage supported loading relational data from CSV files. If a user wants to ingest the contents of a csv file *file-name.csv* into an in-memory dataset */collection-name*, they issue a query of the following format:

```
LOAD /collection-name 'file-name.csv'
```

To support ingestion of a JSON document into the storage, we take the following approach. First, we modify the existing query statement that performs the loading of the document into the storage by adding an extra parameter that designates the type of data being loaded. As a result, if a user wants to ingest the contents of *file-name.json* into an in-memory dataset *collection-name*, they issue a query of the following format:

```
LOAD /collection-name 'json' 'file-name.json'
```

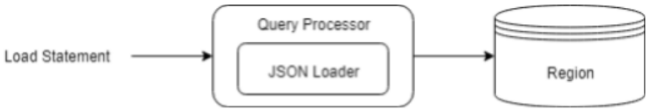

Figure 1. A generic schematic illustrating loading of a JSON document

Then, we serialize the JSON document into a Java object, so that the system can store it, send it over the network to other members of a distributed system. The persistence of in-memory objects on disk is another benefit enabled by serialization. When we load the JSON document into the key-value storage, we choose to use the absolute path to the loaded document as a key and the Java counterpart of the JSON document as a value. This seems to be intuitive, as the path to a filename is unique on a same filesystem, however, when this system is exposed to users on multiple hosts, we will incorporate the information about the host name into a key to keep the key unique.

As an initial approach, we chose to use the org-json[Crockford, 2017] parser. In this approach, we also developed our own set of Java classes to fix the API that reflects the structure of JSON documents, mapping them to Java objects. Having the mapping fixed allows to implement the storage without concerns of the API provided by the parser. We developed the mapping as a separate application to enable rapid development, testing and benchmarking, and we integrated it into the polystore as well.

The class hierarchy of the mapping is as follows. The root type, JsonType, is extended by JsonObject, JsonArray and JsonPrimitive. A JsonObject contains a map of keys of type String to values of type JsonType; a JsonArray contains a list of values of type JsonType and a JsonPrimitive contains an atomic value. Figure 2 depicts the class diagram illustrating the mapping of JSON documents into Java objects.

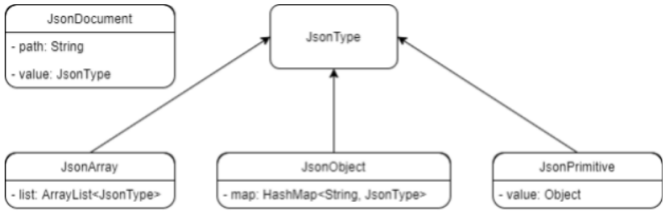

Figure 2. Class diagram illustrating how JSON objects map to Java objects

As a result of our efforts, users can load entire JSON documents into memory and perform basic queries on them very rapidly. For example, a user may issue 3 loading queries followed by a select-everything query:

```
LOAD /test-dataset 'json' 'sample1.json'
LOAD /test-dataset 'json' 'sample2.json'
LOAD /test-dataset 'json' 'large.json'
SELECT * FROM /test-dataset
```

The result of the queries is displayed in Table I. Currently, we display a brief summary of each document.

summary	path
JsonObject containing 8 element(s)	[...]/sample1.json
JsonArray containing 6 element(s)	[...]/sample2.json
JsonObject containing 200000 element(s)	[...]/large.json

Table I
WILDCARD QUERY RESULTS AFTER LOADING THREE DOCUMENTS

As we progressed with the JSON ingestion module, we discovered, that using our mapping, while being convenient, introduces performance and memory overhead. Using the org-json parser did not allow us to ingest the documents with the number of keys greater than 2^{26}. Given the amount of memory on the test machine, we attribute this to the parser limitations. Another JSON parser, Jackson[FasterXML, 2017], performed better on large documents, although it did not allow us to ingest the documents with number of keys greater than 2^{27}, making an improvement of only one binary order of magnitude. We have evaluated the org-json parser with the mapping, the org-json parser without the mapping and the jackson parser in Section III.

B. Querying

The polystore answers the users' queries in a dialect of SQL extended with java objects' properties traversal. This traversal is specified with a common in OOP languages dot-notation of accessing the object's properties. For example, the following listing declares the class Employee, implementing the Serializable interface to store the objects of that type in our storage engine.

```
class Employee implements Serializable {
  public String name;
  public Integer id;
  public Date birthday;
  // implementation details omitted
  // ...
}
```

If a dataset *employee-set* contains Java objects of type *Employee*, it is possible to query all of the names by running the following query:

```
SELECT e.name FROM /employee-set e
```

Furthermore, in case of nested objects, a user may chain the attributes using a dot notation. For example, a dataset *jobs-set* contains objects of type *Job* declared in the following listing.

```
class Job implements Serializable {
  public String name;
  public Integer id;
  public Employee employee;
  // implementation details omitted
  // ...
}
```

The names of all employees associated with jobs are accessible with the following query:

```
SELECT j.name, j.employee.name
FROM /job-set j
```

As the polystore provides us with this querying functionality, and the JSON documents are inherently recursive, it is possible to extend it to be able to query the JSON objects. For example, here is a fragment of a JSON document that contains the web pages:

```
{
  "blog": {
    "posts": [...],
    ...
  },
  ...
}
```

We plan to further improve native support for querying of JSON documents as follows. If a user wants to access the first blog post in the document, they will issue the following query:

```
SELECT d.'blog'.'posts'.0 FROM /document d
```

Enclosing the keys within single quotes enables keys that contain spaces and dots. Furthermore, having some properties quoted allows us to distinguish between keys and array indices, as we did in the posts-retrieving query.

III. EVALUATION

The architecture of our solution allows to use different JSON parsers and compare their performance. We generated a set of documents, each of the documents contains a large number of key-value pairs, having integer keys and string values. We evaluated the ingestion of each document in three

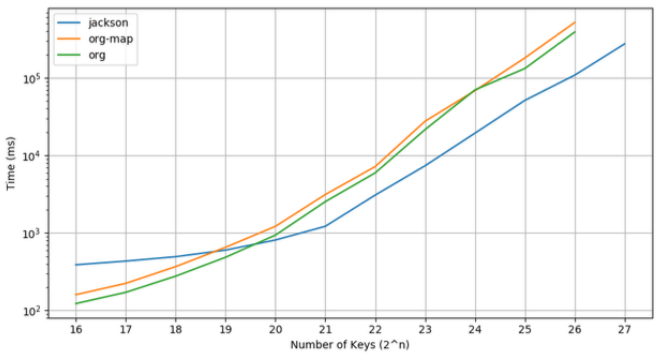

Figure 3. Comparison of ingestion speed using 3 different approaches. The green line corresponds to the org-json parser without mapping step, the orange line corresponds to the org-json parser with mapping step, the blue line corresponds to the jackson parser without mapping step. The x-axis shows the number of keys in the upper level of the JSON document. The document is a generated array of key-value pairs.

scenarios: using the org-json parser with mapping step enabled, using the org-json parser with mapping step disabled, and using the jackson parser with mapping step disabled. For each of the documents and evaluation scenarios we ran 10 tests to account for randomness, the detailed results for org-json and jackson parsers with the mapping step disabled are shown in Tables II and III. The Figure 3 shows the average running time between the 10 runs for each of the documents and evaluation scenarios. As we can see from the figure, having the mapping enabled makes performance worse. We can also see that the org-json parser performs better than the jackson parser on smaller documents, while it performs worse on bigger documents. The ingestion was evaluated on a server with 4 Intel® Xeon® E7-4870 CPUs, 80 2.40 GHz cores and 320 GB of RAM.

Keys	Size	Mean(ms)	Min(ms)	Max(ms)
2^{22}	79.76M	5940.83	5347.95	6298.94
2^{23}	163.76M	21499.28	17296.87	23480.38
2^{24}	331.76M	69825.22	60022.25	123813.93
2^{25}	693.61M	131854.24	113760.88	167162.22
2^{26}	1.40G	389190.08	327658.00	476184.90
2^{27}	2.83G	-	-	-

Table II
THE RESULTS OF RUNNING INGESTION OF GENERATED DOCUMENTS, USING THE ORG-JSON PARSER, SKIPPING THE MAPPING STEP, SAMPLING 10 TIMES FOR EACH DOCUMENT

IV. Future work

Implementing the JSON documents ingestion left a few questions unanswered, and we plan to investigate them in our future work. Improving native support of querying JSON documents is our top priority. Summarizing the JSON documents is a challenge, as documents can be arbitrarily nested. Key search within a document can also be implemented in a way similar to XQuery[Chamberlin, 2003]. Investigating how the JSON ingestion performs on documents with complex structure is an interesting direction as well. Finally, implementing joins with relational data stored in the engine is also an attractive option.

Keys	Size	Mean(ms)	Min(ms)	Max(ms)
2^{22}	79.76 M	3071.76	2867.40	3339.36
2^{23}	163.76 M	7343.85	5826.80	12351.58
2^{24}	331.76 M	19330.26	15867.73	22102.52
2^{25}	693.61 M	51148.33	45644.97	58369.97
2^{26}	1.40 G	108248.99	97224.47	144191.34
2^{27}	2.83 G	273743.91	216575.40	413781.20

Table III
THE RESULTS OF RUNNING INGESTION OF GENERATED DOCUMENTS, USING THE JACKSON PARSER, SKIPPING THE MAPPING STEP, SAMPLING 10 TIMES FOR EACH DOCUMENT

V. Related Work

MS SQL Server enables storing and querying both relational and JSON data[Microsoft, 2017]. Hybrid.poly is using the distributed in-memory storage.

Multiple data format support is the selling point of the polystores, the systems that enable storing and querying heterogeneous data coming from different sources with different data models[Gubanov, 2017], [Podkorytov et al., 2017]. The JSON data format is supported by several engines, for example MongoDB[MongoDB, 2017] is a popular large-scale JSON storage engine.

References

[Chamberlin, 2003] Chamberlin, D. D. (2003). Xquery: A query language for xml. In *SIGMOD Conference*.

[Crockford, 2017] Crockford, D. (2017). Online: Maven repository: org.json — json.

[FasterXML, 2017] FasterXML (2017). Online: Fasterxml/jackson: Main portal page for the jackson project.

[Gubanov, 2017] Gubanov, M. (2017). Polyfuse: A large-scale hybrid data fusion system. In *ICDE*.

[json.org, 2017] json.org (2017). Online: Json.

[Microsoft, 2017] Microsoft (2017). Online: Json data (sql server) — microsoft docs.

[MongoDB, 2017] MongoDB (2017). Online: Mongodb for giant ideas — mongodb.

[Podkorytov et al., 2017] Podkorytov, M., Soderman, D., and Gubanov, M. (2017). Hybrid.poly: An interactive large-scale in-memory analytical polystore. In *ICDM DSBDA*.