SuccinctEdge: A Succinct RDF Store for Edge Computing

Weiqin Xu
UPEM LIGM - UMR CNRS
Marne-la-Vallée, France.
weiqin.xu@u-pem.fr

Olivier Curé
UPEM LIGM - UMR CNRS
Marne-la-Vallée, France.
olivier.cure@u-pem.fr

Philippe Calvez
CSAI ENGIE Lab
Stains - FRANCE
philippe.calvez1@engie.com

ABSTRACT

As edge computing is becoming a new platform for rich applications and services, it becomes more and more important to design adapted data management systems for this environment. In this paper, we present a prototype corresponding to a compact, in-memory RDF store that can answer SPARQL queries requiring reasoning services without necessitating any decompression. This demonstration highlights a design based on succinct data structures, shows some implementation details and provides encouraging performance measures over a set of real-world and synthetic data and query sets.

PVLDB Reference Format:
W. Xu, O. Curé, P. Calvez. SuccinctEdge: A Succinct RDF Store for Edge Computing. *PVLDB*, 13(12): 2857-2860, 2020.
DOI: https://doi.org/10.14778/3415478.3415493

1. INTRODUCTION

Edge computing is a processing paradigm that brings computation and data storage closer to the location where it is needed. It is a growing trend that masks cloud computing outages and enables the design of highly local context aware and responsive services. A main challenge for this environment is data management in the context of mobile devices and sensors as they generally have stringent requirements on energy consumption as well as memory and CPU usages.

Our prototype system, SuccinctEdge[1], has been designed for edge computing from the get go and tackles the RDF data model. It favors a compressed, self-indexed storage approach to a solution based on multiple indexes that could potentially improve query execution but at the cost of a higher memory footprint. The applications we are targeting with SuccinctEdge are executed on devices located at the edge of a network, streaming in nature, *i.e.*, an unbounded dataset of RDF graphs is queried by continuous SPARQL queries, and generally do not have to be persisted in secondary storage. Moreover, our system makes an extensive use of succinct data structures (SDS)[5], a family of data structures that uses a compression rate close to theoretical optimum, but simultaneously allowing efficient decompression-free query operations on the compressed data. Together with our self-indexed approach, SDS guarantees a low memory footprint that fits with an in-memory store approach. Moreover, the decompression-free aspects tends to reduce the number of CPU cycles on standard queries and inferences. Several recent data management systems have been designed with a succinct perspective, *e.g.*, Succinct[2] and ZipG[4]. Nevertheless, to the best of our knowledge, SuccinctEdge is the first one specifically designed for edge computing. In the evaluation section, we emphasize that our prototype provides interesting performance measures compared to state of the art RDF stores designed for the edge. These features will be highlighted during the demonstration.

[1]https://github.com/xwq610728213/SuccinctEdge

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415493

2. SYSTEM OVERVIEW

This section presents an encoding scheme, *i.e.*, LiteMat, for the different entities of the RDF data model (including its ontology layer), the architecture of the system and the main characteristics of the query processing engine.

2.1 LiteMat

LiteMat[3] is a semantic-aware encoding scheme that compresses RDF data sets and supports reasoning services associated to the RDFS ontology language, *e.g.*, inferences associated to the `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf` and `rdfs:subPropertyOf` constructors. To address inferences drawn from these predicates, we assign specific numerical identifiers to ontology terms which are organized as hierarchies, *i.e.*, concepts and predicates. This is performed by prefixing the encoding of a term with the encoding of its direct parent. This approach only works if an encoding is computed using a binary representation and all binary encoding entries are all of the same length. The encoding is performed using a top-down approach, *e.g.*, starting from the most specific concept of the hierarchy (*e.g.*, `owl:Thing` for the concept hierarchy), until all leaves are processed. Then a normalization is performed to guarantee that all encoding entries have the same length, *i.e.*, by setting rightmost bits to 0.

In Figure 1, we consider a small ontology extract containing the following axioms: $A \sqsubseteq Thing$, $B \sqsubseteq Thing$, $C \sqsubseteq B$ and $D \sqsubseteq B$. Figure 1a highlights the top-down encoding approach with (1) setting the local identifier of $Thing$, (2) its direct subconcepts (A and B) and B's subconcepts in (3).

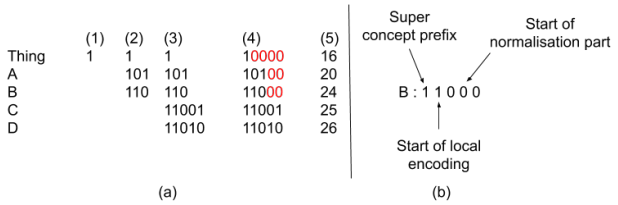

Figure 1: LiteMat encoding example

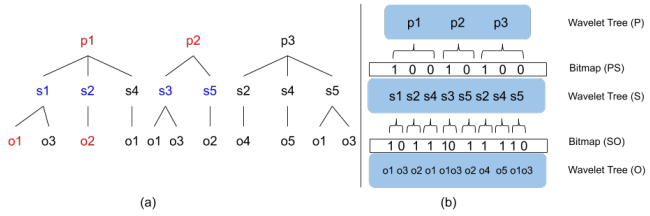

Figure 2: TripleStore example

Then, in (4) the normalisation step is performed, *i.e.*, added right-most bits are written in red. Column (5) provides the integer value attributed to each concept.

The mapping between URIs and their identifiers are stored in dictionaries to enable the so-called locate and extract operations. Moreover, in the URI to identifier dictionaries, additional identifier metadata are stored. For instance, the local length (binary length before the normalization phase) of each dictionary entry is stored along the final identifier entry. Figure 1b emphasizes some metadata of the LiteMat encoding for the B concept: super concept identifier part, start of local encoding and start of the normalization part.

The semantic encoding of concepts and predicates supports reasoning services usually required at query processing time. For instance, consider a query asking for the pressure value of sensors of type `S1`. This would be expressed as the following two triple patterns: `?x pressureValue ?v. ?x type S1`. In the case sensor concept `S1` has n sub-concepts, then a naive query reformulation requires to run the union of n+1 queries. With LiteMat's semantic-aware encoding, we are able, using two bit-shift operations and an addition, to compute the identifier interval, *i.e.*, [lowerBound, upperBound), of all direct and indirect sub-concepts of `S1`. Thus, the following reformulation is required (i) replacing the concept `S1` with a new variable : `?x type ?newVar` and (ii) introducing a filter clause constraining this new variable: `FILTER (?newVar>=lowerBound && ?newVar<upperBound)`.

2.2 Architecture

An overview of SuccinctEdge's architecture is presented in Figure 3. Like most RDF stores, all triples are encoded according to some dictionaries, ours are computed with LiteMat (see Section 2.1). The query engine uses these dictionaries to transform queries and translate their answer sets. The Triple store component adopts a single index based on the predicate, subject, object (PSO) triple permutation. This is motivated by the fact that the basic graph pattern of queries submitted to SuccinctEdge have predicates filled in with URIs (as opposed to variables). This component also highlights that we make a distinction between object (except `rdf:type`) and datatype properties. In the former, objects are individuals and thus encoded with the respective dictionary while in the later, objects are literals and stored as is. In terms of data structures, wavelet trees[5] are used for the property and subject layers as well as the object layer for object properties. In order to relate a wavelet tree of one layer to another, we are using bitmaps. Figure 2b represents the triple set of Figure 2a where a wavelet tree corresponds to balanced tree of bitmaps. For datatype properties, we are using a flat data format to store literals. Finally, triples containing a `rdf:type` property are stored in the RDFType store. These triples generally represent an important proportion of the triple set in real-world RDF datasets, so we proposed an efficient representation based on a vertical partitioning approach[1].

2.3 Query processing

The first step of our query processing approach is to define an order for the execution of the triple patterns (TP) contained in a SPARQL query. The query optimization generates left-deep plans using a cost-based approach together with a set of heuristics (inspired from [6]) which first considers TP with the smallest number of variables and define an order for TP containing the same number of variables, *e.g.*, $(s, p, ?o) > (?s, p, o)$. Once an order is predefined, SuccinctEdge translates the TPs into SDS's standard operations: access, rank and select. For a given SDS these three operations respectively return the value at a certain position, the number of occurrences of a value until a certain position and the index of n-th occurrence of a given value.

Due to space limitation, we only present two translation examples with Algorithms 1 and 2 for respectively TPs $(s, p, ?o)$ and $(?s, p, o)$.

Algorithm 1: Search the triple pattern $(s, p, ?o)$

Input: Predicate s,p
Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$id_s \leftarrow FindIdFromDictionary(s)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
for $index_s$ *in* $wt_s.rangeSearch(index_{sBegin}, index_{sEnd}, id_s)$ **do**
  $index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin}+1, 1)$;
  $index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
  for $index_o \leftarrow index_{oBegin}$ **to** $index_{oEnd}$ **do**
    $id_o \leftarrow wt_o[index_o]$;
    $res \leftarrow res \cup (id_s, id_p, id_o)$;
  end
end
return res;

In cases where reasoning services are necessary to provide an exhaustive answer set, we can replace $index_p$ with a continuous interval corresponding to a LiteMat interval.

TPs containing `rdf:type` are processed differently using the RDFType store component, where some simple structure look-ups permit to efficiently retrieve to subjects of a given concept or the concepts of a given subject.

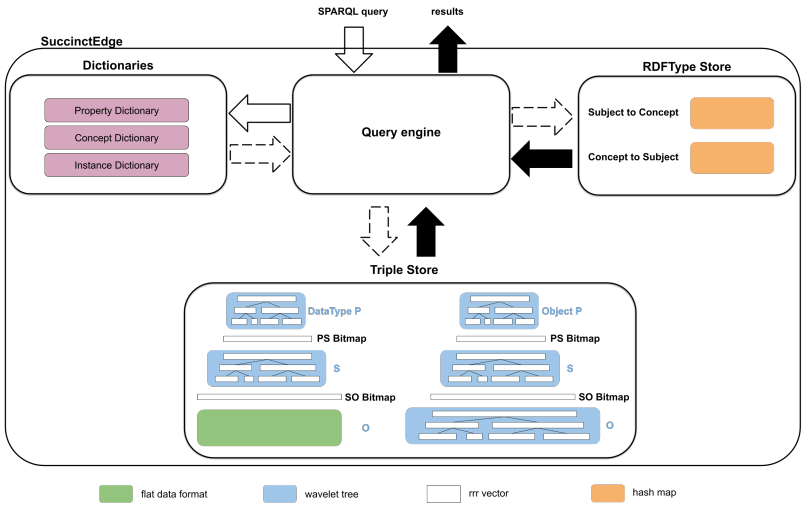

Figure 3: Architecture overview of SuccinctEdge

The next step corresponds to joining the results obtained from the execution of TPs. This occurs when different TPs share a common variable. One of our joining approach amounts to propagate variable assignments from one TP to another. Consider the triple set of Figure 2a and TPs $(?s, p1, o1)$ and $(?s, p2, ?o)$. The first TP gets the following assignments: $?s : \{s1, s2\}$ which will serve to dynamically generate $(s1, p2, ?o)$ and $(s2, p2, ?o)$ for the second triple.

During the join operation, we can benefit from a merge join in cases where the values assigned to a joining variable to the TP are in order. In the case of a star-shaped basic graph pattern (*e.g.*, $(?s, p1, o1)$ and $(?s, p2, ?o)$), thanks to the facts that all the subjects connected to a certain predicate are in order and that all the objects connected to one certain subject are also in order, we can perform a merge join on the subject variable.

Algorithm 2: Search the triple pattern $(?s, p, o)$

Input: Predicate p
Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
$index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
$index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
for $index_o$ *in* $wt_o.rangeSearch(index_{oBegin}, index_{oEnd}, id_o)$ **do**
  $index_s \leftarrow bitmap_{so}.rank(index_o + 1, 1) - 1$;
  $id_s \leftarrow wt_s[index_s]$;
  $res \leftarrow res \cup (id_s, id_p, id_o)$;
end
return res;

Previous executions steps are repeated until all the TPs have been processed. Then the answer set of the query is translated using our dictionaries and presented to the end-user or application.

3. EVALUATION

We have conducted an evaluation of SuccinctEdge over the following dimensions that seem particularly important in an IoT-oriented data management context: duration of structures creation (*i.e.*, generation of dictionaries and encoding the dataset), compression rate, query processing efficiency in both the presence and absence of reasoning services. The evaluation has been conducted on a Raspberry Pi 3B+ with 1GB of RAM and 16GB of microSSD HCI. SuccinctEdge is implemented in C++ and uses the SDS-lite library[2]. We are comparing SuccinctEdge against Apache Jena[3] TDB and RDF4Led[7] and results are presented in Table 1. A comparison against ZipG[4] does not make sense since it targets a distributed cloud setting and is not aiming for the RDF data model nor adopts a declarative query language. JenaTDB is an open-source compact and robust RDF store and RDF4Led has been designed specifically for edge computing. The experimentation uses one synthetic dataset based on the Lehigh University Benchmark[4] and a real-world dataset pertaining to the energy domain. The first one, denoted LUBM1, is composed of over 103.000 triples and is used to get significant measures on the duration, size and query with inference dimensions. The small dataset, denoted EnergySmall, is composed of 500 triples and matches practical retrieval data access from IoT sensors. It is used to evaluate non-inference queries.

Concerning the duration of data structures (*i.e.*, both the dictionaries and datasets), SuccinctEdge is 32% and 53.5% faster than respectively RDF4Led and Jena TDB. In our experimentation, we found out that, for SuccinctEdge, around 25% of total duration amounts to the dictionary creation which is only required when the ontology is updated. SuccinctEdge's dictionary requires a smaller memory footprint than the two other systems but the main gain is obviously witnessed on the data instance graph. Overall, SuccinctEdge's data structures is 6x and 28x more compact than resp. RDF4Led and Jena TDB. Note that all systems are

[2]https://github.com/simongog/sdsl-lite
[3]https://jena.apache.org/
[4]http://swat.cse.lehigh.edu/projects/lubm/

Table 1: Comparison of RDF Stores. Creation time and size evaluation performed over the LUBM1 dataset which contains over 100.000 triples. Query times in msec and sizes in MB. Q1, Q2 and Q3 performed over a graph of 500 triples. Q4 and Q5 are performed on LUBM1

	Creation time(sec)	Dict. size	Data size	Total size	Comp. rate	Q1	Q2	Q3	Q4	Q5
Jena TDB	18.5	2.6	12.5	15.1	86%	803	913	880	20351	7460
RDF4Led	12.8	1.9	5.2	7.1	41%	200	203	203	-	-
SuccinctEdge	8.6	1.6	0.28	1.9	11%	5	5	7.5	5400	702

more compact than the original dataset represented in turtle (instances) and RDF/XML (ontology) with SuccinctEdge requiring only 11% of LUBM1. Considering query processing, we have tested the three systems over of a set of queries that do not require any inferences and which basic graph pattern take the form of a chain, a snowflake and a star, respectively Q1, Q2 and Q3; a set of queries requiring some inferences on both the property and concept hierarchies (Q4 and Q5). Queries Q1 to Q3, performed over EnergySmall, highlight that SuccinctEdge is two orders of magnitude faster that the other two systems. For queries Q4 and Q5, Jena TDB and RDF4Led do not natively support any reasoning services. For Jena TDB, we implemented a query rewriting approach to generate a SPARQL query containing a union of basic graph patterns that cover all inferences. This rewriting is then submitted to Jena TDB. Again SuccinctEdge is more efficient (by one order of magnitude). This rewriting can not be executed on RDF4Led which currently does not support union SPARQL queries.

4. DEMONSTRATION SCENARIO

The demonstration setup consists of (i) a Raspberry Pi 3B+ (similar to the one used in the evaluation) on which all three evaluated systems (Apache Jena, RDF4Led and SuccinctEdge) have been installed and (ii) a laptop from which an end-user can interact with SuccinctEdge. These interactions correspond to selecting a dataset (including ontologies) among a set of synthetic and real-world graphs and registering a query, either selected from a set inference-based and inference-free predefined ones or end-user defined.

The selected graphs range from sizes of a couple of hundreds to one hundred thousands triples. The synthetic data sets are based on the WatDiv[5] and LUBM benchmarks while the real-world is extracted from measures obtained from sensors in the resource management domain (*i.e.*, potable water network with pressure, flow, pH, etc. measures). These graphs are submitted to SuccinctEdge using a streaming approach where a stream corresponds to a complete graph. Several metrics are displayed to the audience. They include the performance of each sub-tasks of query processing as well as the graph construction in terms of space and time. In order to grasp the mechanisms involved in querying our SDS data structures, the query plan selected by the query optimizer is displayed in a detailed manner thanks to the `EXPLAIN ANALYZE` command. The rewriting of SPARQL queries in terms of a sequence of the access, rank and select SDS operations is also displayed to the audience. Moreover, it highlights the inference-oriented query reformulation.

[5]https://dsg.uwaterloo.ca/watdiv/

Using this setting, we demonstrate the efficiency of SuccinctEdge against typical RDF stores found in edge computing.

5. CONCLUSION

In this demonstration, we showcase a frugal in-memory RDF data store designed for edge computing which privileges a small memory footprint over multiple indexing structures. The compactness of the stored data is obtained by using Succinct Data Structures such as bitmaps and wavelet trees. We demonstrated a translation of SPARQL queries into operations associated to these SDS, namely access, rank and select, and thus support a decompression-free execution of these queries. Moreover, the usage of the LiteMat encoding scheme enables to perform standard RDFS-based reasoning services over these queries, *i.e.*, it supports the inference of implicit consequences from explicit knowledge.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 411–422. VLDB Endowment, 2007.

[2] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *Symposium on Networked Systems Design and Implementation, NSDI 2015*, pages 337–350, 2015.

[3] O. Curé, W. Xu, H. Naacke, and P. Calvez. Litemat, an encoding scheme with RDFS++ and multiple inheritance support. In *The Semantic Web: ESWC 2019 Satellite Events - Revised Selected Papers*, pages 269–284, 2019.

[4] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. ZipG: A memory-efficient graph store for interactive queries. In *ACM International Conference on Management of Data, SIGMOD*, pages 1149–1164, 2017.

[5] G. Navarro. Wavelet trees for all. *J. of Discrete Algorithms*, 25:2–20, Mar. 2014.

[6] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for SPARQL. In *International Conference on Extending Database Technology, EDBT , Proceedings*, pages 324–335, 2012.

[7] A. L. Tuán, C. Hayes, M. Wylot, and D. L. Phuoc. RDF4Led: an RDF engine for lightweight edge devices. In *International Conference on the Internet of Things*, pages 2:1–2:8, 2018.