Reducing I/O Load in Parallel RDF Systems via Data Compression

Jesse Weaver and Gregory Todd Williams

Tetherless World Constellation, Rensselaer Polytechnic Institute, Troy, NY, USA
{weavej3,willig4}@cs.rpi.edu

Abstract. The amount of RDF data published to the web is rapidly growing which has led to an increase in research of systems for handling such vast amounts of data. Employing parallelism has been a common approach, for which parallel I/O of RDF data can be very time-consuming. To reduce I/O load without requiring preprocessing, we propose a syntactic subset of the Turtle syntax called Sterno which is amenable for parallel I/O. We also evaluate the performance gain of using LZO compression, assuming preprocessing of data is allowable in a given use case. Our evaluation shows that Sterno documents improve on our previous work in parallel reading of N-triples documents, and LZO compression significantly reduces document size and read time.

Keywords: RDF, Sterno, LZO, compression, parallel I/O, syntax

1 Introduction

The amount of RDF data published to the web is rapidly growing which has led to an increase in research of systems for handling such vast amounts of data. One approach has been to employ parallelism for processing large datasets (e.g. forward-chained reasoning[5, 6, 8–11, 13], RDF query[14], and so-called "reduction"[15]). In our experience, a significant challenge for such parallel systems is efficient, parallel I/O of RDF data.

In previous work[13–15], we approached the challenge by taking advantage of the simple syntax of N-triples documents[2]. However, the N-triples syntax is very verbose, and as a result, relying on it entails significant I/O costs. In this paper, we propose a syntax for RDF called Sterno[1], a syntactic subset of the Turtle syntax[3] that is amenable to parallel I/O. We also propose using Lempel-Ziv-Oberhumer (LZO) compression for which both compression and decompression can be parallelized. Our evaluation suggests that Sterno documents can achieve a compression ratio (compared to their N-triples equivalents) of under 50%,

[1] The name "Sterno" originated as an abbreviation for sternotherus, a genus of aquatic turtle, the most common species of which typically grows to only 7.5-14 centimeters. This name is chosen to reflect that the Sterno syntax is a small, syntactic subset of the Turtle syntax. Additionally, it is an acronym meaning "Simple, TErse Rdf... NOthing else."

and LZO compression can achieve a compression ratio (compared to the original N-triples document) of under 10%. Furthermore, LZO-compressed N-triples documents can be read much more quickly than uncompressed N-triples documents. LZO-compressed Sterno documents can be read more quickly than uncompressed Sterno documents, but not as quickly as LZO-compressed N-triples documents.

2 Motivation

Traditionally, a common approach to parallel I/O in scientific computations is to divide the input data among processes by essentially striping the data. That is, dividing the input into same-sized chunks of data, every n chunks (where n is the number of processes) is assigned to processes such that process $i-1$ gets the i^{th} chunk. (We view process ranks as being in the range $[0, n-1]$.) Such an approach is made possible by the nature of scientific data which often consists of fixed-sized units of data (e.g., integers, doubles), which leads to fixed-size chunks of data. Therefore, it is easy to determine where one chunk of data begins and ends in a file. That is, if a chunk of data is always K bytes, then every byte offset B such that $B \mod K = 0$ marks the beginning of a chunk of data.

In RDF, the fundamental unit of data is the RDF triple, but a triple consists of three pieces of variable-length data called RDF terms. Some systems use dictionary encoding to first assign to each RDF term a unique, fixed-length identifier (e.g., $[5, 7, 11, 12]$), and then these identifiers are used to transform RDF triples into fixed-sized units of data. The drawback of this approach is that the dictionary must be generated when data is loaded, and dictionary encoding of RDF terms is not a straightforward, parallel computation. For query systems with a "load once, query often" kind of use case, such dictionary encoding is a reasonable solution because the dictionary is built once for a bulk load and need not be rebuilt at query time. However, for one-time computations like closure materialization, preprocessing time for dictionary building can be significant and prohibitive.

In our previous work, we had taken the approach to view RDF data in the N-triples syntax as delimited data (delimited by the end-line sequence[2]) which allowed parallel processes to read independent triple sets using very little coordination at the beginning of reading. This approach is promising because it scales well with the number of processes (given appropriate parallel storage hardware) as shown in [13, 14] since no preprocessing is required and parallel processes read disjoint portions of the N-triples file. However, this approach suffers from excessive consumption of disk space as well as significant disk I/O traffic. N-triples is one of the most verbose RDF syntaxes, but it is the only widely used syntax that provides a simple, delimited format that enables scaling of parallel I/O without preprocessing[3].

[2] An end-line sequence is either a linefeed character or a carriage return character optionally followed by a linefeed character, as defined in N-triples.

[3] If not for its named graph feature, or if the named graph portions of its documents are ignored, TriX could also be used. However, it is more verbose than N-triples.

Therefore, in order to improve compression of RDF data and reduce costs associated with disk I/O, we define an RDF syntax called Sterno that extends N-triples with some features from Turtle. Sterno was designed with four criteria in mind:

1. *Divisibility.* Independent chunks of data must begin and end on easily identifiable boundaries. Thus, parallel processes are able to divide input data independently and read from disjoint portions of the file. This is the only criterion that is *absolutely* necessary for parallel I/O. For Sterno, we use an end-line sequence to delimit independent portions of data.
2. *Directive.* Any directives or global information about the data should occur at the beginning of the file, allowing parallel processes to initially and collectively read directives and then independently read disjoint portions of the remainder of the file. Indeed, parallel processes could initially scan the entire file for all directives, but the I/O cost would be prohibitive. Therefore, we consider this criterion necessary for *efficient*, parallel I/O. In Sterno, this essentially means prefix declarations should occur at the beginning of the document.
3. *Balance.* Each independent chunk of data should correspond to roughly the same number of units of data. This allows parallel processes to improve load-balancing if it is known how many units of data are actually being read. This criterion is not necessary but is desirable. This criterion could be met in a number of ways, but for Sterno, we partially meet this criterion by requiring that each triple reside on its own line.
4. *Simplicity.* Features which add significant complexity to parsing are not adopted. This criterion is not necessary for parallel I/O.

N-triples already meets these criteria (only partially for balance), and so the goal of Sterno is to simply adopt features of Turtle (an ubiquitous, widely-supported, RDF syntax of which N-triples is a subset) that improve compression without further violating these criteria.

To further improve compression without violating divisibility (since it is a block-level compression algorithm), we employ LZO compression on RDF data. Parallel I/O using indexed, LZO-compressed files is not a novel idea[4][5], but its applicability to RDF data has yet to be evaluated. We provide an evaluation on its compression of RDF data and its impact on performance.

3 Compression

3.1 Sterno

Sterno extends N-triples with features from Turtle that improve (or do not decrease) compression without further violating the aforementioned criteria. In this

[4] Hadoop-LZO: `https://github.com/kevinweil/hadoop-lzo` - last accessed February 25, 2011.

[5] Appears to be employed by Twitter: `http://www.cloudera.com/blog/2009/11/hadoop-at-twitter-part-1-splittable-lzo-compression/` - last accessed February 25, 2011.

section, we provide a normative description of Sterno as an extension of N-triples as well as an informative description of Sterno as a restriction of Turtle.

```
<file:///foaf.rdf#me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<file:///foaf.rdf#me> <http://xmlns.com/foaf/0.1/nick> "Andr\u00E9" .
<file:///foaf.rdf#me> <http://xmlns.com/foaf/0.1/age> "40"^^<http://www.w3.org/2001/XMLSchema#integer> .
_:list <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/1999/02/22-rdf-syntax-ns#List> .
_:list <http://www.w3.org/1999/02/22-rdf-syntax-ns#first> "line1\n\tline2 \"quoted string\" " .
_:list <http://www.w3.org/1999/02/22-rdf-syntax-ns#rest> <http://www.w3.org/1999/02/22-rdf-syntax-ns#nil> .
_:contrived <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2002/07/owl#Thing> .
# What a contrived triple.
```

Fig. 1: N-Triples example.

Extending N-triples. In the following, we delineate features of Sterno that extend N-triples. Differences between N-triples and Sterno syntaxes are illustrated in figures 1 and 2.

- **UTF-8 encoding.** A UTF-8-encoded character is at most four bytes long, but its equivalent escaped representation using `\u` or `\U` uses 6 or 10 bytes, respectively.
- **Prefix declarations.** Prefix declarations allow URIs to be abbreviated. Some care is needed in determining what prefix declarations should be made. For example, if a namespace URI occurs at most once, then declaring a prefix declaration for that URI will actually increase the size of the document. Unlike in Turtle, all prefix declarations must occur at the beginning of the document.
- **Implicit datatypes.** Typed literals having a datatype URI of `xsd:integer`, `xsd:double`, `xsd:decimal`, or `xsd:boolean` may be abbreviated. For the three numeric datatypes, their lexical representations may be embedded directly into the text without quotes or datatype URI. For `xsd:boolean`, the text `true` and `false` may be used directly in the text. RDF datasets containing much numeric data stand to gain significantly from this feature.
- **The `a` keyword.** The single character `a` can be used to replace the URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` when it occurs in the predicate position of a triple.
- **The empty collection.** A pair of parentheses `()` can be used to replace the URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#nil>` when it occurs in the subject or object position of a triple.
- **Anonymous blank nodes.** A pair of square brackets `[]` (with nothing in between) may be used to denote a blank node without a label. While this feature has very limited usefulness, there is no reason to exclude it.
- **Complex blank node labels.** This feature does not improve compression, but there is no reason to exclude it. Blank node labels may include any characters allowed by Turtle.

It is important to understand the impact of prefix declarations when compressing to Sterno. As mentioned, if a prefix is declared for a namespace URI that is used at most once, then the document actually becomes larger. If a prefix X is defined for a namespace URI U, then for each URI starting with U and followed by a string that syntactically conforms to the requirements of a "name" in Sterno (we will say such a URI is amenable to prefixing), the URI is replaced by X followed by a colon, followed by the name. Thus given X, U, and m (the number of times a URI amenable to prefixing occurs with namespace URI U), we can determine whether it is beneficial to declare a prefix (including spaces in declaration) using the following equation.

$$\begin{aligned} |\texttt{@prefix } X\texttt{: <}U\texttt{> .\n}| + m\,|X\texttt{:}| &< m\,|\texttt{<}U\texttt{>}| \\ 8 + |X| + 3 + |U| + 4 + m(|X| + 1) &< m(1 + |U| + 1) \\ 15 + |X| + |U| + m\,|X| + m &< 2m + m\,|U| \\ 15 + |X| + |U| &< m + m\,|U| - m\,|X| \\ \frac{15 + |U| + |X|}{1 + |U| - |X|} &< m \end{aligned}$$

```
@prefix mine: <file:///foaf.rdf#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
mine:me a foaf:Person .
mine:me foaf:nick "André" .
mine:me foaf:age 40 .
_:list a rdf:List .
_:list rdf:first "line1\n\tline2 \"quoted string\" " .
_:list rdf:rest () .
[] a <http://www.w3.org/2002/07/owl#Thing> .
# What a contrived triple.
```

Fig. 2: Sterno example.

Restricting Turtle. In the following, we delineate the features of Turtle that are excluded from Sterno and give reason (i.e., which of the aforementioned criteria are violated) for their exclusion. Figures 2 and 3 illustrate differences in Sterno and Turtle.

- **Whitespace leniency.** For divisibility, Sterno preserves the requirement (from N-triples) that each triple must end with an end-line sequence. For simplicity, Sterno preserves the requirement that whitespace must occur between subject and predicate and between predicate and object.
- **Prefix declaration placement leniency.** To meet the directive requirement, all prefix declarations must occur at the beginning of a Sterno document before the occurrence of any triples.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<#me> a foaf:Person ; foaf:nick "André" ; foaf:age 40 .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
( """line1
   line2 "quoted string" """ ) a rdf:List .
[]a<http://www.w3.org/2002/07/owl#Thing> . # What a contrived triple.
```

Fig. 3: Turtle example assuming document is stored at `file:///foaf.rdf`.

- **Triple-quoted literals.** The purpose of triple-quoted literals in Turtle is to allow vertical whitespace and quotes in the literal. Sterno excludes this feature to maintain divisibility.
- **Property and object lists.** Sterno excludes these features to improve balance. Even when kept on the same line with their respective subjects (thus maintaining divisibility), property and object lists have the potential to severely skew the number of triples that occur within a chunk of data (that is, a single line). Exclusion of this feature also improves simplicity.
- **Non-empty collections.** To improve balance, Sterno excludes the collection syntax (except for the empty collection) because a non-empty collection represents two triples for each item in the collection.
- **Relative URIs.** Resolving a relative URI can be expensive compared to a simple prefix lookup. This feature is excluded in favor of simplicity, and thus, the `@base` declaration from Turtle is unnecessary.
- **Trailing comments.** In Turtle, comments may occur at the end of any line. For simplicity, Sterno preserves the requirement of N-triples that each comment must occur on its own line.

3.2 LZO

LZO is a lossless data compression algorithm which has two important properties: block-level compression and fast decompression.

A block compression algorithm is a compression algorithm in which individual, fixed-size blocks of data are compressed independently of each other; therefore, it is parallelizable. However, there is no obvious delimiter between compressed blocks, making it difficult to divide a block-compressed file for parallel reading and decompression. The solution is to create an index file at the time of compression. The index file consists of byte offsets into the compressed file where blocks begin. The index enables parallel processes to divide a block-compressed file into independent portions that can be decompressed in parallel.

Decompression of LZO-compressed data is extremely fast so that the combined cost of reading the compressed data and decompressing it is usually less than the I/O cost of reading uncompressed data. Compression speed is reasonable but not insignificant (often referred to as "pretty good"). Thus, unless the dataset is already LZO compressed (which is not expected to be the case), then compressing the dataset introduces preprocessing.

4 Implementation

Our system aims to generically support parallel, RDF-processing systems in which each process initially reads a disjoint portion of the input data[6]. As such, we rely on the MPI interface to abstract away from the details of the underlying hardware, trading off potential performance advantages for genericalness and portability. The system has an iterator interface which returns triples in a normalized format – essentially a string in the N-triples syntax except that it is UTF-8 encoded. Thus, when reading N-triples documents, `\u` and `\U` escape sequences are converted to UTF-8-encoded characters. When reading Sterno documents, abbreviated terms must be expanded (e.g., `a`, `()`, abbreviated URIs), and any `\u` and `\U` escaped sequences must be UTF-8-encoded. Thus, although a Sterno document may be smaller than its N-triples counterpart, it may incur more processing time to normalize. Comments are ignored.

The system is implemented in C++ using MPI for interprocess communication and parallel I/O. For LZO compression/decompression, we used the LZO1X algorithm from the open source LZO 2.04 library[7]. The implementation details regarding I/O are given in the remainder of this section.

Regardless of the format of the input file, parallel reading essentially begins the same way. As a general heuristic for efficient I/O, we desire each process to read only entire disk pages along page boundaries. These boundaries may not (and likely will not) align with data boundaries. Given a file of P pages, each of n processes is assigned $\lfloor P/n \rfloor$ pages, and $P \mod n$ processes are each assigned an additional page. Each process' pages are consecutively addressed (i.e., logically adjacent), so the file is effectively divided up into n portions of (logically) consecutive data. We will refer to these portions of the file as segments, and each process holds a byte offset that points to the beginning of its segment and knows the length of its segment. Note that a segment may not correspond to an independent set of triples. Here we have discussed page-aligned segments. Each segment may begin (except for the first one) and/or end (except for the last one) with a fragment of a data chunk. The method of handling these fragments depends on whether the input data is delimited (N-triples and Sterno) or indexed (indexed LZO-compressed).

For Sterno documents, the beginning of the file must be read for prefix declarations. In this case, the processes collectively read from the beginning of the file using `MPI_File_read_at_all_begin` and `MPI_File_read_at_all_end`, loading prefix mappings in memory. When a triple is reached, there are no more prefixes, and the remainder of the file is divided into segments as previously described. For N-triples documents, the entire file is divided into segments since there are no prefix declarations or other directives. In order to ensure that every process has only complete triples, each process i sends the beginning fragment (indicated

[6] Assuming input document does not repeat triples. Technically, if there are duplicate triples in the input document, then each process will not necessarily have a disjoint portion of the *RDF* data.

[7] `http://www.oberhumer.com/opensource/lzo/` - last accessed February 25, 2011.

by the first occurrence of an end-line sequence) to process $i-1$ (for $i > 0$) and receives a fragment from process $i+1$ (for $i < n-1$). `MPI_File_iread_at` is used for reading, and `MPI_Isend` and `MPI_Irecv` are used for fragment passing.

For LZO-compressed, Sterno documents, prefix declarations are read collectively by all processes just as before except this time decompressing as data is read. When the first triple is encountered, processes note which block – say the k^{th} block – contains the first triple. The index file is then logically divided into segments of offset integers (rather than pages as before) ignoring the first $k-1$ offsets, and each process reads only the first offset integer in its segment to determine the beginning of its block-aligned segment in the LZO-compressed data. Instead of passing fragments, offsets are passed where process i receives an offset from process $i+1$ (for $i < n-1$) to be used as the upper bound for its segment. Each process then reads blocks from its segment (reading along page boundaries) and decompresses them. Process 0 has the k^{th} block at the beginning of its segment, and it may contain some of the last prefix declarations. These prefix declarations are ignored like comments. Logically speaking, each process' segment of the LZO-compressed file corresponds to a block-aligned segment of the uncompressed document which may begin and/or end with fragments. Thus, processes perform fragment passing before continuing to read and decompress the remainder of their segments. Reading LZO-compressed, N-triples documents happens in the same fashion except without the initial, collective reading of prefix declarations.

5 Evaluation

Our evaluation is designed to measure compression of RDF data (Sterno and LZO) and the impact compression has on the performance of parallel reading[8] of RDF data. Therefore, we report compression ratios (compared to the original N-triples file) and times for reading.

For evaluation, we use the 2010 Billion Triple Challenge dataset[9] (BTC2010) which contains 3,171,793,030 quads. We stripped out the context URIs since we consider only triples. BTC2010 represents a worst-case scenario for compression using prefix declarations in Sterno since a large number of namespace URIs are used. Note that the document contains a significant number of duplicate triples. This does not take away from the validity of our evaluation since (1) Sterno compression does not benefit directly from duplicate triples but rather from a small number of namespaces, and (2) LZO compression is a block compression algorithm and thus only duplicate triples within a single block (much less likely) will benefit LZO compression. Also, note that BTC2010 is a "messy" dataset containing some invalid URIs (e.g., `<mailto:Ron Alford>`) and blank node labels (e.g., some contain '`.`'). Rather than attempt to somehow repair the data (which would be both arduous and dubious since we do not know the valid versions of

[8] Note that the data is merely read triple-by-triple; each triple is immediately discarded after being read.

[9] `http://km.aifb.kit.edu/projects/btc-2010/` - last accessed February 27, 2011.

what the publishers originally intended), our parsers have been implemented to tolerate these subtle syntax violations for the purposes of this evaluation.

For near optimal compression using Sterno, we determined which namespaces in the BTC2010 should have prefixes declared using the formula from section 3.1. Using a Perl script, we scanned through the N-triples file seeking URIs amenable to prefixing. Each time one was found, its namespace URI was printed. We then used Unix commands `sort` and `uniq` to rank the namespace URIs by descending frequency. We then used another Perl script to generate the prefix declarations using the frequencies and the formula from section 3.1. The Perl script uses a base-36 counter to generate prefix names, skipping any values for which the most significant digit is a base-10 numeric digit (since a prefix name cannot begin with a numeric digit). This allowed us to keep prefix names reasonably short. The entire process of generating prefix declarations took roughly ten hours and generated 5,770,712 prefix declarations. In practice, we would expect data publishers to already know which namespace URIs are worth prefixing.

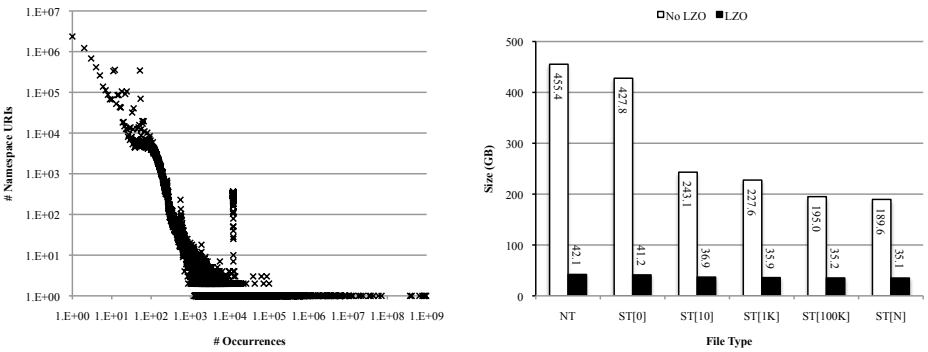
To evaluate the impact of the number of prefix declarations, we generated Sterno files using 0, 10, 1K, 100K, and N=5,770,712 prefix declarations (using the most frequent namespace URIs). The frequency of namespace URIs is given in figure 4a where each plotted point (x, y) indicates that y unique namespace URIs occur exactly x times in the dataset. Only seven namespace URIs occur over a hundred million times. Therefore, we have 12 evaluation scenarios: N-triples (NT) and the five Sterno variants (ST[M] where M is the number of prefix declarations), each in LZO-compressed and non-LZO-compressed forms. The sizes for the 12 documents are given in figure 4b. Without any prefix declarations, Sterno achieves very little compression, 94% of the N-triples document. However, using only ten prefix declarations (for the ten most common namespace URIs) results in a Sterno document about 53% of the size of the N-triples document. ST[N] achieves the greatest compression of roughly 42%. The LZO-compressed versions are much smaller, ranging from 9.3% for N-triples down to 7.7% for ST[N].

We ran our experiments for parallel reading on the Opteron blade cluster at the Computational Center for Nanotechnology Innovations[10] (CCNI). Each node was an IBM LS21 blade server with two dual-core 2.6 GHz Opteron processors, 16 GB of memory, and Infiniband interconnects. The operating system was RedHat 5, and the MPI version was MVAPICH2 1.4. The GPFS[11] file system was used (with Gigabit Ethernet connection to the cluster) for all data storage and parallel I/O. The GPFS setup has 48 storage nodes, each with several LUNs attached. Data and metadata replication are used, but at the cost of performance and space. The GPFS is shared with all machines at the CCNI, and thus, there is always potential for contention with other jobs being run.

The results are given in figure 5. Note that the charts use log-log scale. In figure 5a, the parallel read times are given for files that are not LZO-compressed.

[10] `http://www.rpi.edu/research/ccni/` - last accessed February 28, 2011.

[11] `http://www-03.ibm.com/systems/software/gpfs/` - last accessed February 28, 2011



(a) Namespace URI Frequencies (b) File sizes

Fig. 4: Namespace URIs and Compression

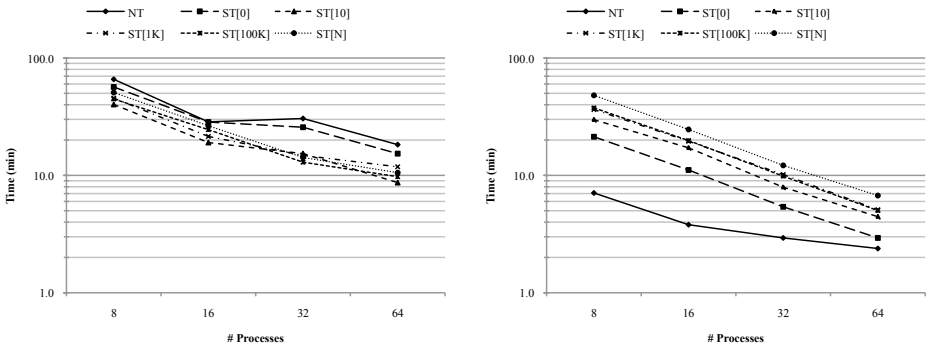
Note that N-triples takes the longest to read while ST[10] is the fastest (except for 32 processes). It is not surprising that it generally takes longer to read ST[1K], ST[100K], and ST[N] than ST[10] since there is less I/O savings to be had by adding more prefix declarations but there is increased cost of expanding URIs when normalizing data. The increase for NT and lack of decrease for ST[0] when moving from 16 to 32 processes is unusual. Preliminary investigation suggests that there is a bottleneck in the hardware system when moving from 16 to 32 processes. Further investigation into a definitive cause is left for future work. On 64 processes, NT took 18.3 minutes ($\approx 2.9 \times 10^6$ triples/sec), and ST[10] took 8.7 minutes ($\approx 6.1 \times 10^6$ triples/sec).

In figure 5b, the parallel read times are given for files that are LZO-compressed. Interestingly, NT is the fastest, benefiting the most from high-redundency when performing compression and requiring only fast LZO decompression without expansion of abbreviated RDF terms. As the number of prefix declarations increases, the time increases. However, the time to read LZO-compressed versions are always faster than for their non-LZO-compressed counterparts. For 64 processes, NT took 2.4 minutes ($\approx 2.2 \times 10^7$ triples/sec), and ST[N] took 6.7 minutes ($\approx 7.9 \times 10^6$ triples/sec).

6 Related Work

Although there are many syntaxes for RDF, to our knowledge, there is no related work on developing RDF syntaxes for the explicit purpose of improving parallel I/O. The most relevant work is dictionary encoding of RDF terms using Hadoop by Urbani *et al.*[12]. Including the compressed triples with the dictionary (presumably uncompressed), a compression ratio ranging from 25-33% was achieved. Other work[1, 4] on compressing RDF data has been done without re-

gard for parallel I/O, focusing primarily on achieving high compression in a data structure that offers basic functionality for supporting higher-level operations.



(a) Without LZO Compression (b) With LZO Compression

Fig. 5: Times for Parallel Reading for 8-64 processes.

7 Conclusion

To reduce I/O costs for parallel, RDF-processing systems, we proposed the RDF syntax Sterno which is readily amenable (that is, no preprocessing required) for parallel I/O and a subset of the common Turtle syntax. We have shown that it is able to achieve compressions as small as 42% for the diverse BTC2010 dataset, and that reading Sterno documents in parallel is usually faster than reading N-triples documents. Additionally, we showed that LZO compression can be used to significantly reduce document size and improve performance of parallel reading, particularly for N-triples documents. If preprocessing is tolerated, then it is always beneficial to LZO-compress the document. The best read performance is achieved with LZO-compressed, N-triples documents.

In short, for one-time jobs or use cases that do not tolerate preprocessing, Sterno is the best choice for compression and read time. For systems that will process the input file multiple times, it is likely worthwhile to LZO-compress the data (ideally, in N-triples syntax) beforehand.

Acknowledgments. We thank the staff at the CCNI for their support, especially Dr. Lindsay Todd for his insights into the details of their systems configurations.

References

1. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "Bit"loaded: A scalable lightweight join query processor for RDF data. In: Proceedings of the 19th International World Wide Web Conference (2010)
2. Beckett, D., Grant, J.: RDF Test Cases. W3C recommendation, W3C (February 2004), http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/
3. Beckett, D., Berners-Lee, T.: Turtle - Terse RDF Triple Language. W3C team submission, W3C (January 2008), http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/
4. Fernández, J.D., Martínez-Prieto, M.A., Gutierrez, C.: Compact Representation of Large RDF Data Sets for Publishing and Exchange. In: Proceedings of the 9th International Semantic Web Conference (2010)
5. Goodman, E.L., Mizell, D.: Scalable in-memory rdfs closure on billions of triples. In: Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (2010)
6. Hogan, A., Pan, J.Z., Polleres, A., Decker, S.: Saor: Template rule optimisations for distributed reasoning over 1 billion linked data triples. In: Proceedings of the 9th International Semantic Web Conference. (2010)
7. Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM – a Pragmatic Semantic Repository for OWL. In: Proceedings of the Conference on Web Information Systems Engineering Workshops. pp. 182–192 (2005)
8. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In: Proceedings of the 19th International World Wide Web Conference (2010)
9. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. Journal of Web Semantics 7(4), 305–316 (2009)
10. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: Proceedings of the 37th International Conference on Parallel Processing. pp. 75–82 (2008)
11. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In: Proceedings of the 7th Extended Semantic Web Conference (2010)
12. Urbani, J., Maassen, J., Bal, H.E.: Massive semantic web data compression with mapreduce. In: Proceedings of the 1st International Workshop on MapReduce and its Applications (2010)
13. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: Proceedings of the 8th International Semantic Web Conference. pp. 682–697 (2009)
14. Weaver, J., Williams, G.T.: Scalable RDF query processing on clusters and supercomputers. In: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (2009)
15. Williams, G.T., Weaver, J., Atre, M., Hendler, J.A.: Scalable Reduction of Large Datasets to Interesting Subsets. Billion Triples Challenge at the 8th International Semantic Web Conference (2009)