Massive Semantic Web data compression with MapReduce

Jacopo Urbani, Jason Maassen, Henri Bal
Department of Computer Science
Faculty of Science
Vrije Universiteit, Amsterdam
{j.urbani,j.maassen,he.bal}@few.vu.nl

ABSTRACT

The Semantic Web consists of many billions of statements made of terms that are either URIs or literals. Since these terms usually consist of long sequences of characters, an effective compression technique must be used to reduce the data size and increase the application performance. One of the best known techniques for data compression is dictionary encoding. In this paper we propose a MapReduce algorithm that efficiently compresses and decompresses a large amount of Semantic Web data. We have implemented a prototype using the Hadoop framework and we report an evaluation of the performance. The evaluation shows that our approach is able to efficiently compress a large amount of data and that it scales linearly regarding the input size and number of nodes.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data compaction and compression

1. INTRODUCTION

The Semantic Web [2] is an extension of the current World Wide Web, where the semantics of the information can be interpreted by machines. The information is represented as a set of statements, and each statement is made of three different terms: a *subject*, a *predicate*, and an *object*. An example statement is

```
<http://www.vu.nl> <rdf:type> <dbpedia:University>
```

This example states that the concept identified by the URI *http://www.vu.nl* is of type *dbpedia:University*. The Semantic Web is made of billions of such statements, and they describe information on a very wide range of domains, from biomedical information[1] to government information[2]. URIs are often used to identify concepts to ensure disambiguity on the Web.

Since the terms consist of long strings (e.g. URIs), most Semantic Web applications, like RDF storage engines, compress the statements to a more compact representation so that they can save space and increase the performance. One of the most often used techniques to compress data is *dictionary encoding*. In the Semantic Web, we empirically estimated that on average a statement takes about 150-210 bytes. If we replace the text with 8 bytes numbers, the same statement will take only 24 bytes, giving us a compression ratio of about 1:6 to 1:8.

Currently the amount of Semantic Web data is steadily growing and compressing many billions of statements becomes more and more time-consuming. Also, a centralized approach may be economically unfeasible because of the need of a large memory to store the dictionary. However, in order to meet the requirements of current high performance applications a fast and scalable compression is crucial.

To the best of our knowledge there is no distributed approach to this problem. In this paper we propose a technique to compress and decompress Semantic Web statements using the MapReduce programming model [4]. This compression technique was fundamental in our recent work on Semantic Web inference engines because it allowed us to reason directly on the compressed statements with a consequent increase of performance. As a result, we were able to reason over tens of billions statements and this is currently the state of the art in the field [12, 11].

The compression technique we present in this paper has (i) performance that scales linearly; (ii) the ability to build a very large dictionary of hundreds of millions of entries and (iii) the ability to handle load balancing issues with sampling and caching.

This paper is structured as follows. In Section 2 we discuss the conventional approach and we highlight the problems that arise. Sections 3 and 4 describe how we have implemented the data compression and decompression in MapReduce. Section 5 evaluates our approach. Section 6 contains a discussion about possible extensions and Section 7 summarizes the related work. Finally, we draw some conclusions in Section 8.

[1] `http://www.linkedlifedata.com`
[2] `http://data.gov.uk`

Algorithm 1 Sequential algorithm of dictionary encoding

```
compress_data(Iterator statements) {
  dictionary_table.initialize()
  for (statement in statements) {
    for (term in statement) {
      if (not dictionary_table.contains(term) {
        newID = dictionary_table.get_new_ID()
        dictionary_table.put(term, newID)
        term = newID
      } else {
        //Replace the string term with a numerical ID
        term= dictionary_table.getID(term)
      }
    }
  }
}

decompress_data(Iterator statements) {
  for (statement in statements) {
    for (term in statement) {
      textualTerm = dictionary_table.getText(term)
      term = textualTerm
    }
  }
}
```

2. CONVENTIONAL APPROACH

The idea behind dictionary encoding is very simple. The function *compress_data* of Algorithm 1 shows a sequential algorithm to compress Semantic Web statements. The compression algorithm starts by initializing the dictionary table. The table has two indexable columns, one that contains the terms in their textual representation and one that contains the corresponding numerical ID. The algorithm reads all the statements and it checks whether the terms exist in the table. If a term exists, then the algorithm retrieves the numerical ID and replaces the term with the number. Otherwise, the algorithm requests a new numerical ID, inserts a new pair in the table, and proceeds with the replacement. The algorithm outputs the compressed statements and the dictionary table.

The function *decompress_data* of Algorithm 1 describes how we decompress the data. For each term in the collection, the algorithm looks up corresponding textual version in the dictionary table and replaces the numerical ID.

The computation of these two algorithms can be "naively" distributed by partitioning the input and processing the partitions on several machines. In this scenario one machine will store the dictionary table and the other machines will query it to retrieve the numerical IDs. The problem with this approach is that all machines will constantly need to query the dictionary, generating massive network communication to the table's location. If the infrastructure is not fast enough this communication will quickly become a performance bottleneck.

There is another problem to consider when we want to compress Semantic Web statements. For most other applications the dictionary table is small enough to fit in the machine's main memory. For example, if we want to compress English text we can store the dictionary table in the main memory because there are normally not more than a few hundred thousands distinct words. However, in a collection of billions of web statements there are normally hundreds of millions of unique terms. If the URI's average length is 80 bytes, then 300 million entries take about 24 GB of space. Unless we use a machine with a very large amount of memory, we are obliged to store the table on disk and a lookup on a data structure stored on disk is significantly slower that a lookup in the main memory.

The combined effect of these two problems makes a distributed approach unattractive in terms of efficiency. Consider that we want to compress one billion statements, and we want to distribute the computation over 32 machines. We store the dictionary table on one machine using a common DBMS. We make an optimistic estimation that querying the database from each of the 32 machines will take at least one millisecond. In order to compress one billion statements, we must query the database three billion times. If the queries are executed in parallel, the execution time will be longer than one day.

The performance can be improved by caching the most popular terms to limit the number of queries to the DBMS. However, in our case only few terms will benefit from the cache because the majority of the terms occurs very rarely. Therefore, although caching will undoubtedly increase the performance, it will not solve the problems of querying the database.

We tackle these issues and we propose a completely different approach using MapReduce to compress and decompress the data. Our approach is able to compress one billion of statements in less than one hour, which is more than one order of magnitude less than a conventional approach. Sections 3 and 4 will describe it in more detail.

3. MAPREDUCE DATA COMPRESSION

MapReduce can be either used alone or in combination with an external DBMS. If we would use an external DBMS to store the dictionary table, we can minimize the number of queries exploiting the sorting ability of MapReduce. In this case, we could write an appropriate map function so that all the statements that share the same term in the same position would be grouped together. The reduce function could query the DBMS once per group and replace at most one term in each statement. Since each statement contains three terms, we would need to launch three MapReduce jobs to completely compress the input.

The advantage of this approach compared to the naive one is that here we need to query the DBMS only once per term, and not at every occurrence. However, the performance will suffer from load balancing problems because the statements are grouped on the single terms, and some of them will generate groups that are too large to be processed by a single machine. Another problem with this approach is that we need to execute one job for each part of the statements (first the subject, then the predicate, etc.).

We propose an alternative algorithm which does not use an external dictionary, but instead builds it internally. We prevent eventual load balancing problems by sampling the most common resources and caching them locally. We do not

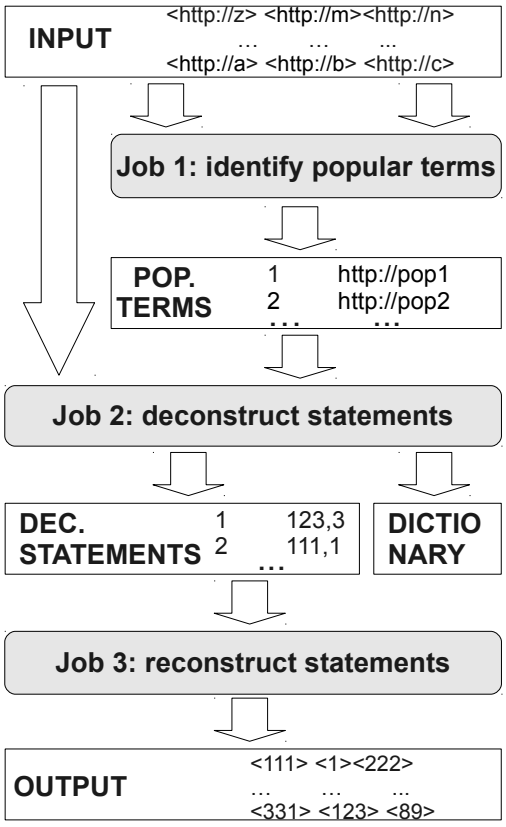

Figure 1: Overall compression algorithm

execute one job for each part of the statements, but instead we first deconstruct the statements, replace the terms with the numerical IDs, and finally reconstruct them.

Figure 1 illustrates the overall algorithm. It consists of a sequence of three MapReduce jobs. The first job identifies the popular terms and assigns them a numerical ID. This algorithm is explained in section 3.1. The second job (section 3.2) deconstructs the statements, builds the dictionary table and replaces all terms with a corresponding numerical ID. The last job (section 3.3) will read the numerical terms and reconstruct the statements in their compressed form.

3.1 Job 1: caching of popular terms

In the Semantic Web, the distribution of the terms is highly skewed, having few popular terms and many that occur only few times. The purpose of the first job is to identify the most popular terms so that we can treat them in a different way. In order to do so, the job counts the occurrences of the terms and select the subset of the most popular ones. Since the input is large, counting all the occurrences becomes an expensive operation because we must emit a number of pairs that is three times the number of statements. To increase the performance, we randomly sample the input and extract the popular terms (i.e. the terms which appear more often than a specified threshold).

Algorithm 2 Dictionary encoding: counting the terms occurrences

```
map(key, value) {
  /*key: void */
  /*value: consists of one statement */
  random = Random.newNumber(0:100)
  if (random < samplingPercentage) {
    emit(value.subject, 1)
    emit(value.predicate, 1)
    emit(value.object, 1)
  }
}

reduce(key, values) {
  /*key: one term in the collection*/
  /*values: they are a sequence of 1*/
  count = 0
  while (values.hasNext)
    count += value.next
  if (count > threshold)
    emit(key,count)
}
```

The algorithm is reported in pseudocode in Algorithm 2. It requires two parameters to run: *samplingPercentage* that sets the size of the sampling subset and *threshold* that sets the minimum occurance to consider a term as popular.

3.2 Job 2: deconstruct statements

The purpose of this job is to deconstruct the statements and compress the terms with a numerical ID. This methodology is designed to avoid having to launch one job for each part of the statement.

The algorithm is shown in Algorithm 3. Before the map phase starts, we load the popular terms that we have previously identified into the main memory. Since there are only few popular terms (see section 5.1), they can fit in the main memory without any problem.

The map function reads the statements and assigns each of them a numerical ID. Since the map tasks are executed in parallel, we partition the numerical range of the IDs so that each task is allowed to assign only a specific range of numbers. In our prototype we use a 64-bit number as identifier and we split it in two parts: the first four bytes will contain the id of the task that has processed the statement and the last four bytes will be used as incremental counter within the task. In this configuration, we can have at most $2^{32} - 1$ map tasks and each task can process an input of at most $2^{32} - 1$ statements. If we reserve more bytes for the internal map counter, then fewer map tasks can process a larger input, otherwise, if we reserve more bytes to store the map task number, then we can have more map tasks that process a smaller input.

For each term in the statements, the map function emits one intermediate pair. The key of the pair will be different whether the term is a popular one or not. If the term is a popular one, then the key will be the numerical ID retrieved from the in-memory cache that we have previously loaded. These pairs do not need to be further processed and can be output immediately. In case the term is not present in the

Algorithm 3 Dictionary encoding: deconstruct the statements and replace the terms

```
map(key, value) {
  /*key: irrelevant*/
  /*value: statement*/
  statement_id = counter++
  for (term in statement) {
    if (popular_terms.contain(term)) {
      id = popular_terms.getID(term)
      emit(id, statement_id+term_position)
    } else {
      emit(term, statement_id+term_position)
    }
  }
}

reduce(key, values) {
  /*key: term*/
  /*value: statements IDs + terms position*/
  if (key is numeric) {
    while (values.hasNext)
      emit(key,value.next)
  } else {
    counter++
    emit(counter,key) /*Dictionary table entry*/
    while (values.hasNext)
      emit(counter,value.next)
  }
}
```

cache, the map function will set the textual term as key. In both cases, the value of the pair will be the statement ID and the position of the term within the statement (1 if it is a subject, 2 if it is a predicate or 3 if it is an object).

We use a specific partitioner function to assign the intermediate pairs to the reduce tasks. This function behaves as follows: if the key of the pair is a number, then the pair will be randomly assigned to a reduce task. Otherwise, if the key is a string, the hash function will be used to determine the reduce task. Using this partitioning technique all the pairs with a term that was not already replaced (i.e. those are the not popular) will be grouped together as usual. The other pairs do not require any other process, therefore they are sent randomly to the nodes in order to avoid any load balancing issue.

Each group will have either a textual or a numerical key. If the key is a number, the pairs in the group refer to an already converted popular term and the function will simply output the pairs. In the other case, the function proceeds assigning a numerical ID to the term.

The numerical IDs are assigned in a similar way as for the statements. We use a long number which is split in two parts: the first will contain the reduce task number while the second will be used as internal counter. For every pair in the group, the reduce function will output a corresponding pair with the numerical ID as key and the unchanged input value. The function will also emit an additional pair with the numerical value as key and the text as value. This pair will be stored as part of the dictionary table.

This job will output: (i) a set of pairs which will have the numerical terms as keys and the information about the statements ID as values; (ii) a set of additional pairs, one for each non-popular term, with the numerical ID as key and the corresponding term's textual representation as value.

Algorithm 4 Dictionary encoding: reconstruct the statements

```
map(key, value) {
  /*key: numerical term*/
  /*value: statement ID + term position*/
  emit(value.statementID, key + value.term_position)
}

reduce(key, values) {
  /*key: statement ID*/
  /*value: numerical term + term position*/
  for (value in values) {
    case (value.term_position) {
      'subject' : statement.subject = value.term_id;
                  break;
      'predicate' : statement.predicate = value.term_id;
                  break;
      'object : statement.object = value.term_id;
                  break;
    }
  }
  emit(null, statement)
}
```

3.3 Job 3: reconstruct statements

The last job reads the previous job's output and reconstructs the statements using the numerical IDs.

The algorithm is shown in Algorithm 4. The keys of the input pairs contain the terms in the numerical format. The values contain the statement ID and the position of the term within the statement (subject, predicate or object). The map function does a partial swap between the key and the value. It emits a new pair which has as key the statement ID and as value the term plus its position.

The pairs will be grouped together for the reduce function. Since we now set the statement ID as key, the pairs will be grouped according to the statement they belong to. Each group will have exactly three pairs, one for each of the parts of the statement. For each group, the reduce function will read the three values and reconstruct the original statement with the numerical terms at the positions contained in the pairs values.

4. MAPREDUCE DATA DECOMPRESSION

In order to decompress the data, we must perform a join between the compressed statements and the dictionary table.

In MapReduce a join is normally executed by using the data which should be matched as key, and the remaining information as value. The reduce function checks whether the group values comply with the join requirements and it performs the join when this is true. However, if we do so, the performance will be affected by severe load balancing problems because certain terms are by far more popular than others and each group is processed by a single machine. We solve this problem by caching the dictionary table of the

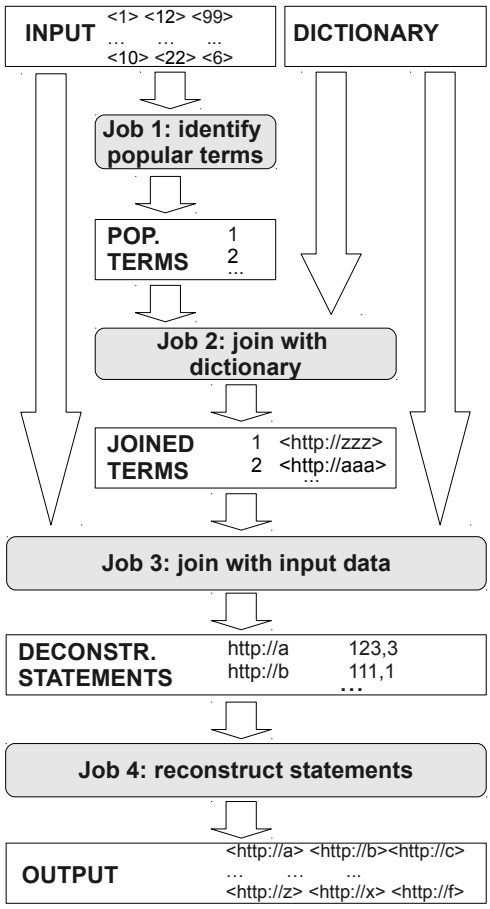

Figure 2: Overall decompression algorithm

most popular resources and perform their substitution.

There is another problem to consider: the MapReduce model assumes that the input consists of an homogeneous set of pairs, but here we have two different types of input (the statements and the dictionary table). We solve this problem by wrapping the statements and the dictionary table in a datastructure, such that they appear as a uniform collection of records.

Figure 2 illustrates the overall decompression algorithm. It consists of a sequence of four MapReduce jobs. The first job identifies the popular terms. The second job performs the join between the popular resources and the dictionary table. The third algorithm deconstructs the statements and decompresses the terms performing a join on the input. The last job reconstructs the statements in the original format.

The first and the last jobs are analogous to the ones explained in sections 3.1 and 3.3 and therefore they will not be further explained. Section 4.1 describes the second job and Section 4.2 describes the third one.

4.1 Job 2: join with dictionary table

The purpose of this job is to retrieve the corresponding textual equivalents of the popular terms. Since the dictionary table contains hundreds of millions of entries, we need to launch a MapReduce job to retrieve them.

Algorithm 5 Dictionary decoding: join with the popular terms

```
map(key, value) {
  /*key: term numerical ID*/
  /*value: textual version of the term*/
  if (popular_terms.contains(key)) {
    emit(key,value)
  }
}
```

The algorithm is shown in Algorithm 5. The map function loads the popular terms in memory and reads the entries of the dictionary table. If the table entry matches one of the popular terms, then the function outputs the table entry. For this task a reduce function is not required.

4.2 Job 3: join with compressed input

The second job deconstructs the statements and performs the join with the dictionary table.

The algorithm is shown in Algorithm 6. Before the map function starts, we load in memory a hash table with the popular table dictionary entries calculated in the previous job. This hash table will have as key the popular terms and as value the corresponding text.

The input of the map function can be either a statement or a dictionary table entry. If the input record is a statement, then the map function deconstructs it as described in Section 3.2. Then, the function checks whether the terms are popular or not. If they are, then the function will emit the pairs setting as key the text retrieved from the hash table. If not, then the numerical ID will be used as a key and the join will be performed on the single term during the reduce phase. If the input record is an entry of the dictionary table, then the function will emit a pair with the numerical ID as key and the text as value.

Similarly as section 3.2, we set a specific partitioner function to assign the pairs to the reduce tasks. The pairs with already converted terms will be randomly sent to the reducers and immediately returned. The other pairs will be grouped as usual and the join will be performed by the reduce function.

The reduce function stores the information on the statements in memory and saves the corresponding text of the numerical key in a variable. After this, the function will output new pairs with the textual term as keys and the values stored in memory as values. The next job will reconstruct the statements such that the terms are restored to their original format and no longer encoded by numbers.

5. EVALUATION

We implemented an open-source prototype[3] of the presented algorithms using the Hadoop framework (version 0.19.1).

[3] `https://launchpad.net/reasoning-hadoop`

Algorithm 6 Dictionary decoding: join against all the terms

```
map(key, value) {
  /*key: in case dictionary table entry this is the numerical
       ID, otherwise it is irrelevant*/
  /*value: either a statement or the textual representation of
       the term*/

  if (value is statement) {
    statement_id = counter++
    for (num_term in statement) {
      if (popular_terms.contains(num_term)) {
        textual_id = popular_terms.getID(num_term)
        emit(textual_id, statement_id+term_position)
      } else {
        emit(num_term, statement_id+term_position)
      }
    }
  } else {
    emit(key, value)
  }
}

reduce(key,values) {
  if (key is text) { /*already processed popular term*/
    for (value in values) emit(key,value)
  } else if (key is number) {
     textual_term = null
     for (value in values) {
       if (value is statement) {
         tmp_storage.add(value)
       } else { /*value is the term textual repr.*/
         textual_term = value
       }
     }
     for (value in tmp_storage) emit(textual_term,value)
  }
}
```

We used 32 nodes of the DAS3[4] cluster to set up our Hadoop framework (the Hadoop and HDFS masters were running on the cluster's main node). Each node is equipped with two dual-core 2.4 GHz AMD Opteron CPUs, 4 GB of main memory and 250 GB of storage. The nodes were connected using a Gigabit Ethernet. The Hadoop cluster was launched with the standard settings.

The input consists of a set of text files where every line contains one statement. The files were initially uploaded into the HDFS distributed filesystem and then processed by our MapReduce algorithms.

The performance was evaluated relative to the runtime and scalability. We present the results as follows. Section 5.1 reports the results of some tests launched on some common datasets. Section 5.2 reports the scalability results.

5.1 Runtime

We first launched our prototype on several real-world and artificial datasets. DBPedia[5] is a collection of statements extracted from Wikipedia pages. Swoogle[6] contains statements collected by Web crawlers. LUBM [5] is a benchmark tool that is widely used in the community. Uniprot[7] contains information on proteins and LDSR[8] is a selected collection of datasets in the Web.

[4] http://cs.vu.nl/das3
[5] http://dbpedia.org
[6] http://swoogle.umbc.edu

Dataset	Size (# stats)	Rate comp.	Runtime (sec.)	
			Com.	Dec.
DBPedia	16 GB (110M)	3.11	450	533
Swoogle	15 GB (78M)	3.49	364	425
LUBM	92 GB (553M)	3.06	1405	2491
Uniprot	213 GB (1857M)	4.37	4474	7670
LDSR	140 GB (931M)	3.07	2570	3886

Table 1: Execution time data compression and decompression on different datasets

The purpose of the tests was to evaluate the performance of our approach on different inputs. These datasets are different because some of them contain larger statements than others (for example, in DBPedia there are statements that contain all the text of one Wikipedia page as object). In addition, they have a different term distribution.

The results are presented in Table 1. The compression rate was calculated dividing the size of the input by the size of the compressed statements and the dictionary table, and varies from 1:3 (LUBM and LDSR) to 1:4 (Uniprot). Our technique is more efficient on larger inputs, where the computation is not dominated by the platform overhead. This is confirmed by our experiments, where we observe that the throughput of the compression algorithm is higher for a larger datasets (400K stats/s for Uniprot) than for a smaller one (214K stats/s for Swoogle). The same holds for the decompression algorithm, but on a smaller range (183K stats/s for Swoogle against the 242K stats/s for Uniprot).

We also note that the decompression algorithm is considerably slower than the compression algorithm by 14% (Swoogle) to 44% (LUBM). The difference is neither proportional to the input size nor on the number of unique terms. We speculate that the degradation of the performance of the decompression algorithm is due to a combination of these factors, but more tests are necessary for a precise analysis.

We evaluated the beneficial effects of the popular-terms cache by launching the compression algorithm on a fixed input (LDSR) and changing the size of the cache. We first disabled the cache, then, we increased the cache size by decreasing the threshold used to identify the popular terms.

The results are shown in Table 2. The first column shows the threshold we used to build the cache. The second column gives the size of the cache for the popular terms. We immediately see that using the cache reduces the runtime, regardless of the threshold. Using a lower threshold increases both cache size and performance. Reducing the threshold below 10M no longer produces any notable performance difference. These results show that only a very small numbers of terms are responsible for the degradation of the performance (12 out of 259M) while the majority does not introduce any problem.

[7] http://www.uniprot.org
[8] http://www.ontotext.com/ldsr/

Threshold	# Cache	Runtime (sec.)	Speedup
Disabled	0	3526	1
100M	1	2387	1.48
50M	1	2387	1.48
10M	12	1991	1.77
5M	36	2039	1.73
1M	111	2019	1.75
0.5M	195	2053	1.72

Table 2: Cache speedup for the compression algorithm.

5.2 Scalability

We tested the scalability of our algorithm by launching the execution with different input sizes and varying the number of nodes. Since real-world datasets do not differ only in the size but also on other aspects (i.e. number of unique terms and size of the statements), we used the LUBM benchmark tool to generate artificial datasets of different sizes. LUBM can generate datasets with a proportional number of unique terms and with a fixed statement size, so we could test the performance without being influenced by the nature of the input.

We started by generating a dataset of 17 million statements. Then, we repeatedly doubled the size until 1.1 billion statements were generated. On each dataset we launched the compression algorithm, immediately followed by the decompression algorithm. The runtime is reported in Figure 3.

If we compare the runtime of the compression algorithm with the one of the decompression algorithm, we note that the latter becomes slightly slower as the input increases. This difference shows that the compression algorithm has a better scalability than the decompression algorithm regarding the input size.

We also tested the scalability of our approach launching the execution on a fixed input on clusters with a different number of nodes. We have used a LUBM dataset of 500 million statements as input and we kept the number of mappers (256) and reducers (128) constant. We changed the number of nodes, starting at 1 and doubling up to 64. In Figure 4 we report the results. The y axis reports the runtime (in logarithm scale) while the x axis reports the number of nodes.

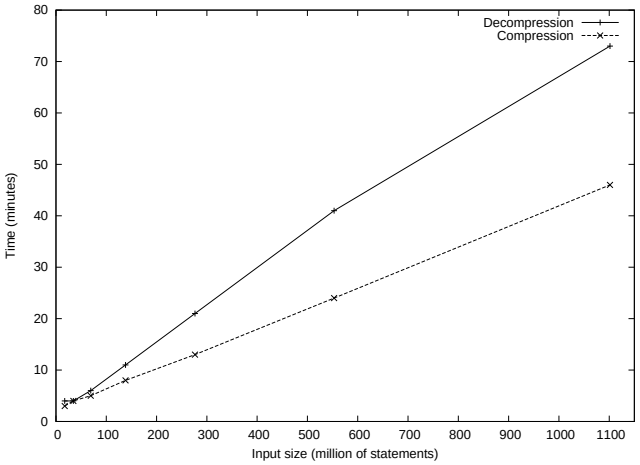

Figure 3: Scalability - input size

The runtime decreases as we increase the number of nodes. Initially, the speedup for both algorithms is superlinear (with 2 nodes the speedup is 3.11 for the compression algorithm and 3.41 for the decompression one) but later the performance gain decreases until it shows linear scalability (between 32 and 64 nodes the speedup is 1.85 for the compression algorithm and 2 for the decompression one).

Since we do not know the Hadoop implementation in detail, we speculate that the initial superlinear speedup is due to the platform overhead which is more relevant when we split the computation between fewer nodes. Another reason for this behavior could be the disk I/O which could act as a performance bottleneck if not sufficiently high. To support this hypothesis we note that the decompression algorithm is 3.1 times slower than the compression one when we use only one node while if we use 64 then the difference drops to 1.71. Since the decompression algorithm generates significantly more I/O communication than the compression algorithm, the reason could be the disk I/O but more research is necessary to confirm this.

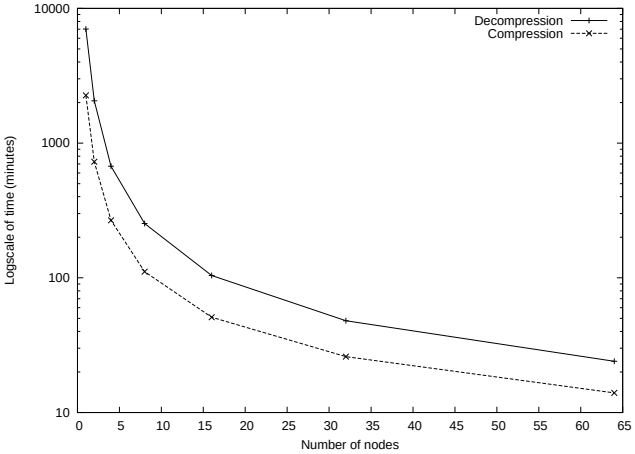

Figure 4: Scalability - number of nodes

6. FUTURE WORK

The work we have presented can be extended in many different ways. For example, the current algorithm can compress the data only once because the dictionary is built from scratch. Future work could aim to expand the algorithm to deal with incremental updates without recompressing the entire input every time. Some additional efforts could also focus at extending the MapReduce model to perform the decompression joins more efficiently.

Future work is also needed to extend these algorithms to work in different domains. Our approach is targeted to domains where there are large dictionaries but it could be interesting, for example, to see whether it could be adapted to perform generic text compression encoding the words with numbers. In this case, we could assign progressive IDs to the sentences and deconstruct them as we did with the statements.

7. RELATED WORK

Single machine dictionary encoding is used in RDF storage engines like Hexastore, 3Store and Sesame to store the information more efficiently [1, 13, 3]. Inference engines like Reasoning-Hadoop [12] and OWLIM [7] also compress the data with this technique.

Dictionary encoding is not only used within the Semantic Web but also in several other domains. In [15] dictionary encoding is used for image compression. In [8] the authors present some parallel techniques to compress data using an pre-existing dictionary.

In some domains, the dictionary is small enough to be kept in main memory. A good comparison between the performance of different in-memory data structures is given in [16]. From the comparison, it is clear that the hash table is the fastest data structure. [6] proposes a new data structure, called burst trie which maintains the strings in sorted, or near-sorted order, and has performance comparable to the one of a tree.

The decompression algorithm requires that we perform a join on the data but the original MapReduce paradigm does not provide any tool to perform efficient joins. An extension to the MapReduce programming model is called Map-Reduce-Merge [14] and it aims to extend MapReduce to support data joins. Other frameworks, like Pig [9], or Hive [10] are built on top of Hadoop and they provide SQL-like languages to run queries on very large datasets.

8. CONCLUSIONS

In this paper we have proposed a technique to compress Semantic Web statements using the MapReduce programming model. We have shown how we can exploit the features of MapReduce to compress a large amount of data building a dictionary of hundreds of millions of entries.

We have implemented a prototype using the Hadoop framework. We evaluated the performance measuring the runtime using existing datasets and tested the scalability increasing the input size and number of nodes. The evaluation showed that both algorithms are able to compress and decompress a large input with a high throughput and that both algorithms are more efficient for larger inputs. We also noticed that the compression algorithm scales more efficiently than the decompressing algorithm.

We believe that this MapReduce technique has given a major contribution to solve this crucial problem in the Semantic Web and it was essential for our work on large scale reasoning. A remaining challenge is to further improve the scalability of the decompression algorithm and generalize this approach for generic data compression.

9. REFERENCES

[1] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[3] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, page 197, 2003.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–147, 2004.

[5] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3:158–182, 2005.

[6] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20:192–223, 2002.

[7] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM – a pragmatic semantic repository for OWL. In *Proceedings of the Conference on Web Information Systems Engineering (WISE) Workshops*, pages 182–192, 2005.

[8] H. Nagumo, M. Lu, and K. Watson. Parallel algorithms for the static dictionary compression. In *Proc. IEEE Data Compression Conf*, pages 162–171, 1995.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[10] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive-A Warehousing Solution Over a Map-Reduce Framework. VLDB, 2009.

[11] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. Owl reasoning with mapreduce: calculating the closure of 100 billion triples. Currently under submission, 2010.

[12] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. In *Proceedings of the ISWC '09*, 2009.

[13] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment archive*, 1(1):1008–1019, 2008.

[14] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, page 1040. ACM, 2007.

[15] Y. Ye and P. Cosman. Dictionary design for text image compression with JBIG 2. *IEEE Transactions on Image Processing*, 10(6):818–828, 2001.

[16] J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80:2001, 2001.