Efficient dictionary compression for processing RDF big data using Google BigQuery

Efficient Dictionary Compression for Processing RDF Big Data Using Google BigQuery

Omer Dawelbeit
School of Systems Engineering,
University of Reading,
United Kingdom.
Email: o.i.o.dawelbeit@pgr.reading.ac.uk

Rachel McCrindle
School of Systems Engineering,
University of Reading,
United Kingdom.
Email: r.j.mccrindle@reading.ac.uk

***Abstract*—The Resource Description Framework (RDF) data model, is used on the Web to express billions of structured statements in a wide range of topics, including government, publications, life sciences, etc. Consequently, processing and storing this data requires the provision of high specification systems, both in terms of storage and computational capabilities. On the other hand, cloud-based big data services such as Google BigQuery can be used to store and query this data without any upfront investment. Google BigQuery pricing is based on the size of the data being stored or queried, but given that RDF statements contain long Uniform Resource Identifiers (URIs), the cost of query and storage of RDF big data can increase rapidly. In this paper we present and evaluate a novel and efficient dictionary compression algorithm which is faster, generates small dictionaries that can fit in memory and results in better compression rate when compared with other large scale RDF dictionary compression. Consequently, our algorithm also reduces the BigQuery storage and query cost.**

I. Introduction

The Resource Description Framework (RDF) [1] is recommended by the W3C for representing information about resources in the World Wide Web. RDF is intended for use cases when information need to be processed by and exchanged between applications rather than people. Resources in RDF can be identified by using Uniform Resource Identifiers (URIs), for example the following URI is used to represent an employee `smithj` in an organisation:

⟨http://inetria.org/directory/employee/smithj⟩

Unknown resources or resources that do not need to be explicitly identified are called blank nodes. Blank nodes are referenced using an identifier prefixed with an underscore such as **`_:nodeId`**. Constant values such as strings, dates or numbers are referred to as literals. Information about a particular resource is represented in a statement, called a triple, that has the format $(subject, predicate, object)$ abbreviated as (s, p, o), moreover, the three parts of the triple are also called *terms*. Subject represents a resource, either a URI or a blank node, predicate represents a property linking a resource to an object, which could be another resource or literal. Formally, let there be pairwise disjoint infinite sets of URIs (U), blanks nodes (B), and literals (L). An RDF triple is a tuple:

$$(s, p, o) \in (U \cup B) \times (U) \times (U \cup B \cup L) \quad (1)$$

RDF triples can be exchanged in a number of formats, primarily RDF/XML which is based on XML documents. Other formats include line based, plain text encoding of RDF statements such as N-Triples [2]. An example of an RDF statement in N-Triple format is shown in Fig 1.

When dealing with large datasets, URIs occupy many bytes and take a large amount of storage space, which is particularly true with datasets in N-Triple format that have long URIs. Additionally, there is increased network latencies when transferring such data over the network. Although Gzip compression can be used to compress RDF dataset, it is difficult to parse and query these datasets without decompressing them first, which imposes a computation overhead. There is, therefore, a need for a compression mechanism that maintain the semantic of the data, consequently, many large scale RDF systems such as BigOWLIM [3] and WebPIE [4] adopt dictionary encoding. Dictionary encoding encodes each of the unique URIs in RDF datasets using numerical identifiers such as integers that only occupy 8 bytes each.

The wide use of RDF data on the Web to represent government, publications, life sciences, geographic and social data has resulted in very large RDF datasets. Such datasets require RDF applications to store, query and process billions of statements and can hence be described as Big Data. A number of NoSQL Big Data models have been reviewed in the literature [5]. In particular document oriented stores such MongoDB and CouchDB in addition to wide columns stores such as Google BigTable and Cassandra [6], provide the ability to store petabytes of data. In exception to Google BigTable — which is a fully managed solution — aforementioned NoSQL representations may require additional management such as provisioning, index creation and maintenance.

In response to demands for cloud-based big data storage and analytical services, cloud services such Google BigQuery [7] provide the ability to store, process and interactively query massive datasets. Google BigQuery is defined as "fully managed, NoOps, low cost analytics database" and utilises columnar database technology [8]. Additionally, BigQuery provides SQL like query languages and REST APIs for the execution of queries and the management of data. BigQuery stores data in tables without indexes and charges for storage based on the size of the data stored and for queries by the size

```
<http://inetria.org/directory/employee/smithj> <http://xmlns.com/foaf/0.1/name> "John Smith" .
```

Fig. 1. Example RDF statement in N-Triple format

of the columns scanned. For example, the size of each cell in a column of type string will be 2 bytes + the UTF-8 encoded string size. BigQuery can be used for both the storage and processing of RDF without any upfront investments. However, due to the large number of long URIs in RDF datasets, the BigQuery cost involved when querying this data can rapidly increase.

In this paper we propose a novel and efficient dictionary compression algorithm which generates small dictionaries that can fit in memory and results in better compression rate when compared with other large scale RDF dictionary compression. Consequently, our algorithm also reduces the BigQuery storage and query cost. The rest of this paper is organised as follows: In Section II a brief review of the work related to RDF dictionary encoding is provided, then in Section III we explain and present our dictionary encoding algorithm. Subsequently, in Section IV experimental evaluation of our dictionary encoding algorithm is conducted using both real-world and synthetic datasets. Finally Section V concludes this paper and highlights possible future work.

II. Related Work

An RDF dictionary encoding approach based on MapReduces was proposed by [9]. The authors distribute the creation of the dictionary and the encoding of the data between a number of nodes running the Hadoop framework. The authors reported the compression of 1.1 billion triples of the LUBM [10] dataset in 1 hour and 10 minutes, with a 1.9 GB dictionary. Another dictionary encoding approach based on supercomputers was presented by [11]. This approach uses parallel processors to concurrently process a dictionary by using the IBM General Parallel File System (GPFS). The authors report the encoding of 1.3 billion triples of the LUBM dataset [12] in approximately 28 minutes by utilising 64 processors. The total size reported for the dictionary is 23 GB and 29.8 GB for the encoded data.

A comparison of RDF compression approaches is provided by authors in [13]. The authors compare three approaches, mainly gzip compression, adjacent lists and dictionary encoding. Adjacent lists concentrates the repeatability of some of the RDF statements and achieves high compression rates when the data is further compressed using gzip. The authors also show that datasets with a large number of URIs that are named sequentially can result in a dictionary that is highly compressible. A dictionary approach for the compression of long URI References (URIRefs) in RDF/XML documents was presented by [14]. The compression is carried out in two stages, firstly the namespace URIs in the document are dictionary encoded using numerical IDs, then any of the URIRefs are encoded by using the URI ID as a reference. In this setting, the authors create two dictionaries, one for the URIs and another one for the URIRefs. The encoded data is then compressed further by using an XML specific compressor. Although this approach shows compression rates that are up to 39.5% better than Gzip, it is primarily aimed at compacting RDF/XML documents rather than providing an encoding that reduces both the size and enables the data to be processed in compressed format.

III. RDF Compression Using Dictionary Encoding

The RDF dictionary encoding approaches surveyed in this paper mainly focused on replacing the long URIs to integer identifiers with the aim to reduce the size of the data. However, very minimal considerations are given to the size of the generated dictionaries, which in most cases contain a large level of redundancy due to the shared namespaces and URI path parts. This is clearly demonstrated with the large-scale MapReduce dictionary compression [9], in which case, generated dictionaries are larger than 1 GB for large datasets. Additionally the approach followed by [12] uses parallel computations to create the dictionary for LUBM, which was many times larger at 23 GB. This results in the dictionary generation and data encoding processes taking longer to complete as these large dictionaries need to be accessed on disk and can not be held in memory in their entirety. In the following sections we present our novel and efficient dictionary encoding algorithm.

A. Reducing the Size of URIs

BigQuery supports 64-bit signed integers, with the most significant bit reserved as a sign bit, the minimum number that can be stored is -2^{63} and the maximum is $2^{63}-1$. To highlight the potential savings when converting URIs from strings to integers, consider the following URIs and URI References (URIRefs):

1) *http://www.w3.org/1999/02/22-rdf-syntax-ns#type*
2) *http://inetria.org/directory/schema#Employee*
3) *http://inetria.org/directory/schema#Company*
4) *http://inetria.org/directory/employee/smithj*
5) *http://inetria.org/directory/employee/doej*

The aforementioned URIs contain on average 44 characters per URI, with UTF 8 encoding each character can consume from 1 to 4 bytes for storage. On average, this equates to a minimum of 46 bytes for each URI, with integer encoding, only 8 bytes are required to store each value, equating to a storage saving of at least 82%. With this evident savings on both storage and cost, a straightforward dictionary encoding approach similar to [9] can be employed. However, due to redundancy one of the major issues with such approach is the size of the dictionary, which prohibit in-memory handling of the dictionary and instead resorts to in-file processing, resulting in a considerable processing overhead.

In this section we propose an efficient dictionary encoding approach to reduce the size of the dictionary and therefore be able to load it in memory to speedup the encoding process. It is

worth noting that we do not compress literal values, according to Formula 1 a literal can not occur on the subject or predicate term, they only occur on the object term. As noted earlier that URIs are identifiers for resources, which are usually queried as a whole, however with literals we can use the BigQuery string functions to be able to search for particular text, as such we do not compress them. When storing the encoded data in BigQuery we use a table called the *triple table* with three columns (predicate, subject, object) of integer type instead of string, these will hold the encoded values of the original URIs and blank nodes. A fourth column *object_literal* of type string is added to store any literals as they are, in which case the *object* column will contain a *NULL* value, an example is shown in Table I below.

TABLE I
ENCODED EXAMPLE OF THE BIGQUERY *triple table*

Subject	Predicate	Object	Object_Literal
345	50	250	
345	300	NULL	"John Smith"

B. Efficient URI Reference Compression

In this section, we propose an approach for compressing URI in NTriple documents using integer values to build up an efficient (compact) dictionary. If we consider, for example the URIs numbers 4 and 5 in the previous list, it can be seen that they both share the `http://inetria.org/directory/employee` URI. In a large dataset many more terms will share similar URIs, which, in a straightforward dictionary encoding the dictionary will contain the URI many times, thus inflating the dictionary with redundancy. If we split each of the URIs into two sections, the first section contains the hostname and part of the path (e.g. `inetria.org/directory/employee`) and the second section contains the last part of the path (e.g. `smithj` and `doej`). The dictionary will only contain the following sections: `inetria.org/directory/employee`, `smithj` and `doej`, instead of the full URIs in 4 and 5 in the previous list.

It can be seen that this approach can reduce the size of the dictionary, noting that we have removed the scheme (e.g. http(s)://) as this can be added later when decoding the data. URIs that belong to the RDF and RDF Schema (RDFS) [15] vocabulary such as item 1 in the previous list (`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`) are limited and are hence encoded in their entirety — using a set of fixed integer values — without being split.

C. Encoding Terms

To reduce redundancy in the dictionary, we split the URIs on the last path separator "/", which means most encoded URIs will now have two parts, both are encoded as integers. A question that begs asking, is how are these two integer values stored together?, one approach we utilise in this paper

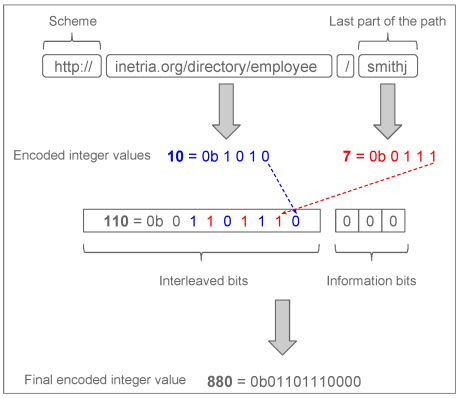

Fig. 2. Dictionary Encoding Using Binary Interleaving

TABLE II
INFORMATION BITS FOR URI REFERENCE COMPRESSION

First bit - parts bit (least significant bit)	0	URI has two parts (e.g. http://inetria.org/directory)
	1	URI has one part (e.g. http://inetria.org or blank node _:jA5492297)
Second bit - scheme bit	0	Scheme is http
	1	Scheme is https
Third bit - slash bit	0	URI does not have slash (/) at the end (e.g. http://inetria.org)
	1	URI has a slash at the end (e.g. http://inetria.org/)

is by using a "bitwise" pairing function [16]. Such a function is based on bit interleaving of two number so they can be joined into one with the reverse process extracting back the two numbers. Fig. 2 shows an example of encoding the URI `http://inetria.org/directory/employee/smithj` using our dictionary encoding algorithm. As seen from the example, first we remove the scheme, then split the remaining part of the URI at the last path separator. This gives us two strings that we encode using incremental integer values (e.g. 10 and 7). These two values are then paired by interleaving their bits resulting in the decimal number 110. Finally, three least significant information bits (summarised in Table II) are added to store information about the original URI, resulting in a final encoded decimal of 880.

D. Decoding Terms

Pairing is only used when a term is a URI with more than one section. Terms that do not contain multiple sections such as blank nodes are encoded into integer values with the additional information bits added. When decoding integer values, we first extract the three lease significant information bits, if the *parts* bit is 1, then the integer value after removing these three bits can immediately be resolved from the dictionary. If the *parts* bit is 0 then we un-pair the integer values of the two sections by restoring their bits, the resultant values are then retrieved from the dictionary and concatenated. The *scheme* bit determines if *http* or *https* should be added to the decoded string, similarly, the *slash* bit is used to add a slash "/" to the

end of the URI. Finally the NTriple angle brackets "⟨" and "⟩" are added to the URI to complete the term decoding.

E. Storage Considerations

As noted earlier, BigQuery supports 64-bit signed integers, the minimum number that can be stored is -2^{63} and the maximum is $2^{63}-1$. Moreover, we reserve additional three bits for storing information regarding the encoded URI, this leaves us with a minimum and maximum between -2^{60} and $2^{60}-1$. For our dictionary encoding approach, we might need to perform bit interleaving for encoding URIs that are split into two sections. Bit interleaving will result in a number that contains the sum of the bits in the two joined numbers, consequently, in order to ensure that the bit interleaved numbers do not exceed the maximum of $2^{60}-1$. Each of the joined sections must be between -2^{30} and $2^{30}-1$, giving us the ability to encode string sections using integer values that ranges from $-1,073,741,824$ to $1,073,741,823$. With the dictionary using both positive and negative integers within this range, we can encode approximately 2.14 billion unique URI sections.

IV. Evaluation

The dictionary encoding is carried out in three steps, firstly we process all the dataset files and extracts all the URI parts for the dictionary keys, secondly the dictionary is assembled in memory, thirdly the original dataset file are encoded by loading the dictionary in memory. Both the algorithm that extract the URI parts and the one that encodes the data can be distributed. However, the dictionary assembly algorithm need to be executed on one computer with sufficient memory to build the dictionary in memory. The size of the memory of this computer is proportionate to the size of the extracted URI parts files. We have implemented our dictionary encoding algorithms using the CloudEx[1] cloud-based task execution framework based the Java programming language and ran the experiments on the Google Cloud Platform [17].

For our experiments we have used two real world datasets, SwetoDblp [18], which is a collection of computer science bibliography data and DBpedia [19], which is a collection structured data extracted from Wikipedia. We also generate synthetic data using the LUBM [10] artificial benchmark tool that generates universities data. We have generated 8,000 (8K) universities with a total of 1,092,030,000 statements. We have experimented with three strategies for splitting the URIs into two parts before encoding, these can be summarised as follows:

- **Standard**, the URIs are split on the last path separator "/", this is our default strategy.
- **Hostname**, the URIs are split on the first path separator "/" after the hostname.
- **Hostname + first path part (1stPP)**, the URIs are split on the second path separator "/" after the hostname.

[1]http://cloudex.io

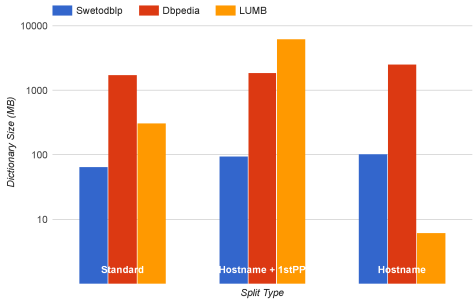

Fig. 3. Comparison of dictionary size when using Standard, Hostname + first path part (1stPP) and Hostname split strategies.

We evaluate the size of the dictionary for each of these strategies, additionally, if the dictionary assembly process is memory intensive we also report the memory usage pattern. Moreover, we show the BigQuery savings achieved by our dictionary encoding in terms of scanned data, which is the primary cost factor when using BigQuery. Finally, we evaluate our dataset compression rate and dictionary sizes against other large-scale distributed RDF dictionary encoding. The compression rate is calculated similar to [9] as follows:

$$rate = \frac{originalSize}{gzippedEncodedSize + gzippedDictionarySize}$$

A. URIs Split Strategy

Table III provides a comparison of the three URIs split strategies, additionally Fig. 3 shows the size of the dictionary for each of the three strategies. As it can be seen that our *Standard* split strategy provides the smallest dictionary for real-world datasets (Swetodblp, DBpedia) compared to the other two split strategies and consequently requires the least memory usage (as evident from the DBpedia dataset). In contrast, the *Hostname* strategy provides the best dictionary size for the LUBM dataset, unsurprisingly, due to the uniformity of this synthetic dataset. The *Hostname* strategy, which provides the largest dictionary for both Swetodblp and DBpedia, seems to provide a very small dictionary for LUBM of only 6 MB. This shows that each dataset has its own characteristics and the split strategy that provides the smallest dictionary, which can be different from one dataset to another.

Additionally, as shown in Fig. 3 our *Standard* split strategy provides the second best dictionary size at 314 MB. On the other hand, the *Hostname + 1stPP* is the worst strategy for LUBM, resulting in a very large dictionary with a size of 6.1 GB. Although an n1-highmem-4 Virtual Machine (VM) with 26 GB of memory was enough to assemble the DBpedia dictionary, we needed an n1-highmem-8 VM with 52 GB of memory to assemble this large LUBM dictionary.

B. Dictionary Assembly Memory Footprint

In terms of the memory footprint of the dictionary assembly task, we have analysed the usage of the memory intensive dictionaries for both the DBpedia and LUBM datasets. As shown in Table III, these include the three strategies for the DBpedia dataset and the *Hostname + 1stPP* for strategy LUBM. The memory usage patterns are shown in Fig. 4 for

TABLE III
COMPARISON OF STANDARD, HOSTNAME + FIRST PATH PART (1ST PP) AND HOSTNAME DICTIONARY SPLIT STRATEGIES FOR SWETODBLP, DBPEDIA AND LUBM DATASETS.

	Swetodblp			DBpedia			LUBM		
	Stand.	Host. +1stPP	Host.	Stand.	Host. +1stPP	Host.	Stand.	Host. +1stPP	Host.
Extract Files Size (MB)	81	103	104	4,031	4,118	4,610	317	5,285	14
No. Extract URI Part	3,738,132	3,788,657	3,791,842	165,119,090	166,856,992	176,000,886	5,857,835	109,218,289	282,937
No. Dict. Keys	2,494,280	2,654,293	2,655,787	53,551,014	55,784,063	66,128,766	5,769,284	109,168,015	169,646
Dictionary Size (MB)	64	97	103	1,760	1,877	2,537	314	6,325	6
Max Mem. (GB)	0.5	0.5	0.5	12	12	13	1	31	0.5
Assemble Time (s)	6	6	6	510	600	820	36	889	8

TABLE IV
DICTIONARY ENCODING METRICS, SUCH AS ENCODED DATA AND DICTIONARY SIZES, COMPRESSION RATE AND BIGQUERY SCANNED BYTES IMPROVEMENTS FOR SWETODBLP, DBPEDIA AND LUBM.

Dataset	Size (GB)	Raw enc. data (GB)	Raw dict. (GB)	Enc. data gzip (GB)	Dict. gzip (GB)	Comp. rate	Query scan (GB)	Impr. (%)
Swetodblp	4.62	0.88	0.06	0.13	0.02	29.93	0.27	94.08
DBpedia	47	13.8	1.72	4.48	0.91	8.72	6.26	86.68
LUBM	169	31.4	0.31	6.16	0.03	27.29	22	86.98

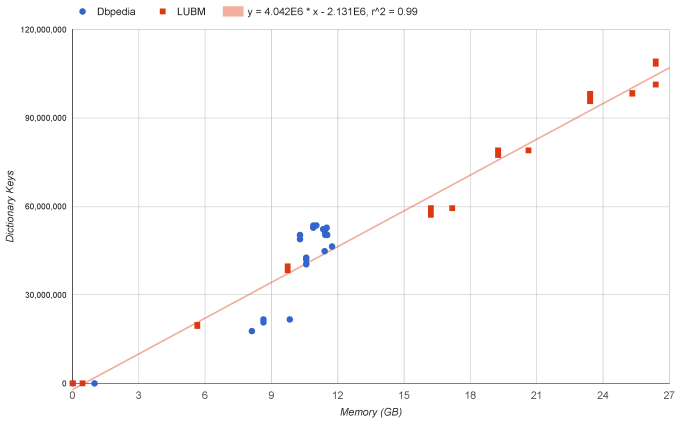

Fig. 4. Dictionary assembly memory usage for DBpedia and LUBM datasets.

both datasets with a regression line showing the relationship between the number of dictionary keys and the dictionary assembly memory usage.

It is worth noting that the maximum allocated Java Virtual Machine (JVM) memory needs to be more than the memory requirements for the dictionary assembly. For example for DBpedia dataset the dictionary required 12GB of memory, whilst the maximum JVM memory in this case is set to 23GB and the actual VM has a maximum of 26GB of main memory. For the LUBM worst case strategy (*Hostname + 1st PP*), the dictionary memory usage is 31GB, whilst the JVM memory is set to 50GB on a VM with 52GB of main memory. We have noticed that if the dictionary assembly uses most of the JVM memory, garbage collection occurs more frequently and the assembly process slows down considerably. Moreover, if the JVM is assigned more memory than the system can accommodate this can result in the crash of the JVM.

From Fig. 4 we can estimate the memory usage for the dictionary assembly based on the number of dictionary keys using the linear regression formula shown on the Figure. For example, given that the maximum memory that can currently be allocated to a VM on the Google Cloud Platform is 208GB (using an n1-highmem-32 VM), we can readily assemble very large dictionaries with more than 700 million keys.

C. BigQuery Scanned Bytes Improvements

The key motivation of implementing our dictionary encoding process, is to reduce the size of the total data stored in BigQuery and scanned when querying the data, which in turn will reduce cost. Table IV shows the dictionary encoding metrics such as the original, encoded and gzip encoded data and dictionary sizes for the Swetodblp, DBpedia and LUBM datasets. The "Size (GB)" column shows the original size of the dataset before encoding. Without encoding, BigQuery will scan the full dataset when executing queries that select all columns in the *triple table* (Table I). On the other hand, when the data is encoded, BigQuery will only scan the size shown in the "Query scan (GB)" column. As shown, our dictionary compression achieves considerable improvements and subsequently BigQuery cost reduction. The percentages of these improvements are 94.08%, 86.68% and 86.98% for the Swetodblp, DBpedia and LUMB datasets respectively.

D. Comparison of Dictionary Encoding

Table V provides a comparison of our LUBM 8K and DBpedia datasets compression rate and dictionary sizes against

TABLE V
COMPARISON OF LARGE SCALE RDF DICTIONARY ENCODING.

Work	Dataset	Size (GB)	Comp. rate	Encoded Data size (GB)		Dictionary size (GB)	
				Raw	Comp.	Raw	Comp.
Urbani et al. [9]	LUBM 8k	158.9	10.02	-	14	-	1.9
Weaver. J [12]	LUBM 10k	224	-	29.8	-	23	-
Our approach	LUBM 8k	169.6	27.29	31.4	6.16	0.31	0.03
Urbani et al. [9]	DBPedia (110M)	17.4	7.33	-	1.4	-	1
Our approach	DBpedia (322M)	47	8.72	13.8	4.48	1.72	0.91

results reported by [9], [12]. In terms of LUBM, our approach provides the smallest dictionary and achieves better compression rate. Additionally, for DBpedia our dictionary and compression rate are better than those reported by [9], this is considering that we use a newer version of the dataset which is three times larger than the version used by the authors, despite this, our compressed dictionary size is smaller. Due to the efficiency of our dictionary encoding the runtime for the overall process to import the LUBM 8K data into BigQuery is 18 minutes and 40 seconds (including 4 minutes to import the data into BigQuery) using 16 n1-standard-2 2 cores with 7.5 GB RAM virtual machines. This outperforms the time taken by the approaches developed by [9], [12], which report dictionary encoding times of 1 hour 10 minutes (on a 32 nodes Hadoop cluster) and 27 minutes 11 seconds (on 64 cores supercomputer) respectively.

V. CONCLUSION AND FUTURE WORK

In this paper we have presented and evaluated a novel and efficient dictionary compression algorithm. We have shown that our algorithm generates small dictionaries that can fit in memory and results in better compression rate when compared with other large scale RDF dictionary compression. The ability to fit the dictionary in memory results in a faster encoding and decoding process. Additionally, it can easily enable applications to hold the entire dictionary in memory whilst encoding or decoding user queries before sending them to BigQuery. We have also shown that our algorithm reduces the both the BigQuery storage and query cost.

As future work we plan to extend our experimentation to cover more real-world datasets and to experiment with larger synthetic dataset of more than 1 billion statements. We also plan to add the ability for user to query the encoded dataset in BigQuery by using the using the Protocol and RDF Query Language (SPARQL) [20].

ACKNOWLEDGMENT

We would like to thank Google Inc. for providing us with credits to run experiments on the Google Cloud Platform.

REFERENCES

[1] G. Klyne et al., "Resource Description Framework (RDF): Concepts and Abstract Syntax," *W3C Recommendation*, 2004.

[2] J. Grant and D. Beckett, "RDF Test Cases," *W3C Recommendation*, 2004, available at http://www.w3.org/TR/2004/REC-rdf-testcases-20040210. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-testcases-20040210

[3] B. Bishop et al., "OWLIM: A family of scalable semantic repositories," *Semantic Web*, vol. 2, no. 1, pp. 33–42, 2011.

[4] J. Urbani et al., "WebPIE: A Web-scale Parallel Inference Engine using MapReduce," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 10, pp. 59–75, jan 2012.

[5] S. Sharma, et al., "A brief review on leading big data models," *Data Science Journal*, vol. 13, no. December, pp. 138–157, 2014.

[6] S. Sharma, "An extended classification and comparison of nosql big data models," *CoRR*, vol. abs/1509.08035, 2015. [Online]. Available: http://arxiv.org/abs/1509.08035

[7] G. Developers, "Bigquery - large-scale data analytics," https://cloud.google.com/products/bigquery/, 2015, [Online; accessed 13-December-2015]. [Online]. Available: https://cloud.google.com/products/bigquery/

[8] D. Abadi et al., "Column-oriented database systems," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1664–1665, 2009.

[9] J. Urbani and J. Maassen, "Scalable RDF data compression with MapReduce," *Concurrency and Computation: Practice and Experience*, no. April 2012, pp. 24–39, 2013.

[10] Y. Guo et al., "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, no. 610, pp. 1–34, 2005.

[11] J. Weaver and J. Hendler, "Parallel materialization of the finite rdfs closure for hundreds of millions of triples," *The Semantic Web-ISWC 2009*, pp. 682–697, 2009.

[12] J. Weaver, "Toward webscale, rule-based inference on the semantic web via data parallelism," Ph.D. dissertation, Rensselaer Polytechnic Institute, Feb. 2013.

[13] J. Fernández, "RDF compression: basic approaches," *Proceedings of the 19th international conference on World wide web*, pp. 3–4, 2010.

[14] K. Lee et al., "Web document compaction by compressing URI references in RDF and OWL data," *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pp. 163–168, 2008.

[15] D. Brickley and R. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema," pp. 1–15, 2004.

[16] S. Pigeon, "Contributions to data compression," Ph.D. dissertation, Université de Montréal, Dec. 2001.

[17] G. Developers, "Google cloud computing, hosting services & apis," https://cloud.google.com/, 2015, [Online; accessed 13-December-2015]. [Online]. Available: https://cloud.google.com/

[18] B. Aleman-Meza et al., "SwetoDblp ontology of Computer Science publications," *Web Semantics*, vol. 5, no. 3, pp. 151–155, 2007.

[19] J. Lehmann et al., "DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[20] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," *W3C Recommendation*, vol. 2009, no. January, pp. 1–106, 2008. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/