An Update Strategy for the WaterFowl RDF Data Store

Olivier Curé[1], Guillaume Blin[2]

[1] Université Paris-Est, LIGM - UMR CNRS 8049, France
ocure@univ-mlv.fr
[2] Université de Bordeaux, LaBRI - UMR CNRS 5800, France
guilllaume.blin@labri.fr

Abstract. The WaterFowl RDF Store is characterized by its high compression rate and a self-indexing approach. Both of these characteristics are due to its underlying architecture. Intuitively, it is based on a stack composed of two forms of Succinct Data Structures, namely bitmaps and wavelet trees. The ability to efficiently retrieve information from these structures is performed via a set of operations, *i.e.*, `rank`, `select` and `access`, which are used by our query processor. The nice properties, *e.g.* compactness and efficient data retrieval, we have observed on our first experimentations come at the price of poor performances when insertions or deletions are required. For instance, a naive approach has a dramatic impact on the capacity to handle ABox updates. In this paper, we address this issue by proposing an update strategy which uses an hybrid wavelet tree (using both pointer-based and pointerless sub-wavelet trees).

1 Introduction

Large amount of RDF data are being produced in diverse domains. Such a data deluge is generally addressed by distributing the workload over a cluster of commodity machines. We believe that this will soon not be enough and that in order to respond to the exponential production of data, the next generation of systems will distribute highly compressed data. One valuable property of such systems would be to perform some data oriented operations without requiring a decompression phase.

We have recently proposed the first building blocks of such a system, namely WaterFowl, for RDF data [1]. The current version corresponds to an in-memory, self-indexed, operating at the bit level approach which uses data structures with a compression rate close to theoretical optimum. These so-called Succinct Data Structures (SDS) support efficient decompression-free query operations on the compressed data. The first components we have developed for this architecture are a query processor and an inference engine which supports the RDFS entailment regime with inference materialization limited to `rdfs:range` and `rdfs:domain`. Both of these components take advantage of the SDS properties and highly performant operations. Of course, SDS are not perfect and a main limitation corresponds to their inability to efficiently handle update operations,

i.e., inserting or deleting a bit. In the worst case, one has to completely rebuild the corresponding SDS (bitmap or wavelet tree) to address such updates. Even if we consider that the sweet spot for RDF stores is OnLine Analytic Processing (OLAP), rather than OnLine Transactional Processing (OLTP), such a drawback is not acceptable when managing data sets of several millions of triples.

The main contribution of this paper is to present an update strategy that addresses instance updates. Due to space limitations, we do not consider updates at the schema level (*i.e.*, TBox). This approach is based on a set of heuristics and the definition of an hybrid wavelet tree using both pointer-based and pointerless sub-wavelet trees.

2 WaterFowl architecture

Figure 1 presents the three main components of the WaterFowl system: dictionary, query processing and triples storage. The former is responsible for the encoding and decoding of the entries of triple data sets. The main contribution here consists of encoding the concepts and properties of an ontology such that their respective hierarchy are using common binary prefixes. This enables us to perform prefix versions of the `rank`, `select` and `access` operations and thus prevents navigating the whole depth of the wavelet trees to return an answer.

The query processing component handles the classical operations of a database query processor and communicates intensively with the dictionary unit. A peculiarity of this query processor is to translate SPARQL queries into sequences of `rank`, `select` and `access` SDS operations over sequences. They are designed to (i) count the number of occurences of a letter appearing in a prefix of a given length, (ii) find the position of the kth occurrence of a letter and (iii) retrieve the letter at a given position in the sequence.

Finally, the triple storage component consists of two layers of bitmaps and wavelet trees[3]. It uses an abstraction of a set of triples represented as a forest where each tree corresponds to a single subject, *i.e.*, its root, the intermediate nodes are the properties of the root subject and the leaves are the objects of those subject-property pairs. The first layer encodes the relation between the subjects and the predicates of a set of triples. It is composed of a bitmap, to encode the subjects, and a wavelet tree, to encode the sequences of predicates of those subjects. Unlike the first layer, the second one has two bitmaps and two wavelet trees. B_o encodes the relation between the predicates and the objects; that is the edges between the leaves and their parents in the tree representation. Whereas, the bitmap B_c encodes the positions of ontology concepts in the sequence of objects. Finally, the sequence of objects obtained from a pre-order traversal in the forest is split into two disjoint subsequences; one for the concepts and one for the rest. Each of these sequences is encoded in a wavelet tree (resp. WT_{oc} and WT_{oi}). This architecture reduces sparsity of identifiers and enables the management of very large datasets and ontologies while allowing time and

[3] Due to space limitations, we let the reader refer to [1] for corresponding definitions

space efficiency. More details on these components are proposed in [1]. That paper presents some evaluations where different wavelet tree implementations have been used: with and without pointers and one so-called matrix [3]. The characteristics of the first two motivated our update approach.

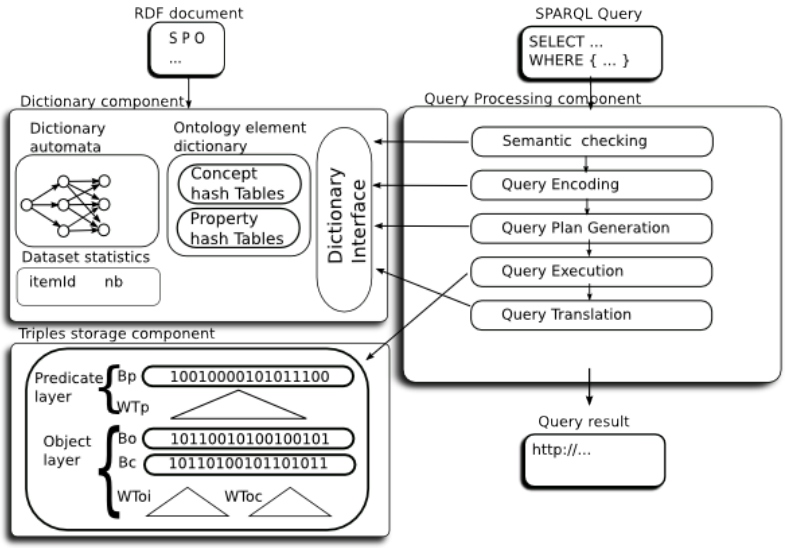

Fig. 1. WaterFowl's architecture

3 Update strategy

As previously mentioned, one of the benefits of using wavelet trees in our system is the ability to compress drastically the data while being able to query it without decompression. The main drawback of such SDS lies in its requirement to precompute and store some small but neccessary extra informations. More formally, the bitmaps used in the inner construction of the wavelet trees requires $n + o(n)$ bits of storage space (the original bit array and an $o(n)$ auxiliary structure) to support `rank` and `select` in constant time. Note that while the implementaton of `rank` is simple and practical, this is not the case for `select` which should be avoided whenever possible, *e.g.*, in our SPARQL query translations. Recently, some attempts were done in order to provide faster and smaller structure for `select` [4]. These precomputed auxiliary informations are used during query processing in order to get constant time complexity. This is why, by definition, the bitmaps are static and cannot be updated. In our context, it implies an immutable RDF store (which is quite restrictive).

In order to overcome this issue, we propose an update strategy using the inner tree structure of the wavelet trees. First, recall that there are mainly two

implementations of the wavelet trees: with and without pointers. On one hand, the implementation without pointers uses less memory space and provides better time performances. On the other hand, any modification to the represented sequence implies a full reconstruction of the wavelet tree; while, in the case where pointers are used, only a subset of the nodes (and the corresponding bitmaps) have to be rebuilt which in our first experiments is much faster than a total reconstruction. A second important point of our approach is based on the fact that in wavelet trees, each bit of an encoded entry specifies a path in the tree. That is, the instance data only influence the size of nodes but not their placement in the tree. Considering these two assumptions together with the huge amount of data we want to handle, our strategy supports an hybrid approach based on the natural definition of a tree. Indeed, a tree can be defined recursively as a node with a sequence of children which are themselves trees. Our hybrid wavelet tree is then defined as a node representing a wavelet tree without pointers of height k and a sequence of 2^k children which are themselves hybrid wavelet trees. In practice, considering a querying scenario composed of read (e.g., select) and write (e.g., add, delete, update) operations, for performance purpose the hybrid wavelet tree can adapt its composition to the scenario by minimizing and maximizing the numbers of pointers in the depth traversal of respectively a read and write operation. Note that this approach of cracking the database into manageable pieces is reminiscent to dynamic indexing solution presented in [2].

4 Conclusion

This poster exploits available wavelet tree implementations to address the issue of updating an ABox. We have already implemented a prototype of the WaterFowl system and of the updating system. They so far provide interesting performance results but we have yet to test with real use cases. This will enable us to observe practical modifications and to study their efficiency. These observations should provide directions for optimizations. Our future work, we will consist in the development of two new components. A first one will address updates at the schema level, *i.e.*, insertions or removals of concepts and properties of the underlying ontology. The second one will consider the partitioning of a data set over a machine cluster together with both ABox and TBox updates.

References

1. Olivier Curé, Guillaume Blin, Dominique Revuz, and David Célestin Faye. Waterfowl: A compact, self-indexed and inference-enabled immutable rdf store. In *ESWC*, pages 302–316, 2014.
2. Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.
3. Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.
4. Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In Ralf Klasing, editor, *Experimental Algorithms*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer Berlin Heidelberg, 2012.