Research online Publishing www.ronpub.com

Open Access

Open Journal of Big Data (OJBD)
Volume 2, Issue 1, 2016
http://www.ronpub.com/ojbd
ISSN 2365-029X

Constructing Large-Scale Semantic Web Indices for the Six RDF Collation Orders

Sven Groppe [A], Dennis Heinrich [A],
Christopher Blochwitz [B], Thilo Pionteck [C]

[A] Institute of Information Systems (IFIS), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany, {groppe, heinrich}@ifis.uni-luebeck.de
[B] Institute of Computer Engineering (ITI), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany, blochwitz@iti.uni-luebeck.de
[C] Institut für Informations- und Kommunikationstechnik (IIKT), Otto-von-Guericke-Universität Magdeburg, Universitätsplatz 2, D-39106 Magdeburg, Germany, thilo.pionteck@ovgu.de

Abstract

The Semantic Web community collects masses of valuable and publicly available RDF data in order to drive the success story of the Semantic Web. Efficient processing of these datasets requires their indexing. Semantic Web indices make use of the simple data model of RDF: The basic concept of RDF is the triple, which hence has only 6 different collation orders. On the one hand having 6 collation orders indexed fast merge joins (consuming the sorted input of the indices) can be applied as much as possible during query processing. On the other hand constructing the indices for 6 different collation orders is very time-consuming for large-scale datasets. Hence the focus of this paper is the efficient Semantic Web index construction for large-scale datasets on today's multi-core computers. We complete our discussion with a comprehensive performance evaluation, where our approach efficiently constructs the indices of over 1 billion triples of real world data.

Type of Paper and Keywords

Regular research paper: *Semantic Web, RDF, index construction, external sorting, string sorting, patricia trie*

1 Introduction

In order to realize the vision of the Semantic Web [38], the World Wide Web Consortium (W3C) recommends a number of standards. Among them are recommendations for the data model Resource Description Framework (RDF) [39] of the Semantic Web and the ontology languages RDF Schema (RDFS) [9] and OWL [29]. Ontologies serve as schemas of RDF and contain implicit knowledge for accompanying datasets. Hence using common ontologies enable interoperability between heterogeneous datasets, but also proprietary ontologies support the integration of these datasets based on their contained implicit knowledge. Indeed one of the design goals of the Semantic Web is to work in heterogeneous Big Data environments. Furthermore, the Semantic Web community and especially those organized in the Linking Open Data (LOD) project [22] maintain a Big Data collection of freely available and accessible large-scale datasets (currently containing approximately 150 billion triples in over 2,800 datasets [23, 24]) as well as links (of equivalent entities) between these datasets.

Douglas Laney [21] coined the 3 V's characteristics of Big Data: Volume, Velocity and Variety[1]. Consider-

[1] Later the community proposed an increasing number of V's (e.g. [36] for 7 V's and [8] for the top 10 list of V's).

ing LOD datasets and the Semantic Web technologies, variety of data is basically dealt with the support of ontologies. The velocity of data can be only indirectly seen in the ongoing increasing growth of the dataset sizes in LOD, which are snapshots and are not typically processed in real-time. Hence there is a need for speeding up the processing of these Semantic Web datasets addressing especially the volume characteristics.

Some (but not all) of the LOD datasets are also available via SPARQL endpoints which can be queried remotely [23]. However, also for these valuable large-scale datasets (as well - of course - for those datasets which are only available as dumps) it is often desirable to import them to local databases because of performance and availability reasons. An incremental update to databases (inserting one triple after the other) will be too time-consuming for importing large datasets and hence will be impractical. Instead, it is essential to construct indices from scratch in an efficient way. Efficient index construction approaches are also needed in many other situations, e.g. whenever databases need to be set up from archived and previously made dumps, because of e.g. a recovery (based on a backup) after a hardware crash or reconfigurations of the underlying hardware for upgrading purposes. Hence we propose an efficient approach to speed up the index construction for large Semantic Web datasets, which is one of the keys for paving the way for the success of Semantic Web databases in the Big Data age.

Today's efficient Semantic Web databases [37, 27, 28, 14] use a dictionary to map the string representations of RDF terms to unique integer ids. Using these integer ids instead of space-consuming strings, the indices of those Semantic Web databases do not waste much space on external storage like SSDs and hard disks. Furthermore and more important, processing integer ids instead of strings is more efficient during atomic query processing operations like comparisons and lowers the memory footprint such that more intermediate results can be hold in memory and less data must be swapped to external storage especially for Big Data.

Semantic Web databases like [37, 27, 28, 14] further use 6 indices (called *evaluation indices*) according to the 6 collation orders of RDF triples. This allows a fast access to the data for any triple pattern having the result sorted as required by fast database operations like merge joins.

Overall this type of Semantic Web databases hence needs to construct the dictionary and 6 evaluation indices, which is very time-consuming. Our idea is to use a very fast algorithm to construct the dictionary, but to also smoothly incorporate the construction of the 6 evaluation indices into the generation of the dictionary. Furthermore, we want to use the parallelism capabilities of today's multi-core CPUs to further speed up the index construction.

The main contributions of this paper include:

- a new index construction approach for Semantic Web data smoothly incorporating dictionary construction and evaluation indices construction by taking advantage of parallelism capabilities offered by today's multi-core CPUs, and
- a comprehensive performance evaluation and analysis of our proposed index construction approach using large-scale real-world datasets with over 1 billion triples.

2 Basic Data Structures, Indices and Sorting Algorithms

In this section we shed light on the foundations of the Semantic Web, its data model and its widely used index approach. Moreover, we discuss the most important family of external sorting algorithms, the *external merge sort* algorithm. Furthermore, we shortly introduce patricia tries which are extensively used by our new Semantic Web index construction approach.

2.1 B^+-Trees

The most widely used database index structures are the B^+-trees [11], which are search trees with self-balancing capabilities and optimized for block-oriented external storage. In comparison to B-trees [5], the B^+-tree stores all records in the leafs and its interior nodes hold only keys. In this way the interior nodes maintain more keys lowering the height of the overall search tree. The database systems often build indices efficiently by sorting the input data, which avoids expensive node splitting (see [25] and extend its results to B^+-trees, or see [14]). Index construction by sorting is typically much faster than processing unsorted data. Hence one of the basic steps of index construction is sorting.

2.2 Semantic Web and Indices

The current World Wide Web aims at the humans as end user: Humans easily understand text in natural language, consider implicit knowledge and detect hidden relationships. The vision of the *Semantic Web* [38] is a machine-processable web [6] for the purpose of new applications for its users. By explicitly structuring (web) information the Semantic Web achieves simplification of automatic processing. In order to push the idea of the Semantic Web, in the recent years the Semantic Web initiative of the *World Wide Web Consortium (W3C)* specified a family of technologies and language standards like

```
@prefix rdf:  <http://www.w3.org/1999/02/22rdfsyntaxns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdfschema#> .
@prefix v:    <http://www.ifis.uni-luebeck.de/vocabulary/> .
@prefix i:    <http://www.ifis.uni-luebeck.de/instances/> .
v:Journal        rdfs:subClassOf v:BibEntity .
v:Article        rdfs:subClassOf v:BibEntity .
i:OJBD           rdf:type        v:Journal .
i:OJBD           v:title         "Open Journal of Big Data"^^xsd:string .
i:Article1       rdf:type        v:Article .
i:Article1       v:title         "Solving Big Problems"@en .
i:Article1       v:publishedIn   i:OJBD .
i:Article1       v:creator       i:Author_BigData .
i:Author_BigData v:name          "Big Data Expert"^^xsd:string .
```

Listing 1: Example of RDF data

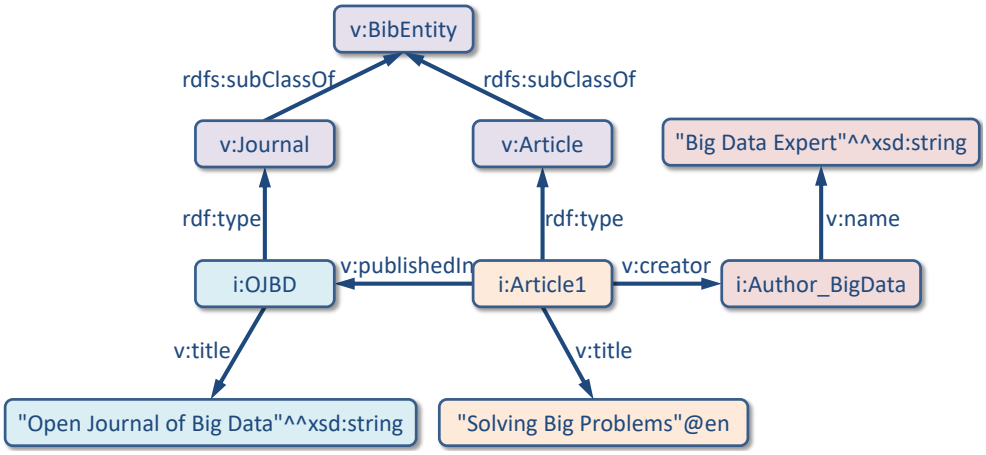

Figure 1: RDF graph of the RDF data of Listing 1

the *Resource Description Framework (RDF)* [39] for describing Semantic Web data. We introduce RDF in the following section.

2.2.1 Resource Description Framework (RDF)

Originally designed to describe (web) resources, the *Resource Description Framework (RDF)* [39] can be used to model any information as a set of triples. Following the grammar of a simple sentence in natural language, the first component s of a triple $(s,\ p,\ o)$ is called the subject, p is called the predicate and o the object. More formally:

Definition (RDF triple): Assume there are pairwise disjoint infinite sets I, B and L, where I represents the set of IRIs, B the set of blank nodes and L the set of literals. We call a triple $(s,p,o) \in (I \cup B) \times I \times (I \cup B \cup L)$ an RDF triple, where s represents the subject, p the predicate and o the object of the RDF triple. We call an element of $I \cup B \cup L$ an RDF term.

RDF data is typically visualized in *RDF graphs* by drawing a directed labeled edge from a subject node to an object node for each triple, where the label of the edge is the predicate and common nodes are collapsed into one unique node.

Listing 1 contains an example of RDF data and Figure 1 the corresponding RDF graph describing an article in the OJBD journal.

2.2.2 Dictionary

Mapping RDF terms to integer ids lowers space requirements in the evaluation indices storing the input RDF triples each of which with three integers instead of possibly large strings. Using difference encoding [27] and avoiding to store leading zero bytes additionally saves

ID	RDF term
0	i:Article1
1	i:Author_BigData
2	i:OJBD
3	rdfs:subClassOf
4	rdf:type
5	v:creator
6	v:name
7	v:publishedIn
8	v:title
9	v:Article
10	v:BibEntity
11	v:Journal
12	"Big Data Expert"^^xsd:string
13	"Open Journal of Big Data"^^xsd:string
14	"Solving Big Problems"@en

Table 1: Possible dictionary for the RDF terms in Listing 1

```
11  3 10 .
 9  3 10 .
 2  4 11 .
 2  8 13 .
 0  4  9 .
 0  8 14 .
 0  7  2 .
 0  5  1 .
 1  6 12 .
```

Listing 2: ID triples of Listing 1 according to the dictionary in Table 1

space. Furthermore, using ids enables space-efficient representations of (intermediate) solutions lowering the memory footprint: more solutions can be processed before swapping to hard disks/SSDs starts increasing the overall performance. For example, Listing 2 contains the ID triples of Listing 1 according to the dictionary in Table 1.

On the other hand using ids causes high costs for the materializations of the RDF terms for (more seldom) operations like sorting or relational comparisons like $<$, $\leq$, $\geq$ and $>$, because these operations require the string representations of the RDF terms and not the ids. Whenever the query result is large, displaying the final *textual* query result is also a costly operation. However, especially for large-scale datasets, the advantages typically outweigh the disadvantages of using ids.

Hence, many Semantic Web query evaluators such as RDF3X [27, 28] and Hexastore [37] as well as LUPOSDATE [14] use dictionary indices to map RDF terms into integer ids.

A dictionary needs two indices mapping RDF terms into integer ids and vice versa. In Big Data scenarios the dictionary indices typically do not fit into main memory. LUPOSDATE uses a B$^+$-tree for storing the mapping of RDF terms into integer ids (and hence the key of the B$^+$-tree is the string representation and the value is the integer id). The other mapping direction is maintained in a file-based array of pointers addressing the strings of RDF terms in a second file: For looking up the string representation of an id the position of the pointer in the first file is calculated by multiplying the id value with the pointer size. Then the string can be directly accessed in the second file according to the retrieved pointer. Hence only two disk accesses are necessary for each lookup.

2.2.3 Evaluation Indices

Hexastore [37], RDF3X [27, 28] and LUPOSDATE [14] maintain indices for the six collation orders SPO, SOP, PSO, POS, OSP and OPS of RDF triples. For example, the collation order SPO describes the order, where the subjects (S) of triples are the primary order criterion, the predicates (P) the secondary, and the objects (O) the tertiary order criterion. In this way a prefix search in the right index can directly deliver the results of a triple pattern in any order: For example, any triple pattern containing RDF terms in the subject and the object position and a variable in the predicate position can be answered by one index access performing a prefix search in the SOP index having the subject and object as prefix key. Furthermore, the result is ordered according to the predicate. Supporting all 6 possible collation orders, fast merge joins over the retrieved sorted sorted data for several triple patterns can be most often applied. For every collation order, for example, SPO, [37] proposes to associate a subject key s_i to a sorted vector of n_i property keys, $\{p_1^i, p_2^i, ..., p_{n_i}^i\}$. Each property key p_j^i is, in its turn, linked to an associated sorted list of $k_{i,j}$ object keys. These object lists (e.g. for SPO) are shared in indices for corresponding collation orders (e.g., PSO). RDF3X [27, 28] and LUPOSDATE [14] just use B$^+$-trees as index structures for the different collation orders, which is a simpler, faster, and hence more elegant approach than [37]. For example, Figure 2 pictures a B$^+$-tree containing ID triples of Listing 2 according to the SPO collation order.

RDF3X and LUPOSDATE use sophisticated data structures and difference encoding to compress their index structures by storing only the different components of a triple in comparison to the last stored triple. Also only the difference of the remaining components to the corresponding ones of the previous triple are stored,

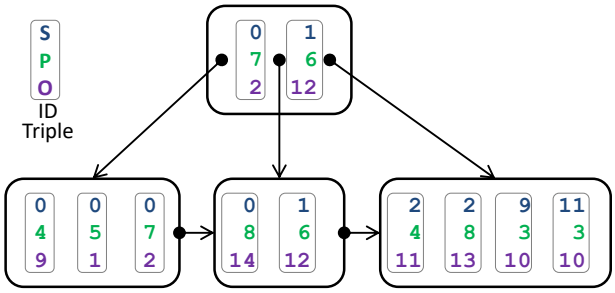

Figure 2: B$^+$-tree containing ID triples of Listing 2 according to the SPO collation order

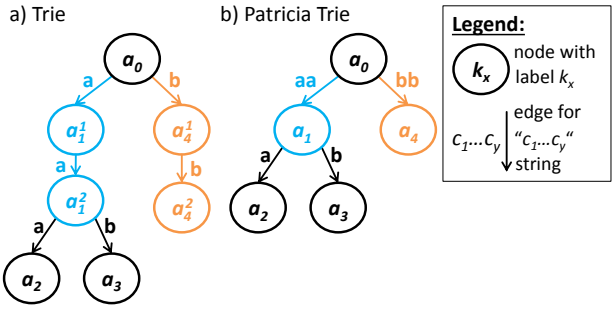

Figure 3: a) Trie and an equivalent b) patricia trie containing the strings "aaa", "aab" and "bb"

which reduces the number of bits necessary for the representation of these components. RDF3X further supports additional special aggregated indices for fast processing of special kinds of queries, which occur only seldom, but coming with the cost of maintaining additional indices.

In this paper we consider the construction of evaluation indices according to LUPOSDATE. However, our approach can be easily modified to construct also Hexastore or RDF3X indices by exchanging the last step in our approach.

2.3 Heap

The smallest item from a collection can be efficiently retrieved by using a (min-) heap (see [26]), which supports inserting as well as removing an item in logarithmic time (in comparison to the items stored in the heap). The internal organization of the heap is a tree, most often a complete binary tree memory-efficiently stored in an array. The heap condition requires the root of each subtree to contain the smallest item of the subtree. Hence adding an item inserts the item as leaf to the heap tree and performs a *bubble-up* operation, which swaps the item with its parent as long as it is smaller than its parent. For removing the smallest item from the root of the heap, the item in the most-right leaf of the bottom level is moved to the free space in the root. Afterwards in order to reestablish the heap condition, during a *bubble-down* operation the root item is recursively swapped with its minimum child if the minimum child is smaller than it.

We can optimize a pair of remove- and insert-operations and avoid one bubble-up operation by just inserting the new item in the root and then performing a bubble-down operation. We use this improvement during merging the runs (see Section 3.9).

2.4 (External) Merge Sort

External sorting approaches sort data not fitting into the main memory. One of the most widely used external sorting approaches is (external) merge sort (see [20]), which first generates initial runs of already sorted data. By merging the initial runs a new round of runs is generated until only one run containing all the data to be sorted remains.

In contrast to binary merge phases, merging many runs at once is more efficient. During this n-ary merge phase, the minimum of these n open runs needs to be determined. Recalling the described data structure of the previous subsection, a heap is the ideal data structure for this purpose.

Initial runs are usually generated by reading as much data into main memory as still fit inside, sorting this data in main memory and swapping the sorted data as run to external storage. Fast main memory sort algorithms like quicksort, (main memory) merge sort (and its parallel version) and heapsort are typically chosen for sorting the data in main memory [14].

In this paper, we propose a kind of specialized external merge sort algorithm for constructing the 6 evaluation indices for the 6 collation orders of RDF, which is smoothly integrated into the dictionary construction utilizing PatTrieSort [17] to be described in Section 2.6.

2.5 Patricia Tries

Tries (e.g., [1]) avoid to store common prefixes of their contained strings (e.g., see Figure 3 a)) by maintaining a special tree structure. Each edge in this tree is labeled with one character, and the concatenation of the characters along the path from the root to the leaf form the stored strings in the trie. For efficient storage and access to the stored strings, duplicates in the edge labels of a node are not allowed, and the edges are lexicographically ordered according to their labels.

Patricia tries are a compressed variant (see Figure 3), where all trie nodes (except of the root node) with only one child are melt together with their single child (and the edge between them is removed). The label of the incoming edge of the new node holds the concatenation

of its previous label and the label of the old edge to the child. For efficient search and update operations, the edges are sorted according to the lexicographical order of their labels. In contrast to tries, the label in a patricia trie can be an empty string (denoted by $\emptyset$), which occurs if the patricia trie contains a substring of another one.

Semantic Web data typically consist of many strings with common prefix, as often IRIs [13] are used. Thus, patricia tries are the ideal data structure to store Semantic Web data in main memory.

2.6 PatTrieSort

In [17], we use patricia tries for initial run generation in an external merge sort variant for strings called *PatTrieSort*, such that initial runs - because of the compact representation of strings in patricia tries - are usually larger compared to traditional external merge sort approaches consuming the same main memory size. Furthermore, PatTrieSort stores the initial runs as patricia tries instead of lists of sorted strings, as patricia tries can be efficiently merged in a streaming fashion having a superior performance in comparison to merging runs of sorted strings.

We propose to use PatTrieSort for sorting the RDF terms as preparing step for generating the dictionary. We further propose to smoothly integrate mapping the RDF terms of triples to ids and sorting the triples according to the six collation orders of RDF in PatTrieSort in order to speed up the overall index construction.

2.7 Further Related Work

While [30, 12] introduce basic sorting algorithms in more detail, [35, 3] are appropriate as surveys on external string sorting.

Some contributions utilize tries already for sorting (e.g., burstsort and its variants [34, 33]). In burstsort, a trie is dynamically constructed as strings are sorted, and is used to allocate a string to a bucket. For full buckets new nodes of the trie are constructed the leafs of which are again buckets. However, these algorithms work only in main memory for the purpose of lowering the rate of cache miss and are not developed for external sorting.

The main idea (and conclusion) of [40] is that it is faster to compress the data, sort it, and then decompress it than to sort the uncompressed data. This approach reduces disk and transfer costs, and, in the case of external sorts, cuts merge costs by reducing the number of runs. The authors of [40] propose a trie-based structure for constructing a coding table for the strings to be sorted. In comparison, we do not use codes, but we also store *compressed runs* by storing the patricia trie containing all the entries of the run, which reduces the space on disk and in memory, too.

The contributions in [4] lay the foundations for a complexity analysis for I/O costs for the string sorting problem in external memory. Its contribution covers the discussion of optimal bounds for this problem under different variants of the I/O comparison model, which allow or not allow strings to be divided in single characters in main memory and/or on disk.

In [16], we already propose approaches to construct indices for the 6 collation orders of RDF. However, the proposed approaches in [16] do not construct a dictionary and the triples are stored with the string representations of RDF terms instead of ids.

3 Constructing Indices according to 6 RDF Collation Orders

We focus on constructing indices for Semantic Web databases supporting dictionaries for mapping RDF terms to integer identifiers (in order to lower space requirements and memory footprint resulting in higher performance for most query types) and retrieving presorted data according to the six RDF collation orders (in order to support as many merge joins as possible). The main tasks for index construction are hence a) dictionary construction, b) mapping the RDF triples to id triples using the integer ids of the dictionary and c) constructing the evaluation indices according to the six different collation orders of RDF. The naive way is to separate these three main tasks in different phases during index construction, and (even worse) to separate the construction of the six evaluation indices into 6 different sub-phases.

In contrast to this naive way, we propose to smoothly integrate all these tasks in order to avoid unnecessary I/O workload and computations.

We propose to apply a sophisticated process of 9 steps, which we describe in the following subsections in more detail. Figure 4 contains an overview of the overall process including an example.

3.1 Building Patricia Tries and Mapping Triples to Temporary IDs

According to [17] utilizing patricia tries for sorting the RDF terms of large-scale datasets is highly efficient. In more detail in [17] we propose the PatTrieSort approach, which constructs patricia tries in main memory, rolls the full patricia tries out into external storage (but storing them as patricia tries) and finally merges them by a merge algorithm specialized to patricia tries.

Our main idea is to smoothly integrate the remaining tasks into PatTrieSort: the mapping of the RDF triples to

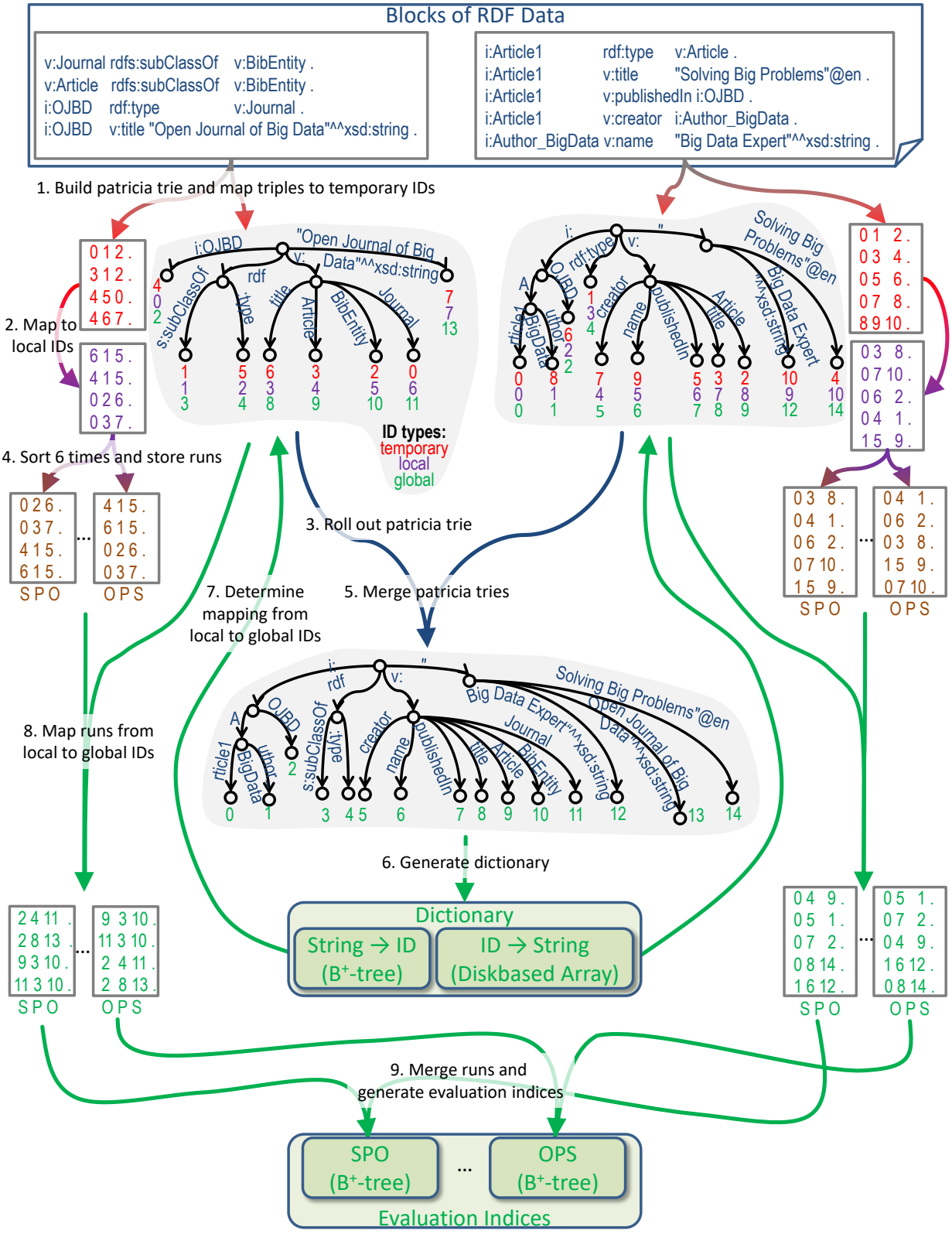

Figure 4: Overview of index construction process with example

id triples and sorting them according to the six collation orders. We also want to save as much memory space as possible, such that more triples can be processed block-wise in main memory leading to larger runs of sorted id triples stored in external storage. Hence we propose to use id triples as early as possible: After reading a triple according to PatTrieSort we first add the string representations of its RDF terms (i.e., subject, predicate and object) into the main-memory patricia trie (see step 1 in Figure 4).

In order to avoid several reads of the RDF data, we propose to hold also the triples in main memory, sort them before the main-memory patricia trie is rolled out according to the six collation order and finally write these sorted runs according to the six collation orders to external storage. However, storing the string representations of the RDF terms for each triple is too space-consuming, which abolishes the advantage of PatTrieSort of space-efficiently storing strings. Hence we propose to transform the RDF triple to an id triple right after reading a triple, which greatly saves main-memory by storing only 3 integers instead of 3 strings for each triple. However, right after reading a triple we have not the ids of its RDF terms (because the dictionary has not been built so far). Hence we propose to map the RDF terms first to a temporary id, which we maintain in the currently constructed patricia trie acting as main-memory key-value store with the RDF terms as keys and the temporary ids as values: We just check if the RDF term to be mapped is already included in the currently constructed patricia trie. In the case that the RDF term is included we use its previously assigned temporary id. In the other case we assign a new temporary id (corresponding to the current number of entries in the patricia trie) for the RDF term and store this temporary id at the leaf of the RDF term in the patricia trie.

3.2 Mapping to Local IDs

Once the current block of triples is completely read in, our approach maps the temporary ids (built according to the occurrences of the RDF terms) of the loaded triples to local ids (built according to the lexical order of the RDF terms) (see step 2 in Figure 4). For this purpose, we determine a mapping of the temporary ids to the local ids by just constructing a one-dimensional array, where the index in the array corresponds to the temporary id and the array value at the index to the local id (which corresponds to the position in the sorted sequence of RDF terms). This one-dimensional array can be constructed during one in-order traverse through the patricia trie of RDF terms.

In one of the following steps, we want to already generate initial runs after sorting the loaded triples, which are later (mapped to global ids and) merged for determining a complete sorted sequence of triples. For this purpose, the relative order between local ids and global ids must be the same, i.e., if a local id id_1 is smaller than another one id_2 ($id_1 < id_2$), then the same order-relation must hold for the corresponding global ids ($global(id_1) < global(id_2)$ with $global$ is a mapping from local to global ids). The simplest way is to use the lexical order of the original RDF terms (which is always implicitly given and can be determined in different blocks of triples independently from each other), although in general it is only important that the dictionary is constructed according to any order. Indeed this initial order of ids according to the lexical order of the original RDF terms will be destroyed after updates on the constructed indices, which typically introduce new ids without considering the lexical order of original RDF terms in order to avoid a costly renumbering of the old ids.

3.3 Rolling Out Patricia Trie

Before we sort the triples with local ids according to the six collation orders of RDF, which consumes 6 times more main memory for maintaining the triples, we roll out the current patricia trie to free up main memory (see step 3 in Figure 4). It is important that the patricia trie is swapped to external memory (like harddisk or SSD) in a format, where the structure of the patricia tries remains, and the nodes of the patricia trie are stored by a left-order traversal through the patricia trie. In this way the patricia trie does not need to be constructed again and can be directly reused (in the later step for mapping the local ids to global ones). Furthermore, besides having a very compact representation of the contained strings, the patricia trie in this form can be processed in a streaming fashion [17], which is especially important for a later merging of all the patricia tries for generating the dictionary.

3.4 Sorting Runs of Triples Cccording to 6 Collation Orders

In this 4th step (see Figure 4) the loaded triples with local ids are sorted according to the 6 collation orders. Our idea is to use the properties of RDF and of the local id triples for further improving the processing speed. We observe the following properties:

- We have to sort triples composed of integers with a limited, relatively small domain (from 0 to $n - 1$, where n is the number of distinct RDF terms in the current block of RDF triples, i.e. the number of contained RDF terms in the previously rolled out patricia trie).

- The 6 collation orders of RDF have 3 primary collation order criteria (subject, predicate and object), each of which having 2 secondary collation order criteria (subject, predicate and object without the primary collation order criterion).

Among the sorting algorithms Counting Sort [20, 31] is a specialized linear runtime algorithm working on keys with small domain. Hence we propose to use Counting Sort for sorting the local ids triples according to the 3 primary collation orders (subject, predicate and object). Counting Sort has another advantage: It already determines the borders of blocks of triples with the same primary key in the sorted output. Hence we can use these borders to sort blocks of triples with the same primary key according the secondary and tertiary keys with a fast standard sorting algorithm like quicksort [19].

We can easily parallelize sorting by

- dealing with the 3 primary collation orders in parallel, and
- sorting the blocks of triples with the same primary key in parallel.

Afterwards the 6 different runs for the six collation orders can be swapped to external memory (like harddisk or SSD). Inspired by [27, 28] we use difference encoding in order to compress the data for saving space and increasing the speed of I/O operations. Difference encoding does not store common components of the current triple compared to the previous one. Furthermore, for the different components of the triple, we only store the difference to the previous triple, which is a smaller number. We also only store as many bytes of an integer id (its difference to the previous triple respectively) as necessary (avoiding to store leading zeros). This lightweight compression scheme is fast to compute, but saves many I/O operations, which overall hence saves also computation costs. All the other blocks of RDF data are now processed in the same way (steps 1 to 4 in Figure 4).

3.5 Merging Patricia Tries

In order to construct a uniform mapping from RDF terms (in form of strings) to global ids (in form of integers) and vice versa (i.e., the dictionary), we need to merge all the generated patricia tries (see step 5 in Figure 4). We propose to use the merge algorithm of [17] working directly on patricia tries, being very efficient and having some extraordinary advantages: According to [17], the merge algorithm for patricia tries can have an arbitrary number of patricia tries as input, reads and processes all these input patricia tries by a left-order traversal, and stores the resultant merged patricia trie again in a left-order traversal. Hence the merge algorithm can merge as many patricia tries at once, as many nodes of patricia tries can be intermediately held in main memory. Typically there is only one merging step necessary even for huge datasets to be sorted, which further improves the speed of the overall algorithm.

3.6 Generating Dictionary

After the patricia tries have been merged, the dictionary can be generated (see step 6 in Figure 4). The dictionary consists of two indices: The first index is a B^+-tree for the mapping of the RDF terms (in form of strings) to the global ids (in form of integer values). The second index maintains the other mapping direction from the global ids to the RDF terms. For fast access, this index is stored in a file-based array of pointers addressing the strings of RDF terms in another file. In this way there are only two disk accesses necessary for retrieving the RDF term of a global id: We can look up the RDF term in the second file at the position addressed by the pointer stored at the position calculated by multiplying the global id value with the pointer size in the first file.

Both indices, the B^+-tree (see [25] and extend its results to B^+-trees, or see [14]) as well as the file-based array can be generated by iterating one time through the sorted sequence of RDF terms.

Generating the dictionary indices can be parallelized by generating the two indices in parallel.

3.7 Determining Mapping from Local to Global IDs

After the dictionary has been generated the next step is to determine a mapping from the local ids of the runs to the global ids of the dictionary (see step 7 in Figure 4). As dictionary lookups are expensive especially for masses of lookups, we want to avoid dictionary lookups as much as possible. For this reason, we load the rolled out patricia tries for each run and traverse one time through the patricia tries. For each element in a considered patricia trie, we look up one time in the dictionary to retrieve its global id. At the same time we build the mapping by using an one-dimensional array, where the index corresponds to current position in the patricia trie (which is the local id) and we set its value to the retrieved global id. Note that the one-dimensional array has a range from 0 to $n - 1$, where n is the number of elements contained in the patricia trie. It is hence a very compact representation of the mapping. After traversing the patricia trie, we can free up its resources as we will only work with the mapping (stored in the one-dimensional array) and do need the patricia trie of the run any more.

3.8 Mapping Runs from Local to Global IDs

In the 8th step of Figure 4 we use the mapping (in form of an one-dimensional array) determined in the previous step to go through the corresponding run and replace all local ids with their global ones by looking up the mentioned one-dimensional array at the index position of the local id. As not all mappings and runs fit into main memory, we again swap the runs (this time containing global ids) to external memory.

3.9 Merging Runs and Generating Evaluation Indices

In the last step and for each collation order of RDF, we merge all its runs containing global ids (see step 9 in Figure 4). As for large-scale datasets many runs must be merged, we use a heap of the current triples of each run (i.e., the heap has the size of the number of runs). Determining the next smallest triple can be done in logarithmic time (in relation to the number of runs) when using a heap in comparison to a linear time for a linear search through all current triples of the runs. During merging, we always remove the smallest triple of the remaining ones located in the root of the heap and insert the next triple of that run, which is the same as the one of the removed triple. We optimize this pair of removing and insertion operations by first placing the new triple in the root and then performing a bubble-down operation in the heap, which avoids one bubble-up operation.

For each collation order of RDF while merging, we can build the evaluation index (in form of a B^+-tree) in one pass through the corresponding final merged run (see [25] and extend its results to B^+-trees, or see [14]).

We can parallelize this step by merging and generating the evaluation indices in parallel for each of the six collation orders.

4 Experimental Analysis

We compare the runtime of our proposed index construction approach for different parameters. The implementation of the index construction approach is open source and publicly available as part of the LUPOSDATE project [14, 15]. We describe the configuration of the test system in Section 4.1, the used dataset in Section 4.2 and the experimental results and analysis in Section 4.3.

4.1 Configuration of the Test System

The test system for the performance analysis uses an Intel Xeon X5550 2 Quad CPU computer, each with 2.66 Gigahertz, 72 Gigabytes main memory, Windows 7 (64 bit) and Java 1.8. We have used a hard disk for reading in the input data and a 500 GBytes SSD for writing out the runs. The input data is read and parsed asynchronously (by 8 threads) using a bounded buffer. For saving space, we compressed the input data by using BZIP2 [32]. Decompression is done on-the-fly during reading in the input data. We have run the experiments ten times and present the average execution times.

4.2 Billion Triples Challenge

The overall objective of the Semantic Web challenge is to apply Semantic Web techniques in building online end-user applications that integrate, combine and deduce information needed to assist users in performing tasks [18]. For this purpose, in last years large-scale datasets were crawled from online sources which are used by researchers to showcase their work and compete with each other. The *Billion Triples Challenge* (BTC) dataset of 2012 [2] consists of 1 436 545 545 triples crawled from different sources like Datahub, DBpedia, Freebase, Rest and Timbl. BTC is perfectly suited as example for large-scale datasets consisting of real world data with varying quality and containing noisy data.

Although we have used input data with over 1 billion entries, we only use one merging step for merging the patricia tries.

4.3 Results

Figure 5 contains the total construction times for importing the whole BTC dataset of 2012 [2]. As explained in Section 3, the RDF triples are block-wise processed during index construction. We have varied the sizes of RDF blocks and present the results when using block sizes of 1, 5, 10, 25, 50 and 100 million triples. Overall and for our test system we achieve best results for an RDF block size of 10 million triples.

For investigating why using smaller or larger block sizes slows down index construction, Figure 6 contains the single times for the different phases of index construction[2]. While the processing times of most phases remain about the same for different block sizes or are not significantly compared to the total index construction time, building the patricia tries becomes slower for larger block sizes and mapping initial runs to global ids faster,

[2] In addition to the evaluation indices, LUPOSDATE constructs indices (called *histogram indices*) with the help of which LUPOSDATE efficiently determines histograms of triple pattern results [14]. Like constructing the evaluation indices, the construction of the histogram indices can be done in one pass through the corresponding final merged run, and roughly takes about the same time as constructing the evaluation indices. Please note that in our presentation of the experimental results the times to construct the histogram indices are included in the times to construct the evaluation indices.

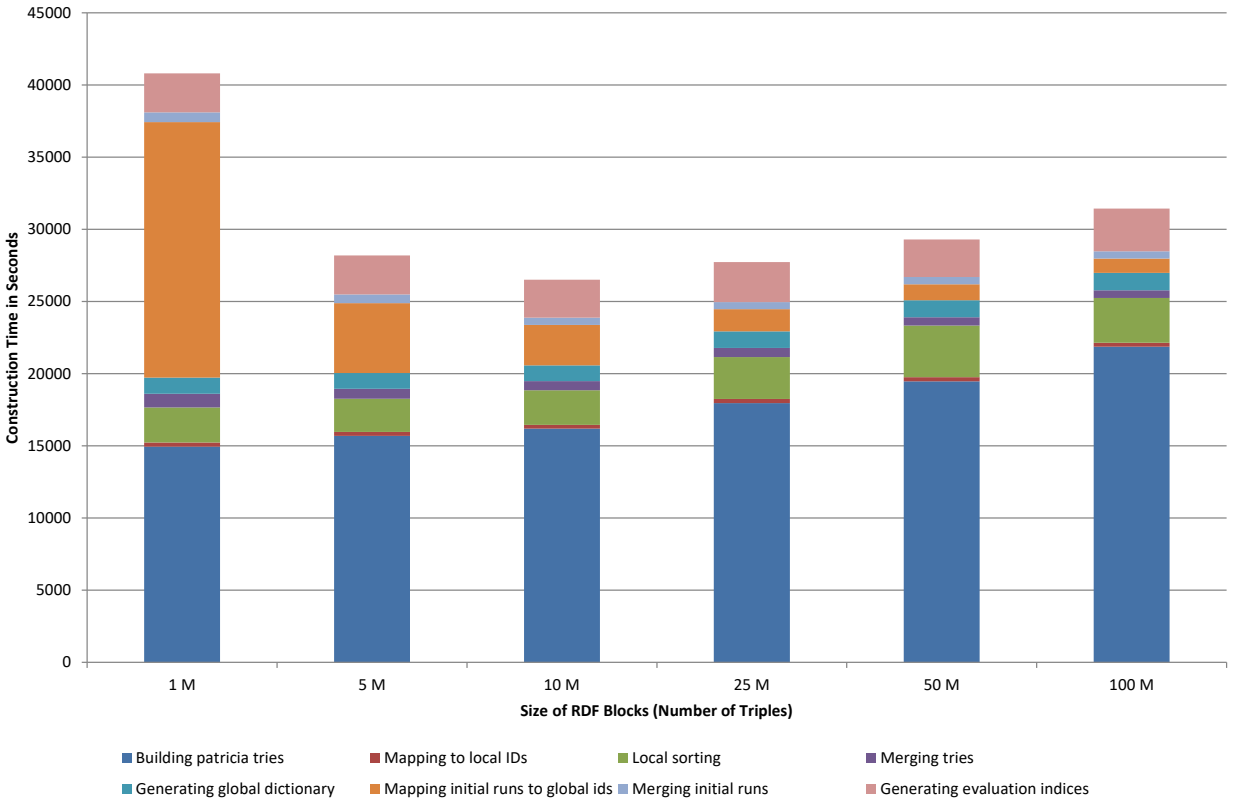

Figure 5: The total index construction times for different sizes of RDF blocks

i.e. we have two contrary trends for the processing times of index construction phases for larger block sizes. We also observed in [17] that larger block sizes increase the times for building the patricia tries, which may be explained by more complex computations necessary to add a string to a fuller patricia trie in comparison to the insertion into emptier patricia tries. For mapping initial runs to global ids we need more lookups in the global dictionary (for determining the global id of a given RDF term) for smaller RDF blocks, as often used RDF terms must be looked up for many RDF blocks containing these RDF terms. Larger block sizes hence decrease the number of lookups for duplicated RDF terms, as each RDF term is only looked up at most once for each block (see Section 3.7).

5 Summary and Conclusions

The Semantic Web with its large-scale datasets (e.g., those collected in the Linking Open Data (LOD) project [23, 24]) cries for efficient index construction approaches in order to build indexes from scratch for succeeding big data analytics.

In this paper we propose an efficient index construction approach in order to generate a dictionary (i.e., a mapping from the string representation of RDF terms to unique integer ids and vice versa) and evaluation indices according to the 6 collation orders of RDF, which are widely used indices in Semantic Web databases. We describe a sophisticated process, where the generation of the evaluation indices is smoothly integrated into the dictionary construction in order to save I/O costs as well as computation costs. Our proposed approach is parallelized in many phases of the index construction to take advantage of the today's multi-core CPUs. Furthermore, the data is compressed as early as possible in order to lower the main memory footprint and process bigger blocks of triples in main memory (by using patricia tries and ids as early as possible), and transfer less bytes to and from external storage (by natively storing patricia tries and applying difference encoding during storing the runs). The proposed index construction approach is also designed to apply very efficient algorithms like Counting Sort working on small domains, which we especially created for this purpose by introducing local ids for each block of triples.

A comprehensive experimental analysis shows the practical feasibility of the proposed approach by analyzing the execution times for importing over 1 billion triples of the overall construction approach.

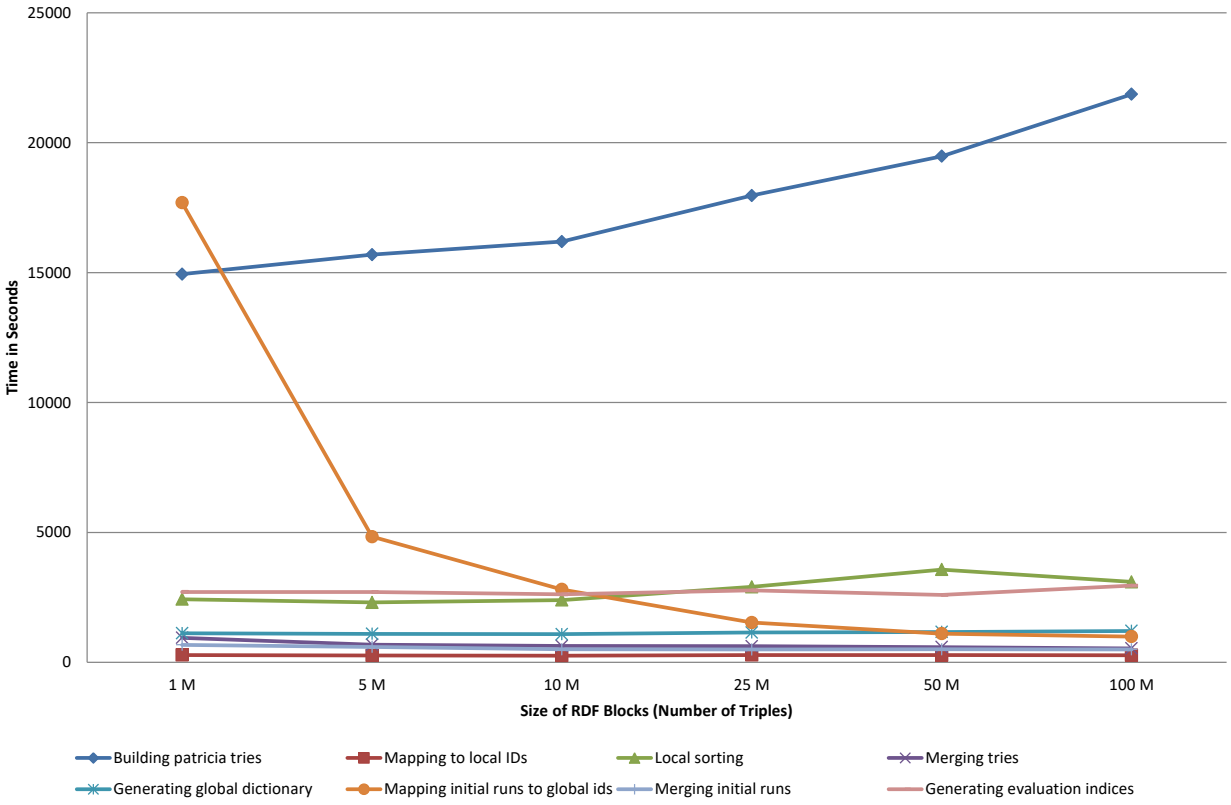

Figure 6: The processing times of the different phases of index construction for different sizes of RDF blocks

Our major contribution to the big data area is hence on volume since our proposed approach can manage, process and organize a large quantity of data, validated by system design and experimental results.

Future work includes research on distributed and hardware-accelerated index construction. Especially the construction of the patricia tries and generating the runs can be independently processed for each block of RDF data and hence these steps seem to be perfectly suitable for distributed processing and hardware-accelerating by FPGAs and/or GPUs. First results for the generation of patricia tries hardware-accelerated by FPGAs are already available [7], which need to be further extended to cover the full index construction process.

Handling large volumes of real data from various sources may cause data quality issues. Future work covers approaches to ensure data quality directly during data import. Furthermore, in recent years there is an increasing number of attacks based on data hacking and data security breaches. In order to overcome related problems we have to integrate methods to ensure that data to be imported is not altered, manipulated or falsely replicated, which we need to take care of even for publicly freely available datasets such as those of LOD. The development of whole systems like adapting [10] for RDF datasets (also in the context of LOD) is one of the key challenges in the near future.

ACKNOWLEDGEMENTS

This work is funded by the German Research Foundation (DFG) project GR 3435/9-1.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms*. Reading, Mass.: Addison-Wesley, 1983. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=0201000237

[2] Andreas Harth, "Billion Triples Challenge 2012 Dataset," http://km.aifb.kit.edu/projects/btc-2012/, 2012.

[3] R. Angrish and D. Garg, "Efficient string sorting algorithms: Cache-aware and cache-oblivious," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 1, 2011.

[4] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, "On sorting strings in external memory (extended

abstract)," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 540–548.

[5] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '70. New York, NY, USA: ACM, 1970, pp. 107–141.

[6] T. Berners-Lee and M. Fischetti, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*, 1st ed. Harper San Francisco, 1999.

[7] C. Blochwitz, J. M. Joseph, T. Pionteck, R. Backasch, S. Werner, D. Heinrich, and S. Groppe, "An optimized Radix-Tree for hardware-accelerated index generation for Semantic Web Databases," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 7 - 9 2015.

[8] K. Borne, "Top 10 Big Data Challenges A Serious Look at 10 Big Data Vs," Gartner, https://www.mapr.com/blog/top-10-big-data-challenges-serious-look-10-big-data-vs, 2014.

[9] D. Brickley and R. V. Guha, "Rdf vocabulary description language 1.0: Rdf schema, w3c recommendation," http://www.w3.org/TR/rdf-schema/, 2004.

[10] V. Chang, Y.-H. Kuo, and M. Ramachandran, "Cloud computing adoption framework: A security framework for business clouds," *Future Generation Computer Systems*, vol. 57, pp. 24 – 41, 2016.

[11] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[13] M. Duerst and M. Suignard, "Rfc 3987: Internationalized resource identifiers (iris)," *Internet Engineering Task Force (IETF)*, http://www.ietf.org/rfc/rfc3987.txt, 2005.

[14] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.

[15] S. Groppe, "LUPOSDATE Semantic Web Database Management System," https://github.com/luposdate/luposdate, 2015, [Online; accessed 26.1.2015].

[16] S. Groppe and J. Groppe, "External Sorting for Index Construction of Large Semantic Web Databases," in *Proceedings of the 25th ACM Symposium on Applied Computing, Vol. II (ACM SAC 2010)*. Sierre, Switzerland: ACM, 2010, pp. 1373–1380.

[17] S. Groppe, D. Heinrich, S. Werner, C. Blochwitz, and T. Pionteck, "Pattriesort - external string sorting based on patricia tries," *Open Journal of Databases (OJDB)*, vol. 2, no. 1, pp. 36–50, 2015. [Online]. Available: http://www.ronpub.com/publications/OJDB_2015v2i1n03_Groppe.pdf

[18] A. Harth and S. Bechhofer, "A new application award - Semantic Web Challenge," http://challenge.semanticweb.org/, 2014.

[19] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[20] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[21] D. Laney, "3D Data Management: Controlling Data Volume, Velocity and Variety," Gartner, http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf, 2001.

[22] Linked Data, "Linked Data - Connect Distributed Data across the Web," 2016, [Online; accessed 4.11.2016]. [Online]. Available: http://www.linkeddata.org

[23] LOD2, "LODStats," 2016, [Online; accessed 4.11.2016]. [Online]. Available: http://stats.lod2.eu/

[24] LOD2, "Welcome - LOD2 - Creating Knowledge out of Interlinked Data," 2016, [Online; accessed 4.11.2016]. [Online]. Available: http://lod2.eu

[25] R. Miller, N. Pippenger, A. Rosenberg, and L. Snyder, "Optimal 2-3 trees," in *IBM Research Lab. Yorktown Heights*, NY, 1977.

[26] R. E. Miller, N. Pippenger, A. L. Rosenberg, and L. Snyder, "Optimal 2, 3-trees." *SIAM J. Comput.*, vol. 8, no. 1, pp. 42–59, 1979.

[27] T. Neumann and G. Weikum, "RDF3X: a RISC-style Engine for RDF," in *VLDB*, Auckland, New Zealand, 2008.

[28] T. Neumann and G. Weikum, "Scalable join processing on very large RDF graphs," in *SIGMOD*, 2009.

[29] W. OWL Working Group, *OWL 2 Web Ontology Language: Document Overview (Second Edition)*. W3C Recommendation, 11 December 2012, available at http://www.w3.org/TR/owl2-overview/.

[30] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[31] H. H. Seward, *2.4.6 Internal Sorting by Floating Digital Sort, Information sorting in the application of electronic digital computers to business operations, Master's thesis, Report R-232*. Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[32] J. Seward, "bzip2 - bzip2 and libbzip2," http://www.bzip.org/, 2014.

[33] R. Sinha and J. Zobel, "Using random sampling to build approximate tries for efficient string sorting," *J. Exp. Algorithmics*, vol. 10, p. 2.10, 2005.

[34] R. Sinha and J. Zobel, "Efficient trie-based sorting of large sets of strings," in *Proceedings of the 26th Australasian Computer Science Conference - Volume 16*, ser. ACSC '03. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 11–18.

[35] R. Sinha and J. Zobel, "Cache-conscious sorting of large sets of strings with dynamic tries," *J. Exp. Algorithmics*, vol. 9, Dec. 2004.

[36] M. A. u. d. Khan, M. F. Uddin, and N. Gupta, "Seven v's of big data understanding big data to extract value," in *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education*, 2014, pp. 1–5.

[37] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," in *VLDB*, 2008.

[38] World Wide Web Consortium, "W3C Data Activity - Building the Web of Data," https://www.w3.org/2013/data/, 2016.

[39] World Wide Web Consortium (W3C), "RDF/XML Syntax Specification (Revised)," 2004, w3C Recommendation. [Online]. Available: http://www.w3.org/2004/REC-rdf-syntax-grammar-20040210/

[40] J. Yiannis and J. Zobel, "Compression techniques for fast external sorting," *The VLDB Journal*, vol. 16, no. 2, pp. 269–291, Apr. 2007.

AUTHOR BIOGRAPHIES

Sven Groppe earned his diploma degree in Informatik (Computer Science) in 2002 and his Doctor degree in 2005 from the University of Paderborn. He completed his habilitation degree in 2011 in the University of Lübeck. He worked in the European projects B2B-ECOM, MEMPHIS, ASG and TripCom. He was a member of the DAWG W3C Working Group, which developed SPARQL. He was the project leader of the DFG project LUPOSDATE, an open-source Semantic Web database, and one of the project leaders of two research projects, which research on FPGA acceleration of relational and Semantic Web databases. He is one of the workshop chairs of the Semantic Big Data workshop series in conjunction with ACM SIGMOD. His research interests include databases, Semantic Web, query and rule processing and optimization, Cloud Computing, peer-to-peer (P2P) networks, Internet of Things, data visualization and visual query languages.

Dennis Heinrich received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. At the moment he is employed as a research assistant at the Institute of Information Systems at the University of Lübeck. His research interests include FPGAs and corresponding hardware acceleration possibilities for Semantic Web databases.

Christopher Blochwitz received his M.Sc. Diploma in Computer Science in September 2014 at the University of Lübeck, Germany. Now he is a research assistant/ PhD student at the Institute of Computer Engineering at the University of Lübeck. His research focuses on hardware acceleration, hardware optimized data structures, and partial reconfiguration of FPGAs.

Thilo Pionteck is an associate professor at the Universität Magdeburg, Germany. He received his Diploma degree in 1999 and hIs Ph.D. (Dr.-Ing.) degree in Electrical Engineering both from the Technische Universität Darmstadt, Germany. In 2008 he was appointed as an assistant professor for Integrated Circuits and Systems at the Universität zu Lübeck. From 2012 to 2014 we was substitute of the Chair of Embedded Systems at the Technische Universität Dresden and of the Chair of ”Computer Engineering” at the Technische Universität Hamburg-Harburg. His research work focus on adaptive system design, runtime reconfiguration, hardware/software codesign and network-on-chips.