

# A Comparative Study on Storing and Retrieval of URIs for Life Sciences Databases

Atsuko Yamaguchi<sup>1,\*</sup>, Yasunori Yamamoto<sup>2</sup>

<sup>1</sup>Tokyo City University, Setagaya, Tokyo, Japan

<sup>3</sup>Database Center for Life Science, Kashiwa, Chiba, Japan

## Abstract

In the field of life sciences, numerous databases are publicly available in the form of graph structures called RDF (Resource Description Framework). Each node in these graphs is linked to a unique URI representing entities such as genes and proteins. Consequently, these databases contain vast amounts of graph-structured data with numerous URIs. By establishing links to URIs in other databases, these databases become interconnected through links, ultimately forming a massive knowledge graph.

As a result, it is essential to develop a foundational infrastructure for storing and searching a significant number of URIs from life science data in this vast knowledge graph, utilizing database IDs and keywords. Presently, Front Coding is commonly employed for URI compression, but it suffers from a limitation in its inability to perform prefix matching searches. Trie is frequently used as an alternative approach capable of retrieving prefixes. This study assesses the compression ratio and search efficiency by comparing the utilization of Front Coding, Trie, and our proposed methods. The evaluation is conducted using a dataset of URIs extracted from life-science RDF datasets.

## Keywords

Semantic Web, Resource Description Framework, URI, Information retrieval

## 1. Introduction

In the life science field, a large number of heterogeneous data is produced. To facilitate their integration, RDF (Resource Description Framework) is frequently employed. These RDF datasets exhibit graph structures, with URIs serving as universal identifiers. The utilization of large RDF datasets necessitates the development of an efficient method for storing and retrieving a comprehensive set of URIs that appear within these datasets.

A previous study by Martínez-Prieto et al. [1] delved into a comparative analysis of Hashing, Front Coding, and FM-index, using dictionaries extracted from RDF datasets, focusing on URIs and literals. They also introduced a method capable of swiftly locating strings within the dataset, achieving compression ratios of 22% to 66%. However, it's noteworthy that their investigation did not explore the potential of Trie, a technique with the capability to compress sets of strings and retrieve not only the strings themselves but also their prefixes. Efficient federated search hinges on the ability to identify servers offering datasets with specific prefixes in an efficient

---

*IJCKG 2023*


\*Corresponding author.

✉ atsuko@tcu.ac.jp (A. Yamaguchi); yy@dbcls.rois.ac.jp (Y. Yamamoto)

🆔 0000-0001-7538-5337 (A. Yamaguchi); 0000-0002-6943-6887 (Y. Yamamoto)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

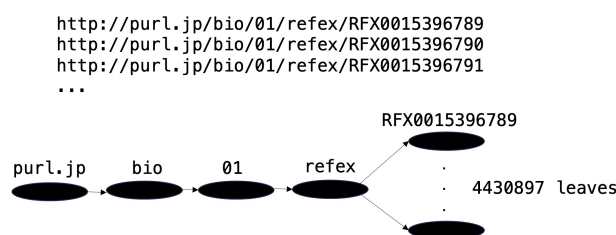
 CEUR Workshop Proceedings (CEUR-WS.org)

manner. In this poster, we undertake a comparative study of Front Coding, Trie, and two novel methods we propose. Our assessment is based on the average time required for locating, as well as the compression ratio achieved by these methods.

## 2. Computational Experiment

Front Coding is a technique employed to compress a set of sorted strings by comparing each string with its preceding string and encoding the shared substring to its length. To facilitate efficient location, the strings are subdivided into subsets, known as buckets. In this study, we employed a bucket size of 128 for Front Coding. Trie, on the other hand, is a tree structure designed for locating a specific string within a set of strings. In a Trie, each node corresponds to a character in a string, and each path in the Trie corresponds to a string within the set. These two methods are often used for locating a string from a set of strings.

However, we observed a distinctive feature among the URIs present in life-science RDF datasets: they tend to exhibit a limited number of prefixes and an abundance of suffixes. Based on this observation, we devised an innovative approach for URI location within a set of URIs. We propose breaking down URIs by "/" delimiter, with each substring corresponding to a node in a tree structure. As depicted in Fig. 1, the number of leaves in this tree can sometimes be substantial. To address this, we divided the search method into two phases. In the first phase, we search for the URI from the root to a leaf, akin to a Trie. If a node possesses a significant number of children, we employ a secondary method in the second phase. In this context, we present two distinct methods for the second phase: the first (Method 1) involves binary search for the children, while the second method (Method 2) entails constructing Tries for each child node.



**Figure 1:** A typical example of URIs in a life science RDF dataset. While there is a single path corresponding to the string from the root to the "refex" node, there are 4,430,897 leaves.

We utilized four life-science RDF datasets sourced from the RDF portal [2]. Due to variations in the magnitude of URIs, four datasets were chosen. Table 1 provides an overview of these datasets, including their names, the count of URIs, and the size of the URI lists, with *Quanto* being the smallest and *KERO* the largest. For the four datasets, it has been observed that typical structures of life science RDF URIs exhibit an explosive increase in the number of children from a certain depth.

It's worth noting that the tree structures utilized in both the Trie and our proposed methods are implemented using LOUDS (Level-Order Unary Degree Sequence) methodology [3]. This choice of implementation is made due to its smaller space usage in comparison to other methods, such as the double array. In our comparative analysis, we evaluated the performance of Front

**Table 1**

Four datasets used for computational experiment

| Name   | The count of URIs | The size (byte) |
|--------|-------------------|-----------------|
| Quanto | 1 995 998         | 113 026 882     |
| jPOST  | 11 412 967        | 510 998 241     |
| RefEx  | 19 828 641        | 832 801 751     |
| KERO   | 290 338 001       | 19 051 677 994  |

Coding, Trie, and our two methods (Method 1 and Method 2) using the RDF datasets. We focused on assessing the compression ratio and the average time required for URI location. To calculate the average time for URI location, we randomly selected 100 samples from the URI lists for each dataset. All of the methods were implemented in C++. The computational experiments were conducted on an Ubuntu 20.04.4 machine equipped with an Intel(R) Xeon(R) 2.40GHz CPU and 200GB of memory.

### 3. Result

Table 2 presents the compressed sizes and compression ratios. The compressed sizes encompass the necessary indexes for efficient location in the case of Trie and our methods. Notably, among the four methods evaluated, both Trie and Method 2 exhibit superior performance, with Method 1 showing the least favorable results, albeit achieving compression ratios ranging from 0.31 to 0.37. Table 3 illustrates the average time required for URI location based on 100 trials. Front Coding emerges as the top performer, closely followed by Method 1. While the compression ratios of Trie and Method 2 are remarkably similar, Method 2 stands out as being two to three times faster than Trie.

**Table 2**

Compressed size and compression ratio

| Name          | Quanto                  | jPOST                     | RefEx                     | KERO                        |
|---------------|-------------------------|---------------------------|---------------------------|-----------------------------|
| Original(bit) | 904 215 056             | 4 087 985 928             | 6 662 414 008             | 152 413 423 952             |
| Front Coding  | 174 363 760<br>(0.1928) | 328 912 984<br>(0.08045)  | 541 609 480<br>(0.08129)  | 11 110 721 176<br>(0.07289) |
| Trie          | 172 591 599<br>(0.1908) | 147 453 553<br>(0.03606)  | 220 324 169<br>(0.03306)  | 6 738 005 419<br>(0.04420)  |
| Method 1      | 285 735 820<br>(0.3160) | 1 443 548 986<br>(0.3531) | 2 103 835 010<br>(0.3157) | 57 241 426 238<br>(0.3755)  |
| Method 2      | 172 591 344<br>(0.1908) | 147 452 340<br>(0.03606)  | 222 547 974<br>(0.03340)  | 6 738 004 412<br>(0.04420)  |

### 4. Discussion and Conclusion

Based on the findings of our computational experiments, we can deduce that the choice of method should be contingent on specific application requirements. If expediting the time of URI

**Table 3**

The average time ( $\mu$ s) to locate.

| Name         | Quanto | jPOST | RefEx | KERO   |
|--------------|--------|-------|-------|--------|
| Front Coding | 4.37   | 4.26  | 4.21  | 7.57   |
| Trie         | 61.91  | 56.05 | 38.99 | 125.70 |
| Method1      | 5.03   | 4.66  | 5.70  | 11.32  |
| Method2      | 21.59  | 18.41 | 10.16 | 72.48  |

location is of paramount importance, particularly in cases where prefix finding is unnecessary, Front Coding emerges as the preferred method. Conversely, when there is a demand for prefix finding and a preference for a lower compression ratio, Method 2 should be the method of choice. For scenarios where both prefix finding and the time of URI location are top priorities, Method 1 offers the most suitable solution.

As delineated in Section 2, URIs commonly encountered in life science datasets exhibit a distinctive pattern of limited prefix variety and extensive suffix diversity. Consequently, our proposed methods are designed with a dual-phase approach that focuses on traversing paths corresponding to the strings from the prefix perspective. In instances where a node has a substantial number of children, Method 1 resorts to binary search to locate the children, refraining from compressing the strings corresponding to them. This non-compression strategy may result in a relatively lower compression ratio. In contrast, Method 2 takes a different approach, constructing a Trie for each node with a large number of children, effectively compressing the strings related to these children, and consequently yielding the smallest size of compressed URIs.

In the context of this poster, we employed LOUDS for implementing tree structures. However, it is worth noting that the double array implementation of Trie is recognized for its speed, even though it entails larger compression sizes compared to LOUDS. An important avenue for future exploration remains a comparative analysis with Trie utilizing a double array implementation.

## 5. Acknowledgments

This work was supported by ROIS-DS-JOINT 2020 and 2021 from Research Organization of Information and Systems (ROIS) and by JSPS KAKENHI grant number 21K12148.

## References

- [1] M. A. Martínez-Prieto, J. D. Fernández, R. Cánovas, Compression of rdf dictionaries, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, 2012, pp. 340–347. doi:10.1145/2245276.2245343.
- [2] S. Kawashima, T. Katayama, H. Hatanaka, T. Kushida, T. Takagi, NBDC RDF portal: a comprehensive repository for semantic data in life sciences, Database 2018 (2018). doi:10.1093/database/bay123.
- [3] G. Jacobson, Space-efficient static trees and graphs, in: Proceedings of the 30th FOCS, FOCS 1989, 1989, pp. 549–554.