Efficient Parallel Dictionary Encoding for RDF Data

Long Cheng[1][2][3], Avinash Malik[4], Spyros Kotoulas[2], Tomas E Ward[1], Georgios Theodoropoulos[5]

[1] National University of Ireland Maynooth, Ireland [2] IBM Research, Ireland
[3] Technische Universität Dresden, Germany [4] University of Auckland, New Zealand [5] Durham University, UK

long.cheng@tu-dresden.de, avinash.malik@auckland.ac.nz
spyros.kotoulas@ie.ibm.com, tomas.ward@eeng.nuim.ie, theogeorgios@gmail.com

ABSTRACT

The Semantic Web comprises enormous volumes of semi-structured data elements. For interoperability, these elements are represented by long strings. Such representations are not efficient for the purposes of Semantic Web applications that perform computations over large volumes of information. A typical method for alleviating the impact of this problem is through the use of compression methods that produce more compact representations of the data. The use of dictionary encoding for this purpose is particularly prevalent in Semantic Web database systems. However, centralized implementations present performance bottlenecks, giving rise to the need for scalable, efficient distributed encoding schemes. In this paper, we describe a straightforward but very efficient encoding algorithm and evaluate its performance on a cluster of up to 384 cores and datasets of up to 11 billion triples (1.9 TB). Compared to the state-of-art MapReduce algorithm, we demonstrate a speedup of $2.6 - 7.4\times$ and excellent scalability.

1. INTRODUCTION

The Semantic Web possesses plenty of special characteristics not available with the traditional web, such as amenability to machine processing, information lookup and knowledge inference. This web is founded on the concept of Linked Data [4], a term used to describe the practices of exposing, sharing and connecting information on the web using recent W3C specifications such as RDF and URIs. As Linked Data increasingly exposes data from multiple domains, such as general knowledge (DBpedia [3]), bioinformatics (Uniprot [2]), and GIS (linkedgeodata [15]), the potential for new knowledge synthesis and discovery increases immensely. Capitalizing on this potential requires semantic web applications which are capable of integrating the information available from this rapidly expanding web. The web engineering challenges are currently pushing computing boundaries at exascale and beyond.

This web is build on the W3C's Resource Description Framework (RDF) - a schema-less, graph-based data format which describes the Linked Data model in the form of subject-predicate-object (SPO) expressions based on the statement of resources and their relationships. These expressions are known as RDF triples. For example, the simple statement from DBpedia (<dbpedia:IBM>, <dbpedia-owl:foundation-Place>, <dbpedia:New-York>) conveys the information that the corporation IBM was founded in New York. The Semantic Web already contains billions of such statements and this number is growing rapidly. As the terms in a RDF statement consist of long string characters in the form of either URIs or literals, storing and retrieving such information directly on an underlying database namely a triple store will result in (1) unnecessarily high disk-space consumption and (2) poor query performance (querying on strings is computationally intensive).

Dictionary encoding has been shown to be an efficient way to ameliorate these problems. Using dictionary encoding all the terms are replaced by numerical ids through a mapping dictionary, and all the original triples are finally converted to id triples before storing. The conventional encoding approach is that all the terms retrieve their ids through sequential access of a single dictionary an approach which is easy to implement but not suitable for compressing large data sets due to time considerations and memory requirements. Consequently, encoding triples in parallel based on a distributed architecture with multiple dictionaries, becomes an attractive choice for this problem. However, under this model there exist three new challenges:

1. Consistency - a term appearing on different compute nodes should have the same id.

2. Performance - ensuring consistency based on naive methods (e.g. serializing requests) can lead to serious performance degradation.

3. Load balancing - the heavy skew of terms [13], which characterizes real-world Linked Data may lead to computation or synchronization hotspots for the nodes responsible for encoding these popular terms.

Both in space and time, the mapping of a term should keep its uniqueness. For example, once the term *dbpedia:IBM* is encoded as id *101* on node A, when encoding this string on another node B, we should also use the same value *101*. Hash functions are potentially useful, but the length of the hash required to avoid collisions when processing billions to terms makes this approach space-inefficient.

We can ensure the consistency of the encoding in the above example by copying the mapping *[dbpedia:IBM, 101]* across nodes, but network communication cost and dealing with concurrency (e.g. locking on data structures) would lead to poor performance.

Compared with the two issues above, load balancing presents a bigger challenge as the distribution of terms in the Semantic Web is highly skewed: there exist both popular (like terms in the RDF and RDFS vocabularies) and unpopular terms (like identifiers for entities only appearing for a limited number of times). For a distributed system, any encoding algorithm needs to be carefully engineered so that efficient network communication and computational load-balance are achieved.

Seventeenth International Workshop on the Web and Databases (WebDB 2014), June 22, 2014 - Snowbird, UT, USA.

In this paper, we propose a scalable solution for encoding massive RDF data in parallel. We develop an algorithm and implement it using the parallel language - X10 [6]. We evaluate performance with up to 384 cores and with datasets comprising of up to 11 billion triples (1.9 TB). Compared to the state-of-the-art [16], our approach is faster (by a factor of 2.6 to 7.4), can deal with incremental updates in an efficient manner (outperforming the state-of-the-art by several orders of magnitude) and supports both disk and in-memory processing.

The rest of this paper is organized as follows: Section 2 provides a review of related work. Section 3 introduces the proposed RDF dictionary encoding algorithm and improvements for the algorithm. Section 4 provides a quantitative evaluation of the algorithm. Section 5 concludes the paper and points to directions for future work.

2. RELATED WORK

Compression has been extensively studied in various database systems, and has been considered as an effective way to reduce the data footprint and improve the overall query processing performance [7] [1]. In terms of efficient storage and retrieval of RDF data, the approaches described in [8] are geared toward efficient storage and transfer, as opposed to having direct access to the data for efficient processing. The compression method adopted by the most popular triple stores, such as RDF-3X [14], is dictionary encoding that performs the string-id conversion on the basis of a single dictionary table. This method does not avail of potential speed-up by parallel implementations. Various distributed solutions used to manage RDF data have been proposed in the literature [11] [12]. Nevertheless, their main focus is on data distribution after all the statements have been encoded. There exists only two efficient methods focused on parallel dictionary encoding of RDF data. One is based on parallel hashing [9] and the other uses the MapReduce model [16].

Goodman et al. [9] adapt the linear probing method on their Cray XMT machine, and realize the parallel encoding on a single dictionary through parallel hashing, exploiting specialized primitives of the Cray XMT. Their evaluation has shown that their method is highly efficient and the run-time is linear with the number of used cores. This method requires that all data is kept in memory and is deeply reliant on the shared memory architecture of the Cray XMT, making it unsuitable for commodity distributed memory systems. They report an improvement of 2.4 to 3.3 compared to the MapReduce system on an in-memory configuration. By comparison, on similar datasets, our approach outperforms the MapReduce system a factor of 2.6 to 7.4, both on-disk and in-memory.

Compared with [9], the MapReduce method proposed by Urbani et al. [16] is more general in that it can be run on ordinary clusters and on-disk. There are three main elements to their system: (1) the popular terms are cached in memory by sampling the data set, so that these popular terms assigned to each task could be encoded locally and consequently prevent eventual load balancing problems, (2) a hash function is used to assign grouped terms to reduce tasks, which then assign the term identifier, keeping the consistency of the encoding, and (3) the MapReduce framework facilitates the parallel execution of the program. Their evaluation on Hadoop has shown that their system is efficient and scales well. Although our algorithm is more straightforward, as we will show in Section 4, our approach is both faster and more flexible, exploiting the finer-grain control of modern parallel language.

3. DICTIONARY ENCODING FOR RDF

In this section, we first describe the details of our RDF dictionary encoding algorithm. Then, we present a set of extensions to our basic algorithm which extend the applicability of the approach to a larger set of problems and computation platforms.

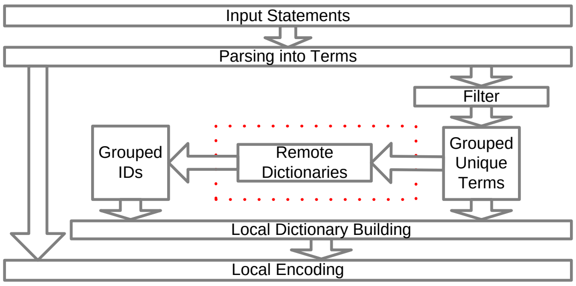


Figure 1: Workflow of RDF encoding in our implementation.

3.1 Main Algorithm

Consider the following RDF statements, using a simplified notation for terms in the interest of conciseness:

<A1 p1 B1>, <A1 p1 B2>, <B1 p2 C2>, <C2 p3 D2>,
<A1 p1 B3>, <B1 p2 C1>, <B2 p2 C3>, <C1 p3 D1>

We utilise a distributed dictionary encoding method for the input data, transforming RDF terms into 64-bit integers and representing statements using this encoding. The data is first divided into a number of equal-size *chunks* and then assigned as input for processing on separate computation nodes. For an example two-node system, the first four statements are assigned to the first node and others are for the second node. Then, the overall implementation strategy for each node and the corresponding data flow are shown in Figure 1, which can be divided into three separate phases as follows.

Step 1. Every statement in the input set is parsed and split into individual *terms*, namely, *subject*, *predicate*, and *object*[1]. Then the duplicates are locally eliminated by a filter, and the extracted set of *unique* terms is divided into individual groups according to their hash values. The number of groups is set to the same as the number of nodes, and terms with the same hash are placed in the same group. We assume that terms with an odd number hash to the first node and constants with an even number hash to the second node (e.g. B1 hashes to node 1, B2 hashes to node 2). Then, the process on the first node will be as below. The terms in the first group (namely {A1,p1,B1,p3}) will be sent the first node itself and others are send to the second node, for the following dictionary encoding.

parsing	[A1,p1,B1,A1,p1,B2,B1,p2,C2,C2,p3,D2] ⇒
filter	(A1,p1,B1,B2,p2,C2,p3,D2) ⇒
hash-groups	{A1,p1,B1,p3} + {B2,p2,C2,D2}

Step 2. Once the grouped unique terms have been transferred to the appropriate remote node, the term encoding can commence. The term encoding implementation at each place is similar to sequential encoding. Each received term access the local dictionary sequentially to get their numerical ids. In this process, if the mapping of a term already exists, its id is retrieved, else, a new id is created, and the new mapping is added into the local dictionary. In both cases, the id of the encoded term is added into a temporary array for so that it can be sent back to the requester(s). The value of a new id is determined by the summation of the largest id in the dictionary and the value *n*, the number of nodes. This guarantees

[1] Although our system can parse and process NQuads, in the interest of simplicity, we will only use triples for all of our explanations.

there is no clash between term ids assigned at different nodes. Furthermore, each id is formatted as an `unsigned` 64-bit integer in order to remove limitations regarding maximum dictionary size[2]. In this case, the first node could receive the ids as following.

send {A1,p1,B1,p3} + {B2,p2,C2,D2}
receive {1,3,5,7} + {2,4,6,8}

Step 3. The statements at each node can be encoded after all the ids of the pushed terms have been pushed back. Since the terms and their respective ids are held in order inside arrays, we can easily insert these mappings into the local dictionary. Once inserted, we encode the parsed triples kept in the first step. Each of the three steps is implemented in parallel at each node, and the whole encoding process terminates when all individual nodes terminate. Namely, we will get the encoded triples shown as below at the first node.

parsed [A1,p1,B1,A1,p1,B2,B1,p2,C2,C2,p3,D2] ⇒
encoded <1 3 5>, <1 3 2>, <5 4 6>, <6 7 8>

Compared to the MapReduce method [16], we do not quantify any skew, but just employ a *filter* structure (for example a simple `hashset`) to process the terms and to extract the unique terms that need to be transferred to remote node. This is done for all terms irrespective of their popularity. Using the filter guarantees that any given term can possibly move to a remote node just once per current node, which is shown very efficient on handle the data skew existing in the semantic web in our evaluations in Section 4.

Moreover, our method can be easily implemented by modern parallel language used in *high performance computing*, for example X10 and MPI etc. This makes our implementation much more straightforward that we only need to send unique terms to remote dictionaries and retrieve their ids, but not transfer any triples at all. In comparison, [16] has to decompose all the triples in the form of <*key,value*> pairs and redistribute all of them among all the nodes. Furthermore, all the terms have to be redistributed again after the encoding process so as to reconstruct all the triples. This could bring very heavy network communication and also computations, impacting the encoding performance.

3.2 Improvements

Flexible Memory Footprint. In our basic algorithm, all the objects (such as the *parsed terms*, *grouped terms*, *remote dictionaries* and *grouped ids* etc. in Figure 1) are kept in memory throughout the encoding process. This limits the applicability of the method to clusters with sufficient memory to hold all data.

To alleviate this problem, we divide the input data set into multiple *chunks*, usually a multiple of the number of computation nodes. The encoding process is then divided into multiple *loop* iterations corresponding to each chunk. In each of these encoding iterations, a node is assigned a specified number of chunks, while we only keep the *remote dictionaries* at each node for next iterations. This method makes our algorithm suitable for nodes with various memory sizes, provided the chunks are small enough. Note that the chunks can be made smaller by simply dividing the input data set into more chunks. It is expected that too many such chunks would lead to a decrease in performance, as there would be redundant filter and push operations for the same terms at the same node in different loops. We assess this trade-off in the evaluation Section 4.

Transactional-style Data Processing. A commonly occurring scenario is real-time processing of RDF data sets. In such cases, data is inserted as part of a *transaction*. Normally the chunks of data inserted are very small, containing only a few hundred statements, and there is no need to distribute data sets. Instead, one could just encode the dataset using a single cluster node, which can be done by assigning specified *thread workers* for the underlying system. Furthermore, parallel transactions with multiple data sets on multiple nodes are also supported using the same way. Finally, an optimized data-node assignment strategy can be integrated with our implementation if needed, but such a strategy is out of the scope of this paper. Similarly, in this paper, we do not address rolling back transactions or deletes. In general, although our system can be extended to support transactional loads, its main utility is in encoding large datasets.

Incremental Update. Another typical application is the incremental update of RDF data sets. It is often required that such systems must encode a new dataset as an increment to already encoded datasets. Typically, the new input data set is large. In this scenario, we extend our algorithms for incremental update through reading local dictionaries in memory before the encoding process.

4. EVALUATION

We implement our method using the X10 parallel language and conduct a rigorous quantitative evaluation of the proposed encoding heuristic in terms of: (a) program execution time, (b) scalability and (c) communication overhead. Moreover, we compare our implementation with the state-of-the-art MapReduce-based encoding technique [16].

4.1 Experimental setup

Platform. We use up to 32 IBM iDataPlex® nodes with two 6-core Intel Xeon® X5679 processors clocked at 2.93 GHz, 128GB of RAM and a single 1TB SATA hard-drive, connected using Gigabit Ethernet. We use Linux kernel version 2.6.32-220, X10 version 2.3 compiling to C++ and gcc version 4.4.6.

For the MapReduce programme [16], we use the latest version and run it on Hadoop v0.20.2. We set the following system parameters: *map.tasks.maximum* and *reduce.tasks.maximum* to 12, the *mapred.child.java.opts* to 2 GB and the rest of the parameters are left to the default values. The implementation parameters are configured with the recommended values: *samplingPercentage* is set to 10, *samplingThreshold* to 50000 and *reducetasks* to the number of cores. We have verified the suitability of these settings with the authors.

Datasets. For our evaluation, we have used a set of real-world and benchmark datasets (also shown in Table 1): DBpedia [3] is an extract of the structured information from Wikipedia pages represented in RDF triples. LUBM [10] is a widely used benchmark that can generate RDF data sets of arbitrary size. BTC [5] is a Web crawl encoding statements as N-Quads, while Uniprot [2] is a large collection of biological function of proteins derived from the research literature. We chose these data sets because they vary widely in terms of size and kind of data they represent. The popularity and diversity of these datasets contributes to an unbiased evaluation.

4.2 Runtime

Compression. Initially, we examine the runtime of our implementation. We perform these tests using 16 nodes (192 hardware cores) and report the compression results achieved by our algorithm in Table 1: Column **# Stats** gives the number of statements (triples) in

[2] It is possible to use arbitrary- or variable-length ids in order to further optimize space utilization, but this is beyond the scope of this paper.

Table 1: Dataset information and compression achieved

Dataset	# Stats.	Input (GB)		Output (GB)		Compr.
		Plain	Gzip	Data	Dict.	Ratio
DBpedia	153M	25.1	3.5	3.5	2.7	4.1
LUBM	1.1B	190	5.5	24.8	17.7	4.5
BTC2011	2.2B	450	20.9	65.6	40	4.3
Uniprot	6.1B	797	58.7	136	46.4	4.4

Table 2: Disk-based runtime and rates (192 cores)

Dataset	Runtime (sec.)		Rates (MB/s)		Imprv.
	MapR.	X10	MapR.	X10	
DBpedia	430	59	59.7	435	7.3
LUBM	1739	453	111.9	429.5	3.8
BTC2011	2817	956	163.6	482	2.9
Uniprot	6160	1515	132.5	538.7	4.0

Table 3: In-memory runtime and rates (192 cores)

Dataset	Runtime (sec.)		Rates (MB/s)		Imprv.
	MapR.	X10	MapR.	X10	
DBpedia	368	50	69.8	514	7.4
LUBM	1382	254	140.8	766	5.4
BTC2011	1809	708	254.7	650.8	2.6
Uniprot	5076	937	160.8	871	5.4

each benchmark. The size of the input data sets is given both in the terms of plain and gzip format in columns 3 and 4. The output column is composed of the compressed statements and the corresponding dictionary tables at all places. Finally, the resulting compression ratio is calculated by dividing the size of the input files (in plain format) by the size of the total output. The compression ratios for the four data sets are similar: in the range of 4.1 - 4.5. Note that although these ratios are smaller than the compression ratio achieved by `gzip`, our output data can be processed **directly** and we can also compress these outputs further using `gzip`, if need be. We achieve smaller compression ratios compared to MapReduce [16], because we use 64-bit integers to encode all terms, while their approach uses smaller integers for encoding parts of terms as well as further `gzip` compression on their output data[3].

Runtime and Throughput. We compare the runtime and throughput between our approach and that of the MapReduce method in two cases: disk-based encoding and in-memory encoding. In the first case, the reading and writing data is on disk (or HDFS based on disk). For the latter, we process all data in memory. For memory based I/O, we pre-read the statements in an `ArrayList` at each place and also assign the output to `ArrayList`. As MapReduce does not provide such mechanisms, we instead set the path of the Hadoop parameter *hadoop.tmp.dir* to a `tmpfs` file system resident in memory. The results of these two cases are shown in Table 2 and Table 3. We define runtime as the time taken for the whole encoding process: reading files, performing encoding and writing out the compressed triples and dictionaries. The throughput is given in terms of input statements processed per second (*Rates*). These rates are calculated by dividing the input size (in plain format) by the algorithm runtime.

From Table 2, our approach is $2.9 - 7.3$ times faster than the MapReduce-based approach for disk-based computation, and $2.6 - 7.4\times$ for in-memory computation as illustrated in Table 3. The smallest speedup occurs for the BTC2011 benchmark, however it should be noted that in this instance, whereas we encode `NQuads`, MapReduce discards the fourth term in the input data and just processes the first three terms. We also noticed that the encoding throughput of Uniprot in both cases is much higher than the other three data sets. We attribute this to the large number of recurring popular terms. Comparing the two cases, we can see that the in-memory encoding is faster than the disk-based one for both algorithms, although not dramatically so. Moreover, the improvements we achieved in Table 3 are greater than those in Table 2 for the LUBM and Uniprot data sets, marginally greater for DBpedia and slightly smaller for the BTC2011 data set. This illustrates that the two algorithms gain disproportionally from the faster I/O over different data sets (with our system showing better gains overall).

Transactional. We simulated two transactional processing scenarios with in-memory encoding: (1) sequential transactions on a single node and (2) multiple parallel transactions on multiple nodes using the LUBM data set. To simulate transactions, we first encode the 1.1 billion triples in the *LUBM8000* benchmark. Next, we prepare a RDF data set that contains 1M triples, split into 10K, 1K, 100, and 10 chunks, respectively. After encoding is complete, we encode these new input chunks (every 10 chunks) sequentially and record the corresponding encoding time. For the multiple parallel transaction scenario, we could only record the encoding time for our implementation since Hadoop uses a centralized model for data storage.

Table 4: Runtime for processing 1M statements in the transactional scenario (192 cores, in memory)

# Stats	*Avg. runtime per 10 chunks (sec.)*		
per chunk	MapR.	X10	X10_Para.
100	439	0.211	0.164
1K	441	0.359	0.391
10K	454	1.761	0.648
100K	454	17.177	2.192

Results are presented in Table 4. One can clearly observe that our approach is orders of magnitude faster than the MapReduce approach for the sequential case. The latter is neither optimized nor suitable for this use-case, since the startup overhead dominates the runtime, as evident from the observation that the average time to process chunks with different sizes is approximately the same. For our system, we observe that the average runtime of our approach increases with increasing chunk sizes, and the trend moves toward linear for the sequential case.

Since we are using 192 cores and the number of chunks used in this scenario is 10, for each transaction with the parallel processing by our prototype, the chunks can be processed at once by 10 threads in parallel. The results in Table 4 show that the runtime is around 0.2 seconds when the number of statements is about 100 in each chunk, which is slightly worse than our expectations for real-time applications, although still well within an acceptable range.

Updates. Finally, we evaluate the incremental updates scenario for RDF encoding again using the *LUBM8000* data set and by splitting it into 2, 4, and 8 chunks, respectively. The resulting data sets are compressed in 2, 4 and 8 different executions respectively. Before each encoding cycle, we empty the cache so as to simulate likely real world conditions. The results comparing our approach and MapReduce are shown in Table 5. As expected, the performance for both algorithms decreases with increasing number of chunks, because of the additional processing required during the encoding

[3] Recall again that we focus on the performance issues of dictionary encoding in this paper, but not the compression ration.

Table 5: Runtime for incremental update scenario with different chunk size (192 cores, on disk)

# Chunks	Chunk Size	Runtime (sec.) MapR.	X10	Imprv.
1	190 GB	1739	453	3.8
2	95 GB	2468	551	4.5
4	47 GB	3900	755	5.2
8	23 GB	6704	1164	5.8

phase (e.g. reading the dictionary into memory). However, the increase in program runtime for our approach is much smaller than MapReduce. A possible explanation is that because our dictionary reading operation is faster, the startup overhead of our system is lower. It is also possible that the efficacy of the popularity caching technique used by MapReduce decreases disproportionately as the number of chunks increases.

4.3 Scalability

We test the scalability of our algorithm by varying the number of processing cores and the size of the input data set. We use the LUBM benchmark in our tests as it facilitates the generation of datasets of arbitrary size.

Number of Cores. We initially fix the input data set to 1.1 billion triples and double the number of cores from 12 (single node) till 384. The test results for our algorithm and those of the MapReduce-based approach are shown in Figure 2(a). These results demonstrate that the run time for both algorithms *decreases* with an increase in the number of cores. The speedup obtained with an increasing number of cores compared to a baseline of 12-cores for both algorithms is presented in Figure 2(b). In our system, with a small number of cores, the runtime is not linear, since for a single node there is no network communication. Nevertheless, starting from 24 cores, the speedup becomes almost linear. In contrast, the speedup of the MapReduce-based approach is almost linear (even super-linear) initially before plateauing for values of 92 cores and greater. This result mirrors the results obtained in [16]. There can be several reasons for the latter slowdown: we hypothesize that this slowdown in MapReduce may be due to load imbalance, increased I/O traffic and platform overhead.

Size of Datasets. We create a large LUBM data set with 11 billion triples, which is roughly equivalent to the *LUBM80000* benchmark. We split this data set into a number of chunks, each of which contains 140K triples, allowing us to study the effect of *loop* described in Section 3.2.

We start our tests with 690 million triples and repeatedly double the size of the input until we reach a dataset comprising 11 billion triples. Additionally, for each dataset, we also vary the number of *chunks read per loop* for our implementation. The results are presented in Figure 2(c). We see that the runtime for both algorithms is nearly linear with the size of the input data sets. We also notice that MapReduce achieves a slightly super-linear speedup until 5.5 billion triples. After that, MapReduce speedup becomes linear with the input size. For our algorithm, we have experimented with 1, 5, and 10 chunks in each loop. One can see that the scalability of our algorithm is not linear with input data when reading 1 chunk per loop. But, speedup becomes better as we increase the number of chunks read per loop, and it matches the ideal linear speedup scenario when reading 10 chunks per loop. The reason may be that small chunks result in redundant *filter* and *push* operations for the same terms at the same node in different loops. Such an interpretation is in sympathy with our expectations described in Section 3.2.

Table 6: Detailed term information during encoding 1.1 billion triples

# Core	# Outgoing (M) Max	Avg.	# Misses (M) Max	Avg.	Miss Ratio Max	Avg.
24	11.65	11.59	10.95	10.95	95.7%	94.5%
48	5.85	5.78	5.46	5.46	96.1%	94.5%
96	2.94	2.89	2.73	2.73	96.1%	94.5%
192	1.48	1.43	1.35	1.35	96.4%	94.5%
384	0.74	0.70	0.90	0.87	96.4%	94.5%

Furthermore, Figure 2(c) investigates the trade-off between reduced memory consumption and performance as well. For the optimal scalability case with reading 10 chunks at a time, we need to process $10 \times 140K = 1.4M$ triples in each loop. Since, in Table 1, we show that 1.1 billion triples is about 190 GB, the size of 1.4 million triples would be about 250 MB, which is well within the RAM availability of most machines. Not withstanding this optimal case implementations using 5 chunks at a time (125 MB) and 1 chunk at a time (25 MB) is only accompanied with little and moderate scalability loss respectively.

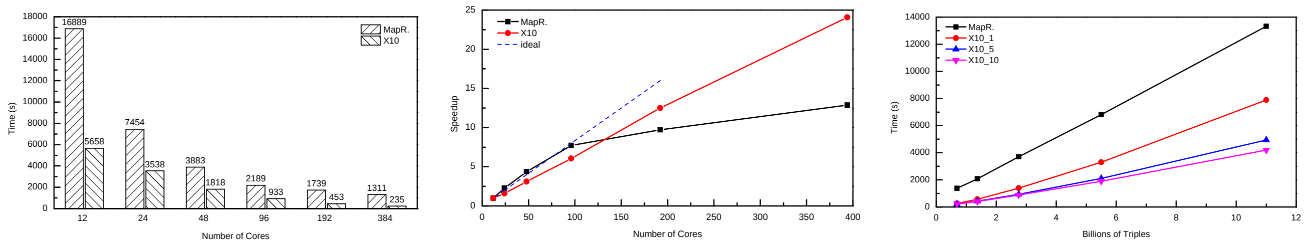
4.4 Load Balancing

We measure the load-balance characteristics of our algorithm in terms of five metrics defined later in this section. We instrument our code with counters to gather data for the first four metrics. The data for the final metric is obtained using the tracing option provided by the X10 implementation.

- *number of outgoing terms*: The number of terms transferred to a remote place. This metric gives insight into the *communication* load balance achieved by our algorithm. For example, the larger the number of outgoing terms, the greater the associated network traffic.
- *number of misses*: The number of terms that are not already encoded (*missed*) in the dictionary and hence require the generation of a new id.
- *miss ratio*: The number of misses divided by the sum of hit and miss for the local dictionary.
- *number of processed terms*: the number of terms processed by a computing node.
- *received bytes*: the size of processed terms in bytes at a computing node.

We encoded 1.1 billion LUBM triples on a varying number of cores to gather data for the first three metrics described above. The results are presented in Table 6. We can see that the average values of the three metrics for all the tests are very close to the maximum values, suggesting excellent load balancing performance. The scalability of our algorithm with an increasing number of processing cores is highlighted well in these results. There is a clear linear decrease in all three metrics with an increase in the number of processing cores. Finally, the results also illustrate a consistent almost uniform miss probability for each dictionary. The average miss ratio is about 94.5%, indicating that we have redundant computation on average for 5 out of every 100 terms. This ratio approached the ideal value of 100%, which is nevertheless difficult to achieve in a distributed systems without significant coordination overhead.

The last two metrics capture the load at each compute node in terms of the number of terms processed and size of data received



(a) Runtime by varying cores

(b) Speedups by varying cores

(c) Runtime by varying size

Figure 2: Scalability of two algorithms: (a) encoding 1.1 billion triples with varying the number of computation cores from 12 to 384, (b) the corresponding speedups achieved by varying the cores, and (c) the number of triples starts with 690 million and repeatedly double to 11 billion (192 cores, on disk)

Table 7: Comparison of received data for each computing node when processing 1.1 billion triples using 192 cores (millions)

Algorithm		Recv. Bytes		Recv. Records	
		Max.	**Avg.**	**Max.**	**Avg.**
MapR.	**Job1**	9.94	4.02	24.04	1.73
	Job2	135.61	79.77	30.91	17.28
	Job3	120.81	106.82	19.61	17.28
X10		194.71	187.82	1.48	1.43

in bytes. These metrics are important for measuring computational load balance and are used here to provide comparison with the performance available using the MapReduce approach. Since MapReduce divides the whole dictionary encoding into three separate jobs and the implementation does not provide the relative metrics, we extract the *reduce input records* and *reduce shuffle bytes* in the reduce phase of each job from the Hadoop logs. These two items indicate the number of records processed and the corresponding data sizes for each of the 192 reduce tasks.

The results are summarized in Table 7 and demonstrate that the difference between the maximum and the average value of these metrics for our method is much smaller than MapReduce, indicating better load balancing. In addition, even when comparing only with the reduce phase of MapReduce, our system results in a lighter workload and less network communication, especially taking into consideration that we are using a longer representation (64 bits).

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a dictionary encoding algorithm for the encoding of big RDF data. We have presented an extensive quantitative evaluation of the proposed algorithm and conducted a comparison with a state-of-art system using the MapReduce model. Our main conclusions are that the proposed algorithm is: (a) highly scalable both with increments in number of cores and in the size of the dataset; (b) computationally fast, encoding 11 billion statements in about 1.2 hours, and achieving a $2.6 - 7.4\times$ improvement over the MapReduce method, both on disk and in memory; (c) flexible for various semantic application scenarios and (d) robust against data skew, showing excellent load balancing.

Current work lies in combining this approach with rapid indexing methods to load large RDF datasets in very little time. Our long term goal is to develop a highly scalable distributed analysis framework for extreme-scale RDF data.

Acknowledgment. This work is supported by the Irish Research Council and IBM Research Ireland.

6. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] R. Apweiler, A. Bairoch, C. H. Wu et al. Uniprot: the universal protein knowledgebase. *Nucleic Acids Research*, 32:115–119, 2004.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: a nucleus for a web of open data. In *ISWC*, pages 722–735, 2007.

[4] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[5] Billion Triple Challenge, http://challenge.semanticweb.org.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[7] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282, 2001.

[8] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto. RDF compression: basic approaches. In *WWW*, 2010.

[9] E. L. Goodman, E. Jimenez, D. Mizell, S. al Saffar, B. Adolf, and D. Haglin. High-performance computing applied to semantic databases. In *ESWC*, pages 31–45, 2011.

[10] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3:158 – 182, 2005.

[11] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: a federated repository for querying graph structured data from the web. In *ISWC*, pages 211–224, 2007.

[12] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4:1123–1134, 2011.

[13] S. Kotoulas, E. Oren, and F. van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. In *WWW*, pages 531–540, 2010.

[14] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB.*, 1:647–659, 2008.

[15] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. Linkedgeodata: A core for a web of spatial open data. *Semantic Web Journal*, 2011.

[16] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal. Scalable RDF data compression with mapreduce. *Concurrency and Computation: Practice and Experience*, 25:24–39, 2013.