A Graph Database for Persistent Identifiers

Triet Doan[1][0000−0002−7247−9108], Lena Wiese[2][0000−0003−3515−9209], Sven Bingert[1][0000−0001−9547−1582], and Ramin Yahyapour[1][0000−0002−9057−4395]

[1] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Am Faßberg 11, 37077 Göttingen
{triet.doan|sven.bingert|ramin.yahyapour}@gwdg.de
https://www.gwdg.de
[2] L3S Research Center / KBS Group, Leibniz University Hannover
Appelstraße 4, 30167 Hannover
wiese@l3s.de

Abstract. The Handle Software manages references to resources of information. However, it does not support a search functionality. A prior implementation with Elasticsearch could not efficiently capture the complex structure of our dataset, especially the relationships between handles. In this paper, we apply a graph database together with Elasticsearch to provide more search capabilities to users. In addition, the graph can efficiently store meta-data provided during handle creation. Further use cases for this graph include redundancy detection (two or more handles pointing to the same URL), or bibliographic network analysis.

Keywords: Persistent identifier · Handle System · Graph database · Neo4j · Elasticsearch.

1 Introduction

Nowadays, people often locate digital objects using Uniform Resource Locators (URLs). However, URLs tend to be broken over time [2]. To overcome this problem, the concept of Persistent Identifier (PID) is introduced. As the name suggests, a PID is an identifier which is valid for a long time. In practice, a PID is mapped to an up-to-date URL [1].

According to FAIR (Findable, Accessible, Interoperable, Reusable) principles, data with PIDs and their meta-data are supposed to be findable [4]. However, there is currently no efficient tool to find PIDs from their meta-data. In prior work, a search engine was created using Elasticsearch. Although it solved the search problem, it did not efficiently capture the complexity of our dataset. The contribution of this paper is to introduce a graph database as a tool that is able to perform advanced searches on PID data; it is also able to search based on the relationships between digital objects.

The paper is organized as follows. Section 2 discusses the system design. The system is evaluated in Section 3 and the conclusion is presented in Sections 4.

2 System Design

Graph databases – a special category of NoSQL databases [3] – represent information by *nodes* and *relationships* and store data in so-called *properties*. The purpose of our system is to employ a graph database to maintain the complex structure of the handle data and to provide a search function together with the ability to explore and analyze the data. To achieve that, a database which is optimized for graph storage and traversal is required. Therefore, the graph database Neo4j[4] that implements the property graph model was chosen. An implementation of PID is the Handle Software, which is developed by CNRI[5]. Every handle consists of two parts: its naming authority (known as its prefix), and a unique local name under the naming authority (known as its suffix). The main disadvantage of the Handle Software is, that it does not provide a search function. There are no restriction on the creation of handle values. Hence, when the system processes a handle value, it does not know the meaning of each data type due to the lack of standardization. To overcome this problem, a schema shown in Figure 1 is used in our system. The solution is to use many smaller nodes where each node contains only one property instead of one node with many properties. In this schema, except the handle nodes which are labeled as `handle`, the label of nodes and relationships are the data types of the handle values, such as `URL` or `Institute`. In this schema, every node is unique: handles which have the same handle values will point to the same nodes.

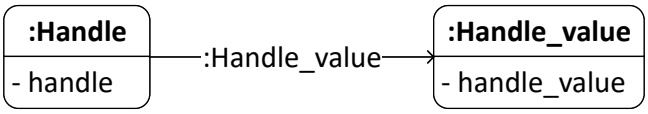

Fig. 1. The schema that used to build a graph from handles and handle values

Table 1 presents an example with three situations which need to be carefully examined. The first deals with data types from the Information Type Registry (ITR)[6]: a data type from the ITR must be resolved to get back a human-friendly type name. For example, resolving the value `21.11104/3eaedeaced10be5805d2` returns `isPreviousVersionOf` as the name of that data type. The second situation occurs when a handle is related to another handle. For example, handle `10.123/456` is the previous version of handle `10.123/789`. To indicate this connection, handle `10.123/456` contains a handle value `10.123/789` with type name `21.11104/3eaedeaced10be5805d2`, which after being resolved is actually `isPreviousVersionOf`. When the system processes this case, instead of creating a new node for each handle value as usual, it only creates a relationship from

[4] https://neo4j.com
[5] https://www.handle.net
[6] http://dtr.pidconsortium.eu

the handle node `10.123/456` to the handle node `10.123/789`. Only when the handle node `10.123/789` is not found, the system will create that node first, then add a relationship between those two handle nodes. Lastly, there are many cases where a handle value does not have an atomic value but a JavaScript Object Notation (JSON). In a naive approach, the system will create a new node and put the whole JSON inside. However, doing so leads to disadvantages. First, because it is just a string, it is very hard to distinguish between the key and the value to search on. Second, all the structures inside the JSON as well as the connections with other nodes are lost. Hence our system must parses each JSON object and creates an appropriate graph from it. The graph in Figure 2 shows what our system generates from the example data in Table 1. As can be noticed from the figure, there are two empty nodes in the graph. These empty nodes are the results of the JSON parsing process. The purpose of these nodes is to group related data together.

Table 1. An example showing that handle `10.123/456` is a previous version of handle `10.123/789`, which is indicated by an ITR value. There is also a JSON inside the `Creator` handle value of handle `10.123/789` that needs to be parsed.

Handle	Data Type	Index	Value
10.123/456	URL	1	http://www.gwdg.de
	Email	2	triet.doan@mail.com
	Name	3	Triet Doan
	INST	4	GWDG
	21.11104/3eaedeaced10be5805d2	5	10.123/789
10.123/789	URL	1	http://www.google.com
	Email	2	triet.doan@mail.com
	Creator	3	`{ "first_name": "Triet", "last_name": "Doan", "address": { "country": "Germany", "city": "Göttingen" } }`

To improve the performance, each node will have one more property called `nodeId`. This property is unique among nodes and used as the key of a node. When a node is created, its `nodeId` is calculated by hashing the value of that node. This process is applied for non-handle nodes. Because the handle string is already unique, the `nodeId` of a handle node is the handle string itself. The `nodeId` property is indexed with a unique constraint. While indexing enhances the performance of the READ operation, other operations (CREATE, UPDATE, DELETE) are slowed down due to the updating of index table. Indeed, our graph database is under a heavy load of CREATE and DELETE operations.

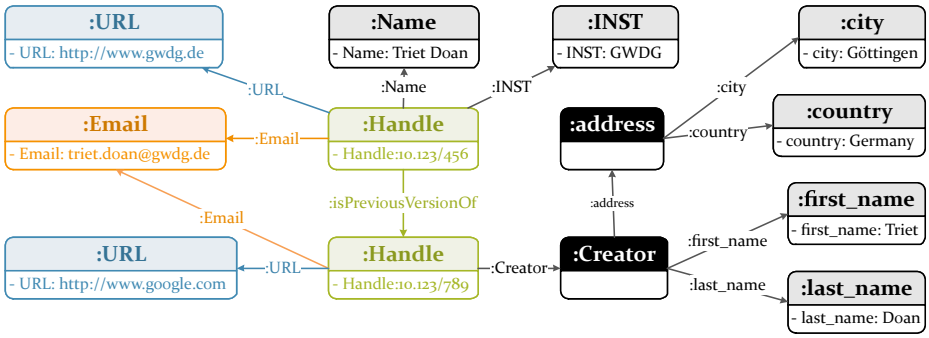

Fig. 2. An example graph which includes all cases that the system has to process

However, because of the uniqueness of every node in our graph database, one single CREATE or DELETE involves many READ operations which greatly benefit from the index. We hence observed that indexing leads to a huge boost in the performance of the system (see Section 3).

3 Evaluation

The execution time was measured when the system was running under a heavy load scenario. During this time, the system had to retrieve data from two data sources and build a graph with around 1 million nodes and 2.5 million relationships. Figure 3 shows the number of handle values processed by the system per minute. The lower green line shows the execution time when data was collected without hashing and indexing. As can be seen from the chart, the system runs quite fast at the beginning with around 1000 handle values processed per minute. However, it quickly becomes slow over time. The reason for this performance loss is that whenever a node is created, the system must make sure that the node is unique. Therefore, the more nodes it has, the longer the checking time. After around 107 hours, which is about 4.5 days, the system became too slow. It processed only 130 handle values per minutes. This test was stopped after 119 hours (almost 5 days). If continued, it would have taken around 7 days to finish. For the second approach, with hashing and indexing, the performance was greatly improved as shown by the upper blue line in Figure 3. It can be seen that it runs quite stable with the number of processed handle values fluctuating between 2000 to more than 3000 per minute. By exploiting the indexing feature, the performance is increased by factor 7.

4 Discussion and Conclusion

Our first achievement is the appropriate graph schema for the handle data. That graph schema is able to deal with the flexibility in the creation of handle values

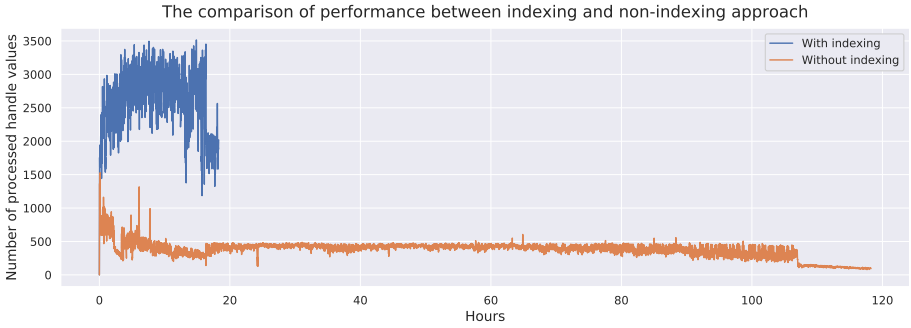

Fig. 3. Performance difference between indexing and non-indexing approach

as well as maintaining a good performance of the system. A search engine for handles is the second achievement. It offers a variety of search options from Elasticsearch and the ability to manage relationships between handles from Neo4j. Basic usages can be done through the Graphical User Interface (GUI), while a web-based tool is ready for more advanced purposes, such as some analyses which are performed to discover hidden knowledge inside the graph. A topic of future work to consider is the interoperability of the system: the graph database can be enriched by importing data from other platforms, such as DOI, ARK, ISBN, or ORCID.

References

1. Hakala, J., et al.: Persistent identifiers – an overview. KIM Technology Watch Report (2010)
2. Markwell, J., Brooks, D.W.: Broken links: The ephemeral nature of educational WWW hyperlinks. Journal of Science Education and Technology **11**(2), 105–108 (2002)
3. Wiese, L.: Advanced data management: for SQL, NoSQL, cloud and distributed databases. de Gruyter Publishing (2015)
4. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., da Silva Santos, L.B., Bourne, P.E., et al.: The FAIR Guiding Principles for scientific data management and stewardship. Scientific data **3** (2016)