Query Service for REST APIs

Query Service for REST APIs

Marwan Sabbouh

1 Introduction

Big Data and Micro Services architectures make heavy use of ReST APIs that are backed by NoSQL databases and full text search engines. Notably missing from these architectures are relational databases due to the requirements of high throughput, high availability, and low latency on CRUD operations. To provide querying capabilities, full text search engines (e.g. Elastic Search) are starting to provide limited join queries capabilities. Additionally, distributed graph query engines (Titan) and graph query languages (e.g. GraphQL), are gaining in popularity. While these technologies are filling the functional gaps, they are also increasing the complexities of big data architectures. In this paper, I describe an approach for querying data that only leverages full text search and NoSQL databases. The approach builds on the conventions specified by data interchange protocols (e.g. DIP, gData, OData) to define a fast graph based indexing technique and an addressing scheme for all entities consumed by the ReST APIs. The graph based indexing algorithm runs in real-time as the data is ingested through the ReST API. The indexing algorithms store the generated indices in the NoSQL database, which are used by the query service for query resolution. In this short paper, I will describe the indexing algorithm and query resolution. For addressing, we will assume an addressing scheme like JSON Path.

2 Background

In this section, I define terminology that helps in explaining the indexing algorithm.

1. A graph is comprised of nodes and relationships. Each graph node is uniquely identified through its type and its unique identifier. A source node is connected to a target node through a relation. Consequently, we define a relation as having a source node and a target node.
2. The domain of the relation is the type of the source node.
3. The range of the relation is the type of the target node.
4. A JSON Object is a collection of property/value pairs. There are two types of properties in a JSON Object: simple properties, and object properties. A simple property is a property whose value is not a JSON Object. An object property is a property whose value is a JSON Object.
5. A JSON Object consisting only of simple properties is a node in the graph.
6. An object property is a relation in the graph.
7. The value of an object property consisting only of simple properties is a target node in the graph.

Figure 1 illustrates a JSON document describing comments on a blog entry. The JSON document consists of three JSON Objects. The root object with simple properties _type, _id, title and with the object property comments. The value of the comments property is another JSON Object with simple properties _type, _id, created and with the object property author. The value of the author property is another JSON Object with simple properties _type, _id, and name.

```
{"_type": "blog",
"_id": "123456",
"title": "novel indexing techniques",
"comments":[ {
                    "_type": "comment",
                    "_id": "78910",
                    "created": "05-12-2017",
                    "author": {
                              "name": "Michael Smith", // NOTE: Duplicated field
                              "_id": "121314",
                              "_type": "person"
                              "_inferred":[" blog-person"]}
          }]
}
```

Figure 1: Sample JSON Document

Figure 2 shows the graph model of the JSON document.

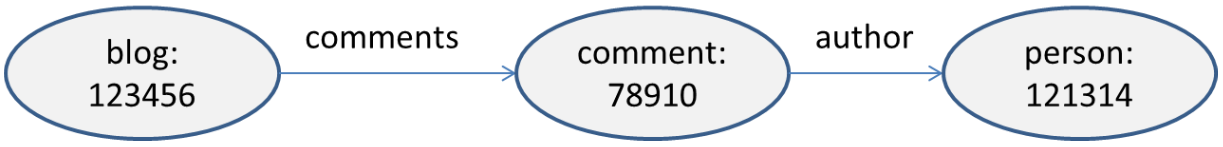

Figure 2: Graph Representation of Figure 1

3 Indexing Algorithm

1. First, the indexing algorithm interprets the nested document shown in figure 1 as a directed graph. This is shown in figure 2.

2. Second, the indexing algorithm makes use of inverseOf inference and transitive inference. The resulting graph is shown in figure 3.

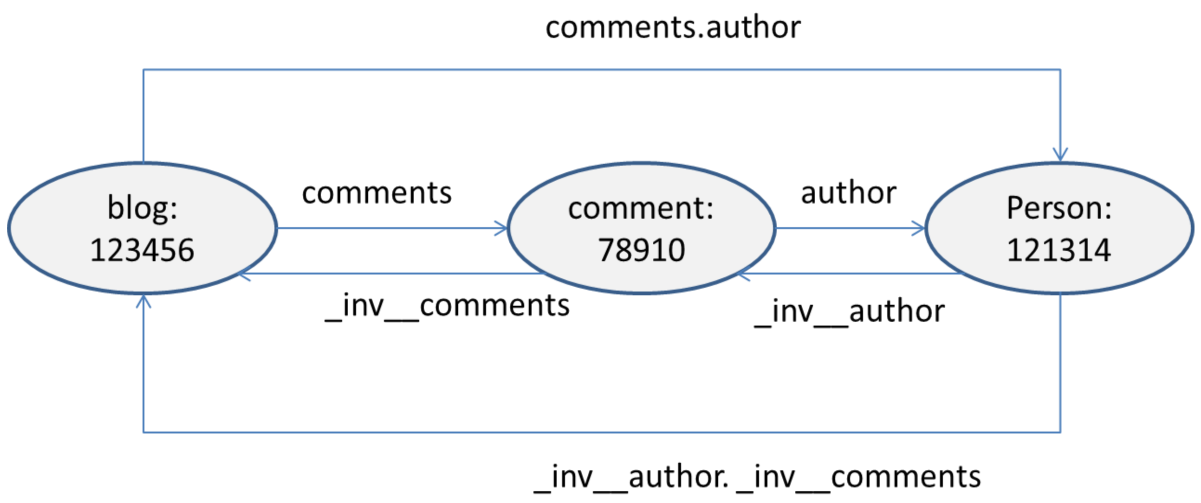

Figure 3: Inferred graph of figure 2

Figure 3 shows _*inv__comments* as the inverse relationship of *comment*, _*inv__author* as the inverse relationship of author, and _*inv__author*. _*inv__comments* as the inverse relationship of *comments.author*. These inverse relationships are generated automatically by the indexing algorithm. Figure 3 also shows the transitive relationship that was inferred by the algorithm. In this example, the transitive relationship is shown to be *comments.author*. Note that, we left the algorithm generate the transitive relationship name and the best it could do is something like "*comments.author*". The algorithm also infers the relationship between nodes blog:123456 and person:121314 . This eliminates the need of the data modeler defining inverseOf and transitive relationships.

3. Third, the indexing algorithm infers the range of all transitive relationships. In this example, the transitive relationship "comments.author" has "blog-person" as *inferred* object type for its target object.
4. Fourth, the indexing algorithm annotates the target object of all transitive relationships with their inferred type. For the example shown in figure 1, the target object of the transitive relationship "comments.author" is person_121314. It contains the property _inferred with its value "blog-person".
   Internally to our system, the indexing algorithm represents the inferred graph as shown in figure 4:

1. comment_78910_ _inv__comments: [blog_123456]
2. comment_78910_ author: [person_12314]
3. person_121314_ _inv__author: [comment_78910] *
4. person_121314_ _inv__author._inv__comments: [blog_123456]*
5. blog_123456_comments: [comment_78910]
6. blog_123456_comments.author: [person_121314, blog-person_121314]*
7. relation_ _inv__author._inv__comments_inverseOf: [comments.author]*

8. relation_ _inv__author_inverseOf: [author]*
9. relation_comments.author_inverseOf: [_inv__author._inv__comments]*
10. relation_ author_inverseOf: [_inv__author]*
11. relation_comments.author_inferred: [blog-person_121314] *

Figure 4: Semantic index

Note that in figure 4 the lines ending with an asterisk indicates inferred knowledge.

4 Querying Algorithm

To illustrate how the indexing algorithm can be used in search, I present the following example. Suppose a user would like to find all persons whose first name Michael and who have commented on blogs. Assuming the availability of an inverted index search engine, e.g. Elastic Search, the query syntax may look something like this:

/search/indexname/blog?q = comment. author. name: "Michael*".

To understand the search query, the URI path is of the following form: /search/indexname/{object type} followed by a typical Lucene query parameters. The fact that the object type is blog, tells the search engine to return instances of object type blog that matches property name with the value starting with "Michael". The JSON path of the property name in the nested document is "comment.author.name". In this case, the JSON document shown in figure 1 will be returned. Please observe that for this query to work the property "name" of object type person must be <u>duplicated</u> in the nested object of object type person.

To avoid data duplication, we can use the semantic index to do the join query. In this case, the query remains the same as shown above: /search/indexname/blog?q = comment.author.name: "Michael*". The query parser does the following:

1. Splits the JSON path "comment.author.name" into relationships comments.author, and property name
2. Find the range of the relation comments.author. This returns type person, and the inferred type blog-author.
3. Form the query to find all instances of person with the property name starts with Michael: /search/indexname/person?q = name: "Michael*" & field: "_id".
4. Append to the query above the inferred type as a query parameter. The above query becomes: /search/indexname/person?q = name: "Michael*" & field: "_id" & _inferred: blog-person.
5. Executes the query. It returns the JSON instances of type person with _inferred property set to blog-person, and the name property starting with Michael. However, due to the query parameter field: "_id" the search engine only returns a list of all the JSON instance identifiers that matches the query. For this simple example, the returned list will contain 121314 which is the value of the property "_id" of the JSON instance of type person.
6. Find the inverse of relationship comments.author (found in the first step) by looking up in the semantic index the value of relation_comments.author_inverseOf . This returns relation _inv__author._inv__comments.
7. Find in semantic index the value of the key person_121314_ _inv__author._inv__comments. This returns blog_123456

The above algorithm demonstrates how to accomplish join query without data duplication. That is, we could remove the name property from the JSON document of figure 1, and the search query will still return the correct result. This is because the query in step 3 is to search for instances of object type person.

5 Experimental Results

I implemented the indexing algorithm as part of the ReST API and the query service rest API. I used Redis in conjunction with Elastic Search to implement the system. The ReST API was implemented in Java using Spring Boot. The early results are quite encouraging. We can index a JSON document consisting of 5000 objects in less than 30 ms. A sophisticated join query consisting of three conditions on different nodes in the graph returned in less than 30 ms. Furthermore, the indexing algorithm also proved useful in implementing merge/patch functionality for ReST API, in addition to playing a key role in business rule specification and enforcement. I will describe the latter technique in another document.

6 Conclusion

As we require more functionality from big data technologies, we are faced with increased technological complexities that are preventing many small to medium-size companies from adopting these technologies. For these architectures to become pervasive, techniques that simplify the big data technological stack are critically needed. I believe the approach described here is a step in that direction.