System Architecture and Query Processing of the WaterFowl RDF Store

Olivier Curé
LIGM CNRS UMR 8049
Université Paris-Est, France
ocure@univ-mlv.fr

Guillaume Blin
LaBRI CNRS UMR 5800
Université de Bordeaux, France
guillaume.blin@labri.fr

Thomas Le Roux, Remy Luciani, Ludovic Feltz, Arnaud Thalamot, Frédérick Richard
Université Paris-Est, France
{firstname.lastname}@univ-mlv.fr

ABSTRACT

Data management systems evolving in big data ecosystems usually address the volume issues with distribution and replication mechanisms. Unfortunately, very few of these systems consider data compression has a salient aspect. We consider that database management systems could benefit from the nice properties of state of the art compressed structures. In this paper, we present an architecture for the storage of RDF triples that is based on highly compressed structures and its set of decompression-free querying operations. Our current prototype proposes efficient storage, querying and reasoning facilities. The potential of our query processor is demonstrated on evaluations conducted on synthetic datasets.

1. INTRODUCTION

The main approaches used to address the volume issues of big data have been the distribution and replication of data over a cluster of commodity hardware. Although generally quite efficient, we consider that by itself, these solutions will soon be insufficient to confront coming data deluges. We argue that local, *i.e.*, on each machine of the cluster, data compression will soon become an essential aspect of such systems. Obviously, database management systems designed on this approach have positive financial and ecological impacts since less machines are needed to serve the same applications. Moreover, we claim that better performances can be obtained if one uses the right compression methods and data structures, *e.g.*, working with a dataset loaded in main-memory. The full potential of such an approach is reached when the compressed structures can be queried with decompression-free operations, *e.g.* `rank`, `select` and `access`. Recently, several research advances have identified such data structures which are regrouped under the Succinct Data Structures (henceforth SDS) designation.

In this article, we present a novel system architecture based on a subset of SDS, namely bit mapss and wavelet trees. Our system, named WaterFowl, tackles the management of Resource Description Framework (RDF) data, a popular format of the open source, *e.g,* Linked Open Data, and big data movements. Several so-called RDF stores have been proposed [9], [12], [1] but to the best of our knowledge, none of them are pushing the compression aspect as far as WaterFowl does. Because the system uses highly compressed data structures, the representation of RDF triples can be stored in main-memory. This limits disk-based Input/Output operations to the loading and recording of the database using (de)serialization mechanisms. The in-memory footprint is also kept to its minimum due to the presence of a single index which has the self-indexed property, *i.e.*, the complete database can be reconstructed from that index. Finally, some of the entities, *i.e.*, concepts and properties, are 'cleverly' encoded according to the wavelet tree specificities. This enables to represent the concept and property hierarchies within binary encodings. More details on the reasoning mechanisms and capacities of WaterFowl are provided in [2].

This work leans on the description of the system architecture and its storage component to present the WaterFowl query processor. It is based on a translation of SPARQL queries into a set of `rank`, `select` and `access` operations which are performed directly on our triple storage component. Our query optimizer uses heuristics that are based on the syntactic and structural aspects of SPARQL triple patterns and which are motivated by complexity considerations of our translations. Before getting into the details of these components, we first provide some background knowledge on RDF and SPARQL as well as SDS.

1.1 Background: RDF and SPARQL

RDF is a schema-free data model that permits to describe data on the Web. It is usually considered as the cornerstone of the Semantic Web. Assuming disjoint infinite sets U (RDF URI references), B (blank nodes) and L (literals), a triple (s,p,o) $\in$ (U $\cup$ B) x U x (U $\cup$ B $\cup$ L) is called an RDF triple with s, p and o respectively being the subject, predicate and object. We now also assume that V is an infinite set of variables and that it is disjoint with U, B and L. We can recursively define a SPARQL[1] triple pattern as fol-

[1]http://www.w3.org/TR/rdf-sparql-query/

lows: (i) a triple $tp \in$ (U $\cup$ V) x (U $\cup$ V) x (U $\cup$ V $\cup$ L) is a SPARQL triple pattern, (ii) if tp_1 and tp_2 are triple patterns, then ($tp_1.tp_2$) represents a group of triple patterns that must all match, (tp_1 `OPTIONAL` tp_2) where tp_2 is a set of patterns that may extend the solution induced by tp_1, and (tp_1 `UNION` tp_2), denoting pattern alternatives, are triple patterns and (iii) if tp is a triple pattern and C is a built-in condition then the expression (tp `FILTER` C) is a triple pattern that enables to restrict the solutions of a triple pattern match according to the expression C. The SPARQL syntax follows the select-from-where approach of SQL queries. The `SELECT` clause specifies the variables appearing in the result set of the query.

In [5], extensions to SPARQL semantics, called entailment regimes, are presented. The WaterFowl system addresses RDF and RDFS entailment regimes in the context of skolemization as presented in [5], *i.e.*, a syntactic transformation that replaces blank nodes by 'new' names. Most of the inferences we are considering are related to entailment rules proposed in [6]. In the context of query answering, these rules are useful to check satisfiability and to rewrite queries. In our system, they are implemented through the use of adapted encodings and data structures which are directly motivated by SDS.

1.2 Background: Succinct Data Structures

The family of SDS uses a compression rate close to theoretical optimum and simultaneously allows efficient decompression-free query operations on the compressed data. This property is obtained using a small amount (o(Z) bits where Z corresponds to the theoretical optimum) of extra bits to store extra information. Initially introduced by Jacobson [7] when considering bit vectors (a.k.a. bit maps), the concept is nowadays extended to wider alphabets. Bit vectors are useful to represent data while minimizing its memory footprint. In its classical shape, a bit map allows, in constant time, to access and modify a value of the vector. Munro [8] designed an asymptotic optimal version where, in constant time, one can (i) count the number of 1 (or 0) appearing in the first x elements of a bit vector (denoted $\texttt{rank}_b(x)$ with b $\in$ {0,1}), (ii) find the position of the x^{th} occurrence of a bit (denoted $\texttt{select}_b(x)$, b $\in$ {0,1}) and (iii) retrieve the bit at position x (denoted `access(x)`). Naturally, these operations on bit maps would be of great interest for a wider alphabet. The original solution was provided by Grossi *et al.* [3] and roughly consists in using a balanced binary tree – so-called wavelet tree. The alphabet is splitted into two equal parts. One attributes a 0 to each character of the first part and a 1 to the others. The original sequence is written, at the root of the tree, using this encoding. The process is repeated, in the left subtree, for the subsequence of the original sequence only using characters of the first part of the alphabet and, in the right subtree, for the second part. The process iterates until ending up on singleton alphabet. Intuitively, one has provided an encoding of each character of the alphabet. Using `rank` and `select` operations on the bit vectors stored in the nodes of the tree, one is able to compute `rank` and `select` operations on the original sequence in O(log |alphabet|) by deep traversals of the tree. Wavelet trees have been well studied since then and both space and time efficient implementations are now available (*e.g.*, pointer-free wavelet tree and wavelet matrix of libcds library[2] and sdsl-lite[3]).

2. WATERFOWL'S ARCHITECTURE

As presented in Figure 1, WaterFowl follows a modular architecture approach. The three main components are the 'Statistics and Dictionary', 'Triple storage' and 'Query processing' components. A client communication manager accepts WaterFowl commands to enable the creation of a database, to provide an RDF dataset and optionally an RDFS or OWL ontology, to load or remove an existing database, to process a SPARQL query and to obtain explanations on its execution. This manager communicates with the 'Statistics and dictionary' component in case of data creation or loading or 'Query processing' module in a case of execution of a SPARQL query.

The dictionary component provides a unique integer identifier to each subject, property and object entries of the RDF dataset. These identifiers are later used to encode the complete RDF triples. This limits the memory footprint by preventing from storing multiple times long URIs in our database. The identifiers are currently stored using 64 bits but due to a contextualization approach, *i.e.*, the second value of an encoded triple is necessarily a property, we can allow for some identifier overlapping, *e.g.*, between property and concept identifiers. During the encoding step, some simple statistics, *e.g.*, number of occurrences of each entry, are being computed. We will emphasize in Section 3 that they support some query optimizations. Due to space limitations, we do not provide additional details on this 'statistics and dictionary' and invite interesting readers to refer to [2].

The triple storage component manages our two-layer SDS structure. We present the main aspects of this component through an example using the popular Lehigh University Benchmark (LUBM) [4] and depict an extract in Figure 2. Once the dictionaries have been defined, the triples can be encoded in a structure that makes an intensive use of SDS. To illustrate the structure, we will encode the following simple RDF triples: `{(Uni0, rdf:type, ub:University), (Uni0, ub:name, "University0"), (Dpt0, rdf:type, ub:Department), (Dpt0, ub:name, "Department0"), (Dpt0, ub:subOrganizationOf, Uni0), (AP0, rdf:type, ub:AssociateProfessor), (AP0, ub:name, "Cure"), (AP0, ub:teacherOf, C15), (AP0, ub:teacherOf, C16), (AP0, ub:worksFor, Dpt0), (C15, rdf:type, ub:Course), (C15, ub:name, "Course15"), (C16, rdf:type, ub:Course)}` .

The triples are first ordered by subjects, predicates and then objects. The ordered forest of Figure 2(a) will serve to demonstrate the creation of our two-layers structure where each layer is composed of bitmaps and wavelet trees. The first layer encodes the relation between the subjects and the predicates; that is the edges between the root of each tree and its children. The bitmap B_p is defined as follows. For each root of the trees in Figure 2(a) – that is each subject – the leftmost child is encoded as a 1, and the others as a

[2]https://code.google.com/p/libcds/
[3]https://github.com/simongog/sdsl-lite
[4]http://swat.cse.lehigh.edu/projects/lubm/

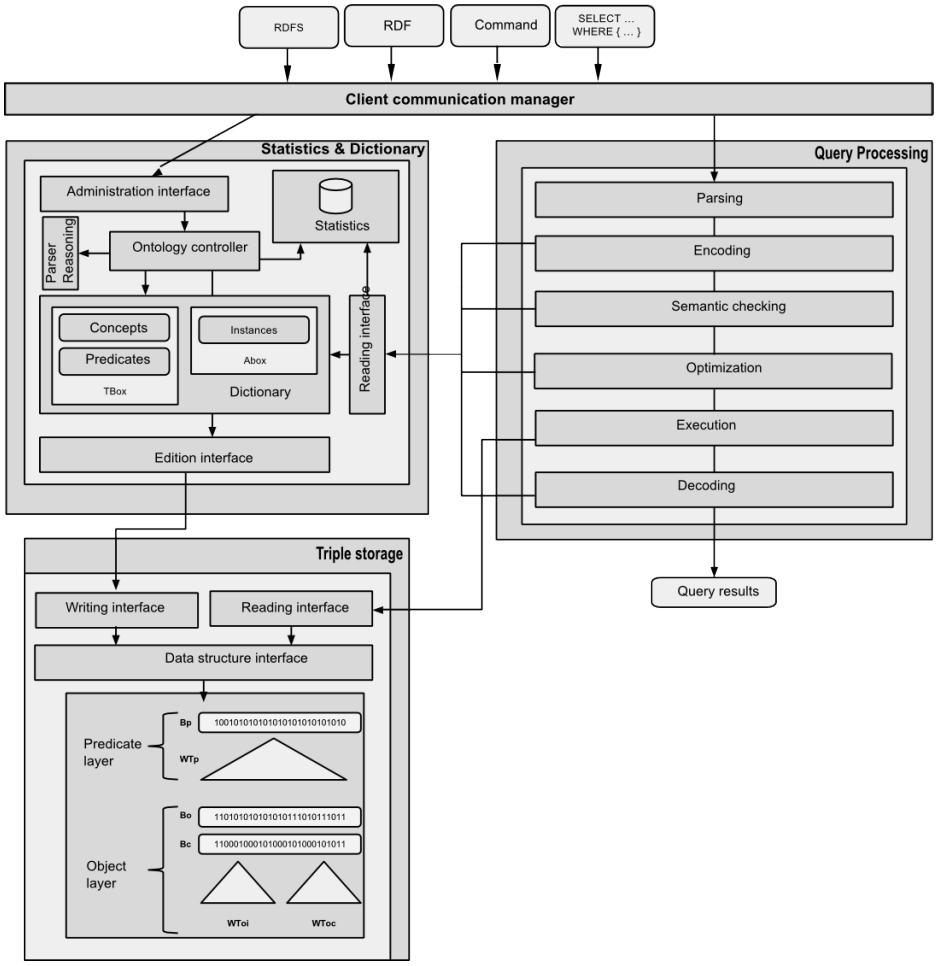

Figure 1: Architecture overview

0. On the whole, B_p contains as many 1's as subjects in the dataset and is of length equal to the number of predicates in the dataset. In Figure 2(c), one obtains 101001000101 since there are 5 subjects with the last subject having 1 predicate, the first and fourth subjects having 2 predicates, the second one having 3 while the third one have 4. The wavelet tree WT_p encodes the sequence of predicates obtained from a pre-order traversal in the forest (*i.e.*, second row in Figure 2(a)). The construction of the wavelet tree follows the algorithm presented in Section 1.2.

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 obtained from a pre-order traversal in the forest (*i.e.*, third row in Figure 2(a)). The bitmap B_o is defined as B_p considering the forest obtained by removing the first layer of the tree representation (that is the subjects). In Figure 2(a), one obtains 1111111101111. The bitmap B_c stores a 1 at each position of an object which is a concept; a 0 otherwise. This is processed using a predicate contextualization, *i.e.*, in the dataset whenever a *rdf:type* appears, we know that the object corresponds to an ontology concept. In Figure 2(a), considering that the predicate *rdf:type* is encoded by 00, one obtains 1010010000101. Finally, the sequence of objects obtained from a pre-order traversal in the forest (*i.e.*, third row in Figure 2(a)) is splitted 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 space efficiency. The query processing component is fully described in the next section with a special focus on the optimization aspects.

3. QUERY PROCESSING

WaterFowl's query processor consists of the sub-components displayed in the rightmost box of Figure 1. Some of them are following a standard relational database management system approach but with the peculiarity of substituting SQL for SPARQL.

EXAMPLE 1. *Throughout this section, we are using LUBM's query #2 as a running example.*

```
SELECT ?x ?y ?z WHERE {
?x rdf:type lubm:GraduateStudent .
?y rdf:type lubm:University .
```

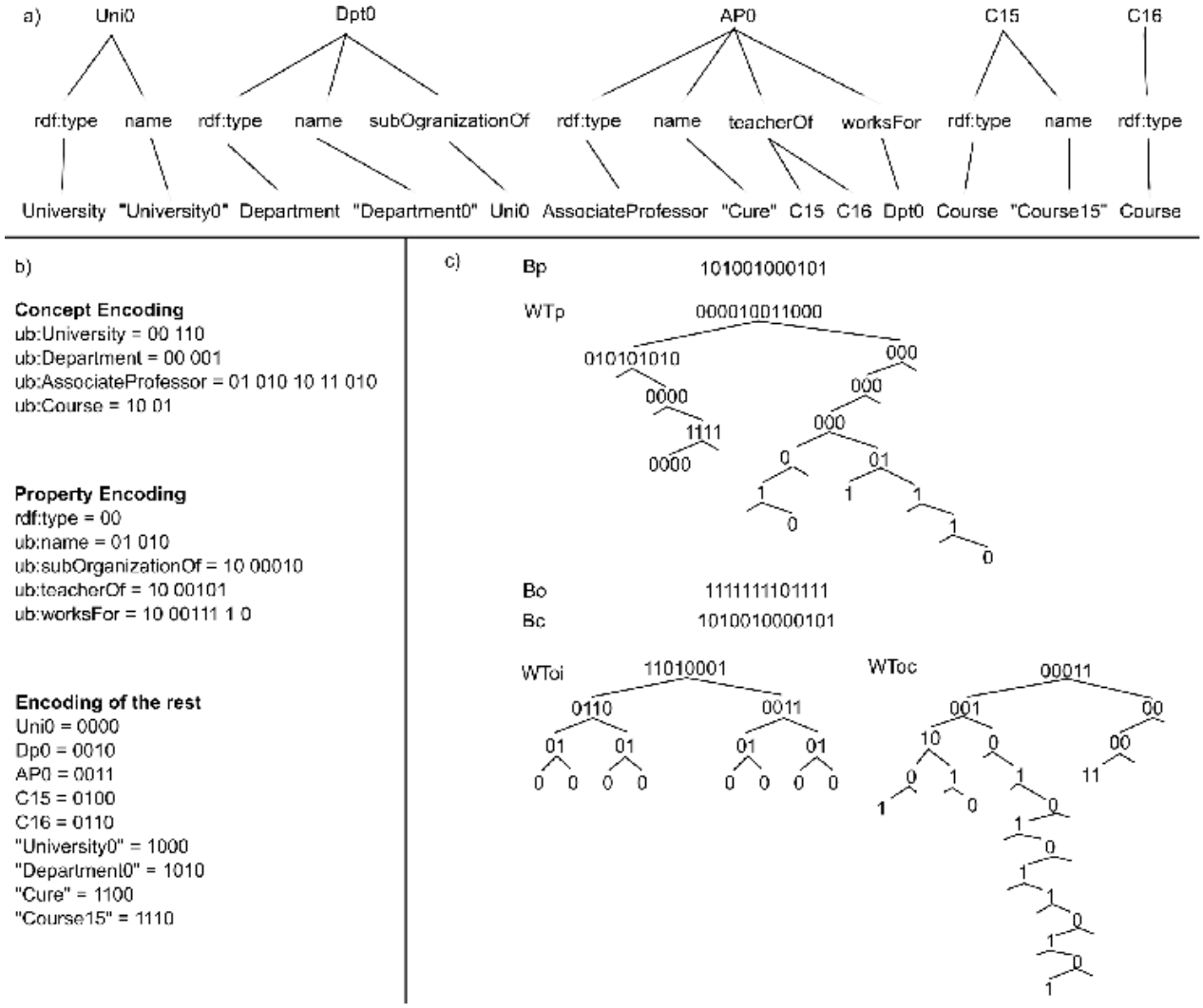

Figure 2: Two-layer overview with a forest representation of the RDF triples (a), entry encoding (b) and the two layers of bit maps and wavelet trees (c).

```
?z rdf:type lubm:Department .
?x lubm:memberOf ?z .
?z lubm:subOrganizationOf ?y .
?x lubm:undergraduateDegreeFrom ?y.}
```

This is the case for the parsing sub-component which is relying an external framework, namely Apache Jena [5], to perform the parsing of SPARQL queries and to transform it in a specific algebra. Parsing is performed on the non-encoding version of the query. The main idea is to identify malformed queries – from the SPARQL point of view, not to datasets.

The encoding/decoding operations aim at translating URIs, literals and possibly blank nodes found in the SPARQL query into the corresponding database's identifiers, resp. translate identifiers present in the result set into URIs and literals. Given a certain dictionary, *i.e.*, either concept, property or instance, we present an encoding for the triples of our running example. Note that in Table 1 we also provide the statistics associated to each triple entry which are not a variable, *e.g.*, the dataset contains 18128 occurrences of the `rdf:type` property which is encoded as '0'. These statistics do not consider correlations bteween subjects, properties or objects, *i.e.*, line #1 states that object #8 appears 1874 in the dataset but we do not know how many times it appears in the context of the #0 property. Hence, this approach only provides some approximations, but their computation is fast and require a limited memory footprint, moreover they are sufficient in the context of our query processor.

The three remaining sub-components, namely semantic checking, optimization and execution, are specific to the SDS support of RDF triples and deserve dedicated sub-sections.

3.1 Semantic checking

This sub-component checks the satisfiability of a query and tries to minimize its set of triple patterns. This can only be performed for databases proposing an access to an associ-

[5]https://jena.apache.org/

	Triples			Statistics		
#	S	P	O	S	P	O
1	?x	0	8	_	18128	1874
2	?y	0	54	_	18128	979
3	?z	0	3211	_	18128	15
4	?x	6	?z	_	7790	_
5	?z	15	?y	_	239	_
6	?x	2	?y	_	2414	_

Table 1: Encoded triple patterns and statistics of our running example

ated ontology, *e.g.*, expressed in RDFS or OWL. Considering RDFS entailment regime, this requires to access information about the domain and range of properties. In the case of OWL ontologies, we are also interested on functional and inverse functional properties. All the addressed metadata are stored in the 'Statistics and Dictionary' component.

Together with its ontology, LUBM's query #2 presents a simplification possibility. The second triple imposes that the `?y` variable is of type `lubm:University` but the ontology states that the range of `lubm:undergraduateDegreeFrom` is necessarily of this type. Hence the second triple pattern of this query can be removed and still retain the semantics of the original query.

3.2 Triple pattern optimization

This sub-component aims at optimizing the performance of the query execution. In the context of a SPARQL query, which can contain a very large number of triple patterns and hence a large join order search space, this optimization mainly amounts at identifying an efficient triple pattern order execution. The absence of schematic structure in RDF makes this order discovery more involved than in a relational database context. As a consequence, most RDF stores rely on heuristics. Like in HSP [11], our main heuristics are based on the syntactic and structural aspects of triple patterns. Nevertheless, we enrich our approach with the statistics that our system has computed during the encoding step. Our main heuristic is based on an ordering of the eight possible SPARQL triple patterns. In order to define that order, we take into account the cost of executing these patterns in the context of our storage architecture, *i.e.*, in terms of `rank`, `select` and `access`. Table 2 summarizes the cost of all possible triple patterns. In theory, `rank`, `select` and `access` can be performed in O(1) for bitmaps, *e.g.*, using the RRR structure [10] and for Wavelet Trees, `rank`, `select` and `access` can be processed in O(log|A|), with |A| the size of the alphabet. In practice, one must try to avoid the use of `select` due to the significant space overhead associated to this operation. Thus, we can define the following order (from less to most expensive):
(SPO) $\preceq$ (SP?) $\preceq$ (S?O) $\preceq$ (?PO) $\preceq$ (S??) $\preceq$ (??O) $\preceq$ (??P) $\preceq$ (???)

Note that like in HSP, the more variables in a triple pattern, the more expensive is the execution. Nevertheless, due to the forest shape of our overall structure, given a number of variable, it is always less expensive to start from the subject and descend the two layer structure.

This order enables to define a sequence in which the triple patterns of a given query have to be executed. In the situation where several triple patterns of a query have the same shape, we use statistics to define an order among them. For instance, in Table 1, the first three triples follow a (?PO) pattern. Using the statistics, we can state that the triples should be executed in the order #3, #2 and #1 due to the selectivity on the number of objects, resp. 15, 979 and 1874.

Considering the semantic checking (which removes triple pattern #6 from our running example) and the optimization steps, the execution order is #3, #2, #1, #5 and #4.

3.3 Query execution

The Execution sub-component is tightly collaborating with the optimization one. That is after each triple pattern execution, denoted as an iteration, over the database, we are retrieving some new data from the store but also some information that may serve to reorder the sequence of triple patterns waiting for execution. Hence, the optimization is dynamic and is possibility invoked at each execution step. Consider the order computed on our running example. Once triple pattern #3 is executed, *i.e.*, (?z,P,O), the remaining triple patterns of our list can be updated by considering that the variable ?z is now a constant and that we can evaluate its size. The only triple impacted in Table 1 is #5 which can be defined as (α,15, ?y), where α is the list of binding for ?z. It can be considered as a (SP?) triple pattern and is thus preferred to the other one variable triple patterns, namely #2 and #1, which are (?PO). This process goes on until no triple patterns need to be executed.

This dynamic optimization/triple pattern execution takes the form of a graph exploration as opposed to left deep or bushy plans based on joins. This Breadth First Search (BFS) approach is well adapted to WaterFowl's two-layer storage architecture. At each triple pattern iteration step, a set of binding is retrieved or updated for a given variable and that set can be used in successive iteration steps. Two structures, denoted `node` and `container`, ensure the storage of intermediate results generally needed in BFS approaches. The latter is a collection of `nodes` that represent a SPARQL variable. Hence, a SPARQL query is attached to different `containers` in order to store intermediate results. These `containers` are identified by the variables they are representing and store two flags to support join operators: `exclude` and `unlink`. The former one indicates whether a variable is distinguished or not, while the latter specifies whether it is required to clean among the nodes, *i.e.*, removes, in a garbage collector manner, the nodes that are not related anymore.

A node is identified by its container and value pair and is made up of a list of nodes that represent its relationships. These relationships are either symmetric or bijective in order to allow each node to be aware of its links. In the case of a node removal, this node notifies its relations.

Ideally, the injection and cleaning operations are performed in two distinct processes. The injection takes care of the insertion of a node and creates the links if the joins are authorized (according to the `exclude` flag). Cleaning consults concerned containers in order to remove unnecessary nodes.

Pattern	Translation
S P O	4 select BM + rank WT + select WT + rank BM + access BM + access WT
S P ?	4 select BM + rank WT + select WT + rank BM + nO_SP * (access BM + access WT)
S ? O	2 select BM + nP_S * (2 select BM + nO_SP * (access Bc + rank Bc + access WT))
? P O	min (nO * (select WT + 2 rank BM + select BM + access WT, nP * (select WT + 2 rank BM + 2 select BM + access WT))
S ? ?	2 select BM + nP_S * (2 select BM + access WT + nO_SP * (access Bc + rank Bc + access WT))
? ? O	nO * (2 select WT + 2 rank BM)
? P ?	nP * (select WT + 2 rank BM + 2 select BM + access BM + access WT)
? ? ?	nS * (2 select BM + nP_S * (select WT + 2 rank BM + 2 select BM + nO_SP* (access WT))

Table 2: Triple pattern heuristic: A letter (S,P or O) stands for a constant (URIs, blank nodes or literals), the '?' stands for a variable. nS, nP, nO are resp. the number of occurrences of a given subject, predicate and object. nP_S is number of predicates of a given subject. nO_SP is number of objects for a given subject-predicate pair. Finally, BM and WT stand for Bit Map and Wavelet Tree.

Table 3: Query answering times (sec) on univ1000

	#1	#2	#4	#5	#14
RDF-3X	1.65	14.8	4.2	2.5	1640
BigOWLIM	138	5.7	705	16771	3320
Jena TDB	3.5	2.18	4.8	6.3	2998
WaterFowl	1.7	10.1	3.6	2.3	1680

A synchronization mechanism prevents concurrent injection and cleaning modifications.

4. EVALUATION

In this section, we present the results of our evaluation performed on a set of synthetic dataset corresponding to an instance of LUBM [4]. All experiments have been conducted on a HP Z800 workstation with 2 Quad-Core Intel Xeon Processors with 12Mbytes L2 cache, 8Gbytes of memory and running Gentoo 2.6.37 generic x86-64. It contains two 500GB SATA disks running at 7200 rpm.

Table 3 emphasizes that the performances with the RDF-3X system are comparable. Unsurprisingly, the two other systems are slower than RDF-3X on Queries #1 and #14. A fact which has been highlighted on many other evaluations. Note that these queries have different characteristics since they respectively correspond to large input with high selectivity, complex 'triangle' query pattern and large input with low selectivity. Query #2 is performed more rapidly by Jena TDB and BigOWLIM but WaterFowl is faster than RDF-3X. We consider that this is due to a better consideration of this query particular pattern. Note that queries #4 and #5 are involving inferences and demonstrates the good performances of WaterFowl due to its integrated support for reasoning.

5. CONCLUSION

We have presented WaterFowl, an RDF Store based on an in-memory, compact, self-indexed approach. The query processor we have detailed in this paper already provides interesting performance results for simple SPARQL queries as well as ones requiring inferences. In future work, we would like to introduce a full-featured text search engine for support regular expressions present in SPARQL queries. Moreover, we aim to transform the current unmutable version of WaterFowl into a database that accepts updates operations. Finally, to fulfill our big data vision, we are working a distributed version of our system.

6. REFERENCES

[1] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50, 2010.

[2] O. Curé, G. Blin, D. Revuz, and D. C. Faye. Waterfowl: A compact, self-indexed and inference-enabled immutable rdf store. In *ESWC*, pages 302–316, 2014.

[3] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[4] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[5] S. Harris and A. Seaborne. SPARQL 1.1 query language W3C recommendation. http://www.w3.org/tr/sparql11-query/, 2013.

[6] P. Hayes. RDF semantics, W3C recommendation. http://www.w3.org/tr/rdf-mt/, 2004.

[7] G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.

[8] J. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.

[9] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.

[10] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[11] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for sparql. In *EDBT*, pages 324–335, 2012.

[12] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.