A Comparison of Data Structures to Manage URIs on the Web of Data

Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux

eXascale Infolab, University of Fribourg—Switzerland
{firstname.lastname}@unifr.ch

Abstract. Uniform Resource Identifiers (URIs) are one of the corner stones of the Web; They are also exceedingly important on the Web of data, since RDF graphs and Linked Data both heavily rely on URIs to uniquely identify and connect entities. Due to their hierarchical structure and their string serialization, sets of related URIs typically contain a high degree of redundant information and are systematically dictionary-compressed or encoded at the back-end (e.g., in the triple store). The paper represents, to the best of our knowledge, the first systematic comparison of the most common data structures used to encode URI data. We evaluate a series of data structures in term of their read/write performance and memory consumption.

1 Introduction

Uniform Resource Identifiers (URIs) are essential on the Web of data, since RDF graphs heavily rely on them to uniquely identify and connect online entities. Due to their hierarchical structure and serialization, sets of related URIs typically contain a high degree of redundant information and are very often dictionary-compressed or encoded at the back-end (e.g., in the triple store). In our own Diplodocus system [20, 18, 19], for instance, every URI is encoded as an integer number during the loading phase, and almost all subsequent operations are applied on the fixed-size, compact, and encoded version rather than on the variable-size original string. After resolving a query, though, we have to translate those ID back to their original values to display results to the client.

Working on Diplodocus, we observed that a significant part of query execution times can be consumed by encoding and decoding IDs assigned to URIS back and forth. For this reason, we present in the following and to the best of our knowledge the first systematic comparison of the most common data structures and hash functions used to encode URI data. Although related studies on data structures or hash-tables were already performed [15][1][2], they were not using the very large sets of URIs we typically operate on in the context of Semantic Web applications. Semantic Web URIs, for instance, are not standard strings, since

[1] http://attractivechaos.wordpress.com/2008/08/28/comparison-of-hash-table-libraries/

[2] http://incise.org/hash-table-benchmarks.html

they exhibit some very unique properties including longer lengths, high overlaps between related URIs, and hierarchical structures. Also, previous studies focused mostly on a few specific operations (like insertion, random updates or deletions), without giving a clear picture of the most important operations on URIs in our context (e.g., repeated look-ups or memory impact, etc.).

This paper analyzes the performance of various data structures from a pragmatic point of view. Therefore we formulate the following research question: **Which data structure performs best when encoding a URI dictionary for a triplestore?** In our analysis we take various factors into account like data size, data type (synthetic or real world data), and specific use-cases, e.g., read-mostly or read/write workloads.

The rest of this paper is structured as follows: We start by briefly reviewing the related work below in Section 2. We introduce the generic data structures and system-specific structures we benchmark in Section 3. We describe our experimental setup, the datasets we use, and our experimental results in Section 4, before concluding in Section 6.

2 Related Work

Most of the triplestores and RDF data management systems today include some component to encode the URIs appearing the RDF triples. We only cite a handful of approaches below, that are directly used in our performance evaluation hereafter. We refer the reader to recent surveys of the field (such as [11], [9], [8] or [13]) for a more comprehensive coverage of RDF systems and of the methods they use to encode data.

In RDF-3X [17], Neumann et al. use standard B+-tree to translate strings into IDs. Instead of using a similar approach to perform translations back (from IDs to literals after query processing as ended), they implement a direct mapping index [7]. This solution is tuned for id lookups, which helps them achieve a better cache-hit ratio.

Several pieces of work including [1] or [3] implement dictionary-mapping schemes. Typically, these systems implement two independent structures to handle two-way encoding/decoding (id to value, and value to id). For the value to id mapping, many approaches use disk-resident solutions. To perform the id to value mapping, approaches typically use auxiliary constant-time direct access structures.

In [14], Martinez-Prieto *et al.* describe advanced techniques for effectively building RDF dictionaries and propose a working prototype implementing their techniques. In their approach, values are grouped by the roles they play in the dataset such that all resulting encodings are organized by their position in the triples (e.g., subject, predicate, or object). Hence, the client has to specify the role of the desired piece of data when retrieving it.

3 Evaluated Methods

Reflecting on the approaches and systems described above, we decided to focus our evaluation on a set of generic data structures and to include in addition a few popular systems and approaches that were designed specifically to handle Semantic Web data. We present a series of generic data structures in Section 3.1, and a set of approaches we borrowed from Semantic Web systems in Section 3.2 below.

Our goal is primarily to analyze the performance of different paradigms (tries, hash tables, search trees) on RDF data (specifically, URIs). We compare different implementations of the same paradigm to see how the implementation might affect the performance and provide factual information to the community. We found that implementations matter: our results (see Section 5) show striking performance differences between various implementations. Our goal is not to show the superiority one given data structure, but to empirically measure and analyze the tradeoffs between different paradigms and implementations.

3.1 Generic Data Structures

We describe below the various data structures we decided to evaluate.

Hash Table (STL) [3] std::unordered_map is an unordered associative container that contains key-value pairs with unique keys. It organizes data in unsorted buckets using hashes. Hence, search, insertion and deletion all have a constant-time complexity.

Google Sparse Hash Map [4] Google Sparse Hash is a hashed, unique associative container that associates objects of type Key with objects of type Data. Although it is efficient, due to its intricate memory management it can be slower than other hash maps. An interesting feature worth mentioning is its ability to save and restore the structure to and from disk.

Google Dense Hash Map [5] google::dense_hash_map distinguishes itself from other hash-map implementations by its speed and by its ability to save and restore contents to and from disk. On the other hand, this hash-map implementation can use significantly more space than other hash-map implementations.

Hash Table (Boost) [6] this is the unordered_map version provided by the Boost library; It implements the container described in C++11, with some deviations from the standard in order to work with non-C++11 compilers and libraries.

Binary Search Tree (STL) [7] std map is a popular ordered and associative container which contains key-value pairs with unique keys. Search, removal,

[3] `http://en.cppreference.com/w/cpp/container/unordered_map`

[4] `https://code.google.com/p/sparsehash/`

[5] `https://code.google.com/p/sparsehash/`

[6] `http://www.boost.org/doc/libs/1_55_0/doc/html/unordered.html`

[7] `http://en.cppreference.com/w/cpp/container/map`

and insertion operations all have logarithmic complexity. It is implemented as a red-black tree (self-balancing binary search tree).

B+ Tree: [8] STX B+ Tree is designed as a drop-in replacement for the STL containers set, map, multiset and multimap, STX B+ Tree follows their interfaces very closely. By packing multiple value-pairs into each node of the tree, the B+ tree reduces the fragmentation of the heap and utilizes cache-lines more effectively than the standard red-black binary tree.

ART Tree: Adaptive radix tree (trie) [12] is designed to be space efficient and to solve the problem of excessive worst-case space consumption, which plagues most radix trees, by adaptively choosing compact and efficient data structures for internal nodes.

Lexicographic Tree: Lexicographic Tree is an implementation of a prefix tree, where URIs are broken based on their common parts such that every substring is stored only once. An auto-incremented identifier is stored in the leaf level. The specific implementation we benchmark was initially designed for our own Diplodocus [20, 18] system.

HAT-trie: HAT-trie [2] represents a recent combination of different data structures. It is a cache-conscious data structure which combines a trie with a hash table. It takes the idea of the burst trie and replaces linked-lists bucket containers there with cache-conscious hash tables.

3.2 Data Structures from RDF systems

We describe below the two specific URI encoding subsystems that we directly borrowed from popular Semantic Web systems.

RDF-3X: As triples may contain long strings, RDF-3X [17] adopts the approach of replacing all literals by IDs using a mapping dictionary (see, e.g., [5]) to get more efficient query processing, at the cost of maintaining two dictionary indexes. During query translation, the literals occurring in the query are translated into their dictionary IDs, which is performed using an optimized B+-tree to map strings onto IDs. For our experiments, we extracted the dictionary structure from the presented system. We also maintained the entire dictionary in main memory to avoid expensive I/O operations[9].

HDT: HDT[14] follows the last approach described above in our Related Work section; Data is stored in HDT in four dictionaries containing: i) common subjects and objects ii) subjects iii) objects and finally iv) predicates. When benchmarking this data structure, we followed exactly the same scenario as for the previous one, i.e. we extracted the dictionary structure from the system and then fitted the data in main memory. Similarly, the structure is available on our web page.

[8] `https://panthema.net/2007/stx-btree/`

[9] see `http://exascale.info/uriencoding`

4 Experimental Setup

We give below some details on the dataset, the hardware platform, and the methodology we used for our tests. Then, we present the results of our performance evaluation. All the datasets and pieces of code we used, as well as the full set of graphs that we generated from our tests, are available on our project webpage: `http://exascale.info/uriencoding`.

4.1 Datasets

We extracted URIs and literal values from well-known RDF benchmarks. To get additional insight into the various datasets, we compressed them with a standard tools (bzip2 [4]) and analyzed the structure of their URIs. Along with the descriptions of the datasets below, we present the compression ratios we obtained with bzip2 (denoted as CR), the number of levels in a radix trie (#L) built on top of each dataset, and the average number of children per level in the top-3 levels of the trie (L1, L2, L3).

DS1: 26,288,829 distinct URIs (1.6GB) were extracted from the dataset generated by the Lehigh University Benchmark (LUBM)[10] for 800 universities. LUBM is one of the oldest and most popular benchmarks for the Semantic Web. It provides an ontology describing universities together with a data generator producing well-structured datasets. [CR 42:1, #L 15, L1 7.5, L2 5.9, L3 4.9]. The URIs in this dataset are highly regular and mostly keep entities labels of around 50 classes ("Department", "University", "Professor", etc.). The entities are organized as a forest with universities as root nodes of each tree.

DS2: 64,626,232 distinct URIs (3.3GB) were extracted from the dataset generated by the DBpedia SPARQL Benchmark[16], with a scale factors of 200%. [CR 10:1, #L 59, L1 58, L2 50.8, L3 15.4]. It is a real dataset, with distinct entity names, such that there is no distinct recurring pattern in them. Properties may be strings, numbers (real and integer), dates, URIs (http, ftp) and links to other entities. Labels and properties may have a language suffix (2 character string). Properties may have a property type suffix which is a URI from a set of around 250 URIs.

DS3: 24,214,968 distinct URIs (2.1GB) were extracted from the dataset generated by the Berlin SPARQL Benchmark (BSBM)[10], with a scale factor 439,712. [CR 72:1, #L 17, L1 33, L2 14.8, L3 10]. This dataset describes entities and properties in a e-commerce use-case. The way of identifying entities is similar to LUBM. Entities have however a rich set of properties (around 50% of all elements in the dataset).

DS4: 36,776,098 distinct URIs (3.2GB) were extracted from a dataset generated by BowlognaBench[6] for 160 departments. [CR 49:1, #L 17, L1 22.5, L2 2.5, L3 1.6]. The dataset is almost fully constituted by entities labels. The way of creating these entities is similar to LUBM.

[10] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`

- **DS5:** 52,616,588 distinct URIs (3.2GB) were extracted from the dataset generated by the Lehigh University Benchmark[10] for 1,600 universities. We generated this data set to work with a larger number of elements in order to evaluate scalability. [CR 42:1, #L 15, L1 7.5, L2 5.9, L3 4.5].
- **DS6:** 229,969,855 distinct URIs (14GB) were extracted from dataset generated by the Lehigh University Benchmark[10] for 7,000 Universities. This is the biggest dataset we considered. [CR 42:1, #L 15, L1 7, L2 6.2, L3 5.8].

Due to the space limitations, only a subset of the results, mostly from DS2 and DS6, are presented below. After conducting the experiments and carefully analyzing the results we noticed that those two datasets represent the most interesting scenarios. DS6 is the biggest dataset we use and can show how data structures scale with the data size. DS2 is a real dataset and is especially interesting given the heterogeneity of its URIs (length, subpath, special characters, etc.). The full experimental results are available on our project webpage: `http://exascale.info/uriencoding`.

4.2 Experimental Platform

All experiments were run on a HP ProLiant DL385 G7 server with two Twelve-Core AMD Opteron Processor 6180 SE, 64GB of DDR3 RAM, running Linux Ubuntu 12.04.1 LTS. All data were stored on a recent 2.7TB Serial ATA disk.

4.3 Experimental Methodology

We built a custom framework for working with the various data structures, in order to measure the time taken to insert data, as well as the memory used and the look-up time. The framework covers URI-to-ID mappings and URI look-ups.

When measuring time, we retrieve the system time the process consumes to perform the operation (e.g., loading data, retrieving results) and exclude the time spent on loading data from disk in order to eliminate any I/O overhead. We also retrieve the memory consumed by the actual data by extracting the amount of resident memory used by the process.

As is typical for benchmarking database systems (e.g., for tpc-x[11]), we run all the benchmark ten times and we report the average value of the ten runs.

During our experiments, we noticed significant differences in performance when working with ordered and unordered URIs, thus we additionally tested all data structures for both of those cases. Finally, in order to avoid the artifacts created by memory swapping, we had to limit DS6 to 100M elements when benchmarking the data structures.

Figure 1 gives an overview of our test procedure for the data structures and subsystems. First, we load all URIs available from a file into an in-memory array to avoid any I/O overhead during the benchmarking process. Then, we iteratively insert and query for URIs by batches of 100k: At each step, we first measure the time it takes to load 100k URIs, and then do 100k random look-ups on the

[11] *http://www.tpc.org/*

elements inserted so far, until all URIs are inserted. In summary, we report the following for the data structures:

- total insertion time [s];
- incremental insertion time by steps of 100k inserted URIs [s];
- relative memory consumption, which is the ratio between the dictionary memory consumption and the total size of the inserted URIs;
- lookup time by steps of 100k inserted URIs [s].

As noted above, our goal is to compare the various structures from a pragmatic perspective. For each structure, we investigate its performance on bulk load (total insertion time) and on dynamically incoming data (incremental insertion time). Using the relative memory consumption, we show if the data structure performs any compression or if it introduces any space overhead. Finally, we investigate how fast it performs URI lookups w.r.t. the size of the data structure (number of URIs loaded).

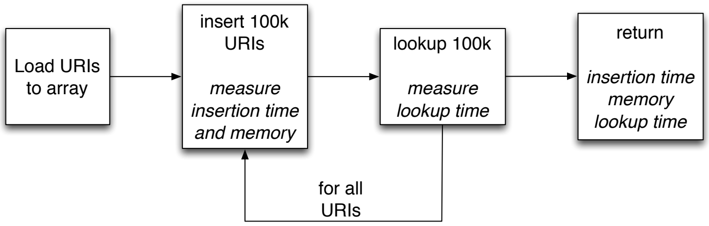


Fig. 1. Overview of our test procedure for the data structures

5 Experimental Results

5.1 Results for Generic Structures

Figures 2 and 3 show the insertion time for DS2 and DS6, respectively for a varying number of keys and for the full dataset. We observe that for synthetic data all the tree-like structures preform slightly better. As the data is more regular, it is easier to decompose URIs in that case.

We observe that for inserts, hash tables work equally well for ordered and unordered data (as they anyway hash the value before inserting it), which is not the case for other data structures. In addition, hash tables are on average faster than their alternatives. The only exception is Google Sparse Hash Map, which was 5 times slower than the other hash tables.

Tries and search trees are very sensitive to the key ordering. Shuffled datasets were taking 3-4 times more time to be inserted than the same datasets with sorted keys. On the other hand, in case of sorted datasets, they are as fast as

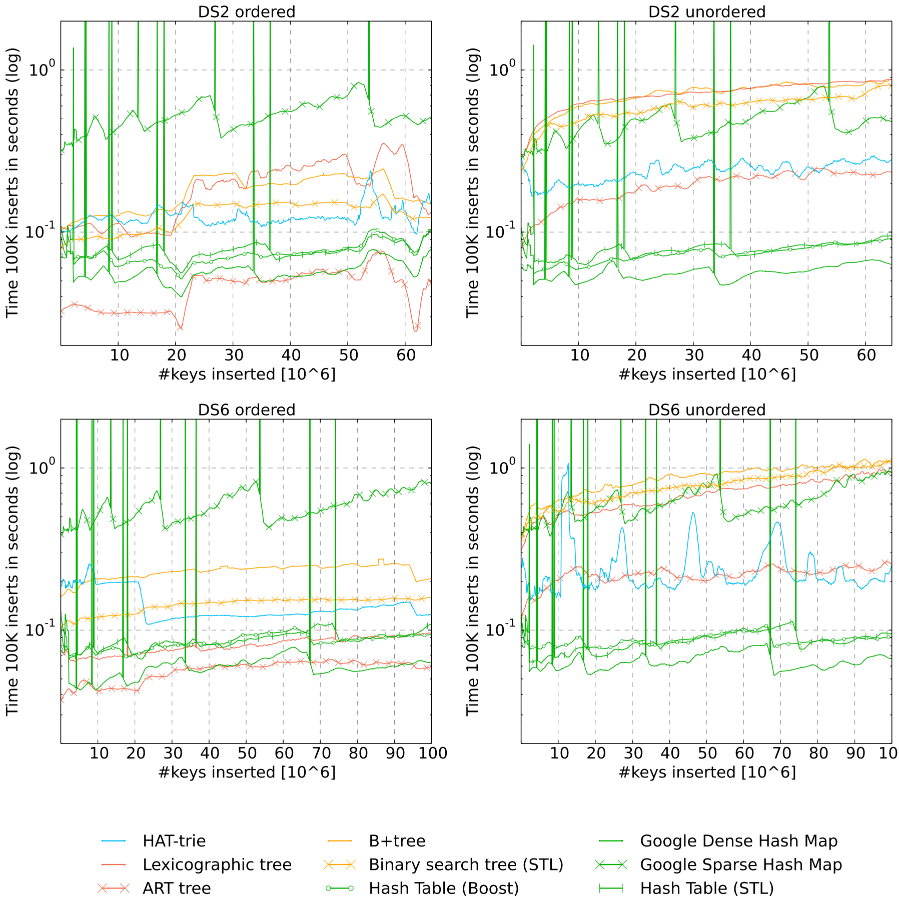


Fig. 2. Insertion time as a function of dictionary size for DS2, DS6

hash tables. ART-Tree is clearly more efficient in that context than the other data structures.

The average insert time—given as a function of the data structure size (see Figure 2)—stays nearly constant for all structures. We know that it is actually logarithmic to the size of a dictionary for tries and search trees, though the curves are reaching their flatter part quite early.

Figure 2 is also showing a very prominent drawback of hash tables: timeouts when inserting data caused by regular hash table resize (the size of the underlying structure is typically doubled every time the table is filled up to a certain percent). The timeouts might last for several seconds. The other data structures do not exhibit such a behavior.

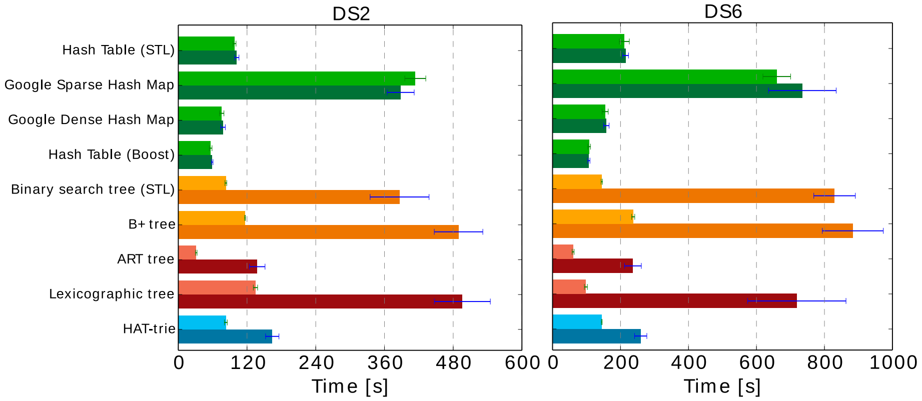


Fig. 3. Total time for fill a dictionary (DS2, DS6)

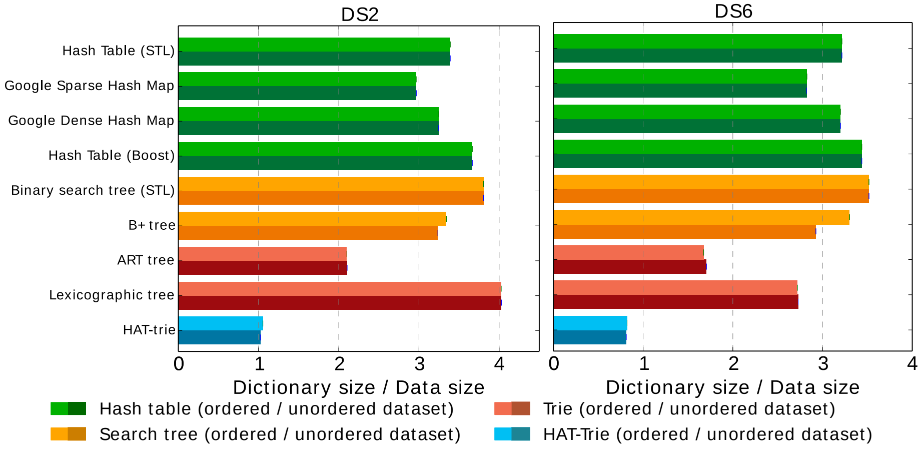


Fig. 4. Relative memory consumption (DS2, DS6)

Figure 4 shows the relative memory consumption of the data structures under consideration. Most of the structures consume 2-3 times more memory than the original datasets. However, the optimized tries (ART-tree and HAT-Trie) show outstanding results. ART-TRee consumes about 1.5x more memory than the size of DS6. HAT-Trie takes less memory than the original data (90% of DS6). So, it can actually compress the data while encoding it. We connect this feature to the fact that tries (prefix trees) can efficiently leverage the structure of rdf URIs, which are characterized by repetitive prefixes (domains and subdomains of sets of entities).

Figure 5 reports the look-up times for 100K random records after inserting 100K records incrementally for ordered and unordered datasets. As for the loading times, the regularity of the data positively influences the look-ups. Regular

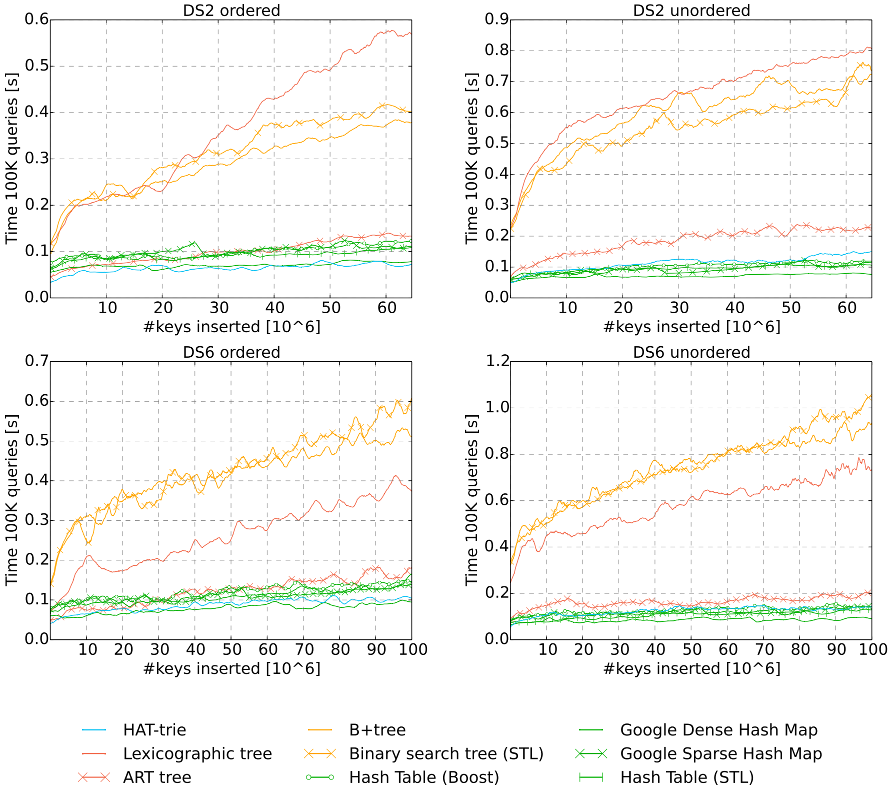


Fig. 5. Look-up times when inserting elements incrementally by 100k (DS2, DS6)

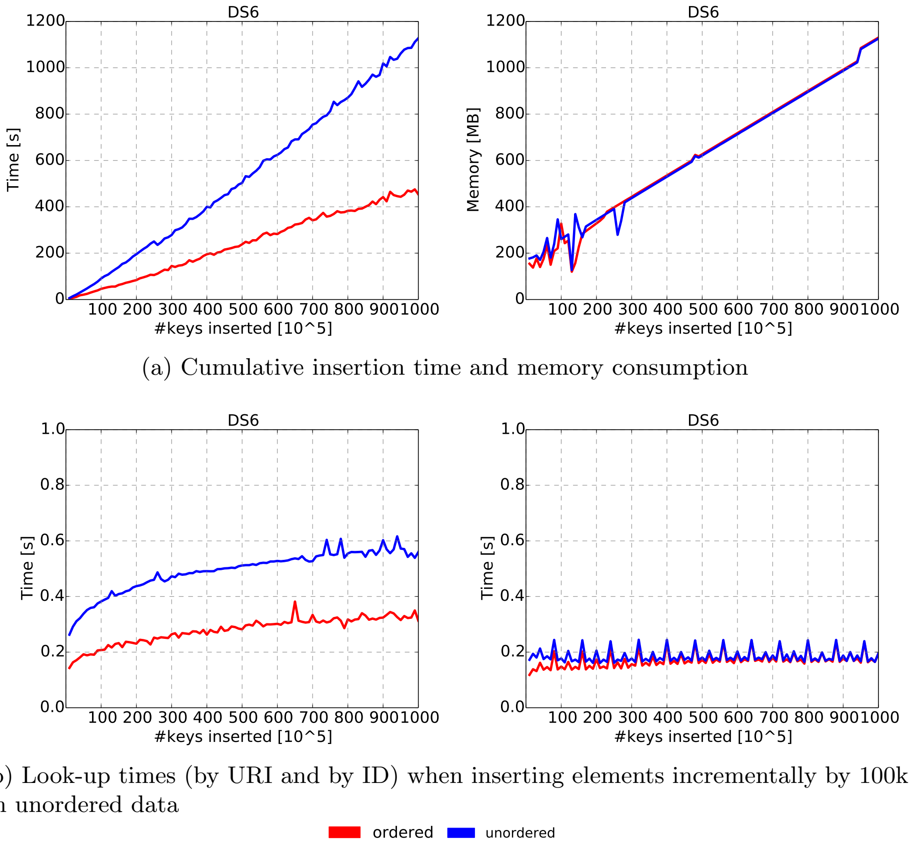
and synthetic data is easier to handle and the performance is closer to linear, especially when the URIs are ordered. We observe a strong impact on performance for the prefix and the search trees, while hash tables stay indifferent to the order in which the data is inserted. Further analyzes are done on sorted datasets only.

Search trees (B+tree and STL Map) and Lexicographic tree look-up times grow logarithmically with the size of the dictionary. In general, they are 3-6 times slower than the fastest data structures. All the others included hash tables, the HAT-Trie and ART-Trie are showing similar results, and can handle 100K queries in approximately 0.1 second regardless of the size of the dictionary.

The aforementioned features make hash tables an excellent option for dynamic or intermediate dictionaries, which are crucial for many data processing steps. They are fast in inserts and queries and do not require the keys to be sorted. For RDF archival or static dictionaries, a better option would be a modern data structure like the ART-tree or HAT-trie. They are as fast as hash tables for queries and consume much less memory (HAT-trie actually compresses the data). The sensitivity to the key's order is not crucial for a static case, since data can be pre-sorted.

5.2 Results for RDF Subsystems

Dictionary structures from RDF systems behave very differently, since they represent very polarized ways of dealing with the problem of storing triples. HDT is an in-memory compressed and complex set of structures to manage URIs. RDF-3X on the other hand represents a disk-oriented structure (that is then partially mapped into main-memory) based on B+tree.



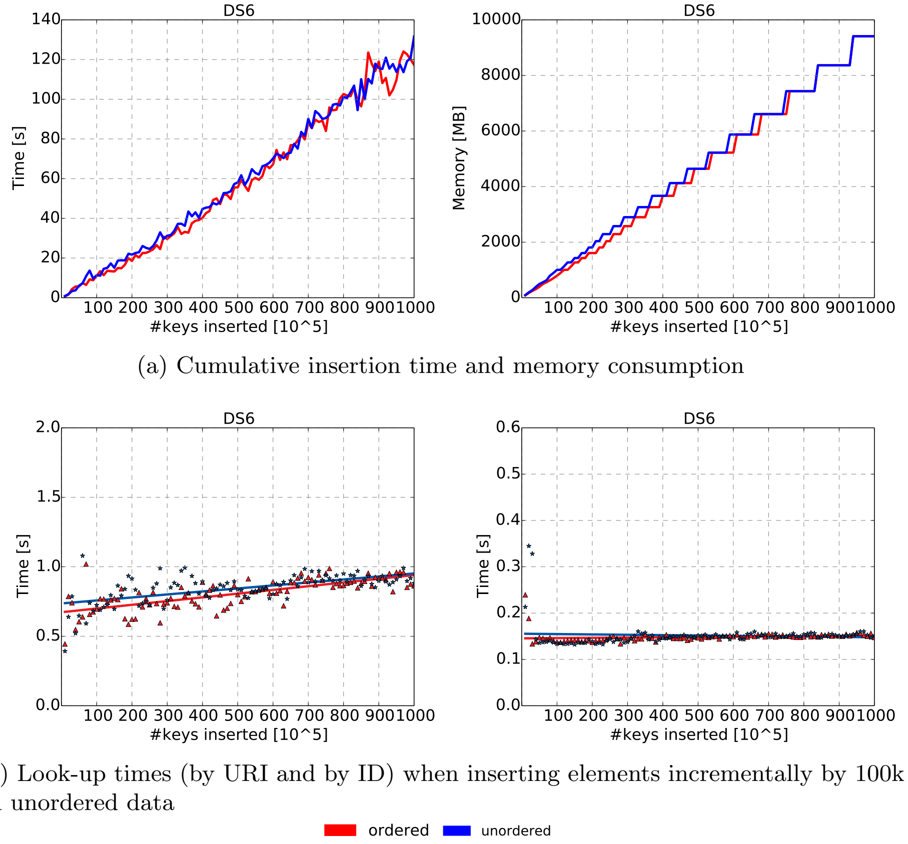
(a) Cumulative insertion time and memory consumption

(b) Look-up times (by URI and by ID) when inserting elements incrementally by 100k on unordered data

Fig. 6. Results for HDT (DS6)

Figure 6a shows the cumulative time and memory consumption during insertion for HDT. To insert all elements of DS6, it takes about 450 seconds for ordered values and 1100 seconds for unordered, consuming about 1.1GB of memory in both cases. Loading elements is linear in time. Memory consumption increases linearly also.

We benchmarked look-ups both by URI and by ID (Figure 6b). Unsurprisingly, retrieving data by string elements is more expensive than by integers, about 3 times for unordered elements. However, for ordered elements the difference is not that large, i.e., less than 2x. URIs look-ups are close to being constant in time, though we can observe some negative influence from growing amounts of data. Retrieving values by ID does not depend on the order of the elements and performs in constant time, without much influence from the data size.



(a) Cumulative insertion time and memory consumption

(b) Look-up times (by URI and by ID) when inserting elements incrementally by 100k on unordered data

Fig. 7. Results for RDF-3X (DS6)

The RDF-3X dictionary needs about 120 seconds to load all elements of DS6, consuming at the same time more than 9GB of memory (Figure 7a). The insertion costs are independent of the order of the elements in the dataset. We can also observe here a difference between look-ups by URI and ID (Figure 7b), however for RDF-3X the difference is significantly bigger; it is more that 5 times slower to retrieve string values than integers. The string look-up time here is also less sensitive to the order of values. Retrieving values by ID performs in linear time when increasing the data size.

6 Conclusions

URI encoding is an important aspect of the Web of data, as URIs are omnipresent in Semantic Web and LOD settings. Most RDF systems use their own encoding scheme, making it difficult to have a clear idea on how different methods compare in practice. In this paper, we presented, to the best of our knowledge, the first systematic comparison of the most common data structures used to manage URI data. We evaluated a series of data structures (such as sparse hash maps or lexicographic trees) and RDF subsystems in terms of their read/write performance and memory consumption. Beyond the selection of graphs presented in this paper, all the datasets and pieces of code we used, as well as the full set of graphs that we generated from our tests, are available online[12].

We make a series of observations from the results obtained through our performance evaluation:

1. Data loading times can widely vary for different index structures; Google's dense map, the Hash Tables from STL and boost, ART tree, and HAT-trie are one order of magnitude faster than Google's sparse map, Binary Search Tree, and the B+ and lexicographic trees implementations we benchmarked for reasonably big datasets
2. Data loading times for more sophisticated structures from RDF-3X or HDT are considerably slower; RDF-3X is typically one to two orders of magnitude slower than the standard data structures. HDT is even slower, as it is almost one order of magnitude worse than RDF-3x.
3. Memory consumption also exhibits dramatic differences between the structures; most of the usual data structures are in the same ballpark (differences of about 20% for big datasets), with HAT-trie significantly outperforming other generic data structures (three times less memory consumed comparing to the average). RDF-3X is also very effective in that context, requiring 30 to 40% less memory than any of the standard data structures. The clear winner in terms of resulting data size is however HDT, requiring one order of magnitude less space than the other structures (which confirms the validity of the compression mechanisms used for that project).
4. The time taken to retrieve data from the structures also vary widely; Google's dense map, ART tree, HAT-trie, and the Hash Tables from STL and boost are here also one order of magnitude faster than the other structures.
5. Look-up performance for more sophisticated structures borrowed from RDF systems are competitive; HDT is a few times slower than the best hash-tables for look-ups, while RDF-3X is around 5 to 10 times slower.

[12] http://exascale.info/uriencoding

6. Cache-aware algorithms (e.g., HAT-trie) perform better than others since they take advantage of the structure of the cache hierarchy of modern hardware architectures.
7. Finally, the order of inserted elements matters for most of the data structures. Ordered elements are typically inserted faster and look-ups are executed more efficiently, though they consume slightly more memory for the B+tree.

Overall, the HAT-trie appears to be a good comprise taking into account all aspects, i.e., memory consumption, loading time, and look-ups. ART also appears as an appealing structure, since it maintains the data in sorted order, which enables additional operations like range scans and prefix lookups, and since it still remains time and memory efficient.

We believe that the above points highlight key differences and will help the community to make more sensible choices when picking up hashes and data structures for the Web of Data. As a concrete example, we decided to change the structures used in our own Diplodocus system following those results. As we need in our context to favor fast insertions (both for ordered and unordered datasets), fast look-ups and relatively compact structures with no collision, we decided to replace our prefix tree (LexicographicTree) with the HAT-trie. We gained both in terms of memory consumption and efficient look-ups compared to our previous structure; We believe that this new choice will considerably speed-query execution times and improve the scalability of our system.

Our benchmarking framework can easily be extended to handle further data structures. In the future, we also plan to run experiments on new dataset such as Wikidata and bioinformatics use-cases.

Acknowledgement

This work was funded in part by the Swiss National Science Foundation under grant numbers PP00P2_128459 and 200021_143649.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.
2. N. Askitis and R. Sinha. Hat-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 97–105. Australian Computer Society, Inc., 2007.
3. J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic WebISWC 2002*, pages 54–68. Springer, 2002.
4. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

5. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment, 2005.
6. G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux. Bowlognabenchbenchmarking rdf analytics. In *Data-Driven Process Discovery and Analysis*, pages 82–102. Springer, 2012.
7. A. Eickler, C. A. Gerlhof, and D. Kossmann. *A performance evaluation of oid mapping techniques*. Fakultät für Mathematik und Informatik, Universität Passau, 1995.
8. Faye, O. Cure, and Blin. A survey of RDF storage approaches. *ARIMA Journal*, 15:11–35, 2012.
9. Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large owl datasets. In *In International Semantic Web Conference*, pages 274–288. Springer, 2004.
10. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.
11. B. Haslhofer, E. M. Roochi, B. Schandl, and S. Zander. Europeana RDF Store Report. In *University of Vienna, Technical Report*, 2011. http://eprints.cs.univie.ac.at/2833/1/europeana_ts_report.pdf.
12. V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.
13. B. Liu and B. Hu. An evaluation of rdf storage systems for large data applications. In *Semantics, Knowledge and Grid, 2005. SKG '05. First International Conference on*, page 59, nov. 2005.
14. M. A. Martínez-Prieto, J. D. Fernández, and R. Cánovas. Compression of rdf dictionaries. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 340–347. ACM, 2012.
15. W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.
16. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011*, pages 454–469. Springer, 2011.
17. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
18. M. Wylot, P. Cudre-Mauroux, and P. Groth. Tripleprov: Efficient processing of lineage queries in a native rdf store. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 455–466. International World Wide Web Conferences Steering Committee, 2014.
19. M. Wylot, P. Cudré-Mauroux, and P. Groth. Executing Provenance-Enabled Queries over Web Data. In *Proceedings of the 24rd International Conference on World Wide Web*, WWW '15, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.
20. M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux. dipLODocus[RDF]: short and long-tail RDF analytics for massive webs of data. In *The Semantic Web–ISWC 2011*, pages 778–793. Springer, 2011.