Scalable Reduction of Large Datasets to Interesting Subsets

Gregory Todd Williams *, Jesse Weaver, Medha Atre, James A. Hendler

Tetherless World Constellation, Rensselaer Polytechnic Institute, Troy, NY, USA

Abstract

With a huge amount of RDF data available on the web, the ability to find and access relevant information is crucial. Traditional approaches to storing, querying, and reasoning fall short when faced with web-scale data. We present a system that combines the computational power of large clusters for enabling large-scale reasoning and data access with an efficient data structure for storing and querying the accessed data on a traditional personal computer or other resource-constrained device. We present results of using this system to load the 2009 Billion Triples Challenge dataset, materialize RDFS inferences, extract an "interesting" subset of the data using a large cluster, and further analyze the extracted data using a personal computer, all in the order of tens of minutes.

Key words: Billion Triples Challenge, Scalability, Parallel, Inferencing, Query, Triplestore

1. Introduction

The semantic web is growing rapidly with vast amounts of RDF data being published on the web. RDF data is now available from many sources across the web relating to a huge variety of topics. Examples of these RDF datasets include the Billion Triples Challenge [1] dataset (collected by a webcrawler from RDF documents available on the web), the Linked Open Data project [2] (in which a number of independent datasets are linked together using common URIs), and the recent conversions of governmental datasets to RDF [3].

With so much structured data available on the web, it is important to be able to find, extract and use just the data that is relevant to a particular problem in a reasonable amount of time. Several factors make this difficult, including different modeling of similar data across datasets, the lack of tools to perform reasoning on large datasets, and the challenge of interacting with and analyzing large datasets in realtime. Extracting specific subsets of interest from very large datasets can alleviate the challenge of analysis, but can also be a time consuming task despite often being simplistic in nature. Many tools require lengthy loading and indexing times which cannot be amortized when used to support a small number of queries meant to extract a subset of data.

We propose a composition of approaches that addresses these issues. In particular, we present a system that makes use of a large cluster/supercomputer to allow RDFS materialization of inferences over large datasets and efficient extraction of specific data subsets. We also make use of a compressed triplestore that may be used to directly answer interactive queries on a wide range of (non-cluster) systems. We evaluate this system on the Billion

* Corresponding author. Tel: +1 518 276 4431

Email addresses: willig4@cs.rpi.edu (Gregory Todd Williams), weavej3@cs.rpi.edu (Jesse Weaver), atrem@cs.rpi.edu (Medha Atre), hendler@cs.rpi.edu (James A. Hendler).

[1] http://challenge.semanticweb.org/, accessed 2010/02/01.

[2] http://linkeddata.org/, accessed 2010/02/01.

[3] http://data-gov.tw.rpi.edu/ and http://data.gov.uk/ being the most prominent, accessed 2010/02/01.

Triples Challenge 2009 (BTC2009) dataset [4].

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents a use case that motivates the work presented in the rest of this paper. Section 4 describes the core components of our system, and in section 5 we evaluate our system by performing inferencing on the dataset, extracting relevant data, and loading the reduced dataset into a compressed triplestore. Finally, section 6 concludes the paper and discusses work we hope to pursue in the future.

2. Related Work

Our system consists of three components: parallel inferencing, parallel query, and a compressed triplestore. In this section we present works related to these components.

The parallel inferencing work most related to ours is "Scalable Distributed Reasoning Using MapReduce" and WebPIE by Urbani et al.[38,39], MaRVIN[3,29,30], the parallel Web Ontology Language (OWL) inferencing work by Soma and Prasanna[35,36], and the OWL knowledge base partitioning work by Guo and Heflin[15]. Less related parallel inferencing works include reasoning over distributed hash tables [14,23]; composition of approximate, anytime reasoning algorithms executed in parallel mentioned at the end of [33]; proposals of parallel computation techniques for ontology reasoning [8]; and approaches for exploiting vertical parallelism in tableaux-based description logic reasoning [25].

The MapReduce-based systems by Urbani et al.[38,39] map triples to the same reducer if, together, they can produce inferences. In [39], they report being able to produce the partial RDFS closure (that is, using only the more interesting RDFS entailment rules) over a dataset of 864.8 million triples in just under one hour using 32 processors. In [38], they show reasoning with so-called "OWL Horst" (also known as pD*) semantics[37] over 10 billion Lehigh University Benchmark[16] (LUBM) triples in five hours using 64 processors. On UniProt (a real world dataset of 1.51 billion triples), they report OWL Horst inferencing in 6.1 hours on 32 nodes.

Compared with MapReduce-based approaches, we opt for a system in which file IO operations and inter-processor communication are implemented directly, avoiding higher-level abstractions. This choice allows for a more efficient implementation which does not have to conform to a strict separation of map and reduce tasks and can more naturally encode traditional database operations such as joins. However, by avoiding MapReduce, we add complexity and give up features such as fault tolerance. Recently, HadoopDB[2] has shown promising results by combining the benefits of MapReduce and traditional relational databases. Such a hybrid architecture is worth further study as it applies to scalable queries over RDF data.

MaRVIN[3,29] provides sound, anytime, and eventually complete RDFS reasoning using a "divide-conquer-swap" strategy. Every processor uses a reasoner and processes a fraction of the data to produce local inferences. A scoring mechanism is used to determine which triples are most useful for further inferencing, and these triples are exchanged between processors in an attempt to mix up the triples and find more inferences. Their evaluation in [29] shows presumably nearly complete reasoning on 14.9 million SwetoDBLP [5] triples in roughly 23 minutes using 64 processors.

Soma and Prasanna[36] present an approach to rule-based reasoning with OWL Horst semantics. Before parallel inferencing occurs, a fair amount of preprocessing is required. OWL ontologies are compiled into rules and a partitioning is determined ahead of time, after which processors collaborate to produce inferences. While it is uncertain what execution times were actually achieved, they indicate that they were able to achieve a speedup of 18 for 16 processors on the LUBM 10-university dataset (approximately one million triples). Other datasets evaluated, however, did not see as much speedup.

Guo and Heflin[15] present a limited partitioning scheme for OWL knowledge bases that allows for a subset of OWL Lite reasoning over independent ABoxes with the same TBox. Our reasoning work employs a similar division between assertions and ontology.

The SAOR work [20–22] also employs a division of assertions and ontology similar to the one used in our reasoning system. While SAOR focuses on a scalable, disk-based, reasoning system using some RDFS and OWL-based rules on a single machine, our reasoning system differs greatly from SAOR in

[4] http://vmlion25.deri.ie/index.html, accessed 2010/02/01.

[5] http://lsdis.cs.uga.edu/projects/semdis/swetodblp/, accessed 2010/02/01.

that it is not disk-based, it is parallel, and it implements more of the RDFS entailment rules and none of the OWL-based rules.

Related work on parallel and/or distributed RDF query processing includes RDFPeers[9], Continuous RDF Query Processing over DHTs[24], parallel evolutionary query answering suggested in [28], YARS2[18], Virtuoso[12,13], GridVine[10], Clustered TDB[31], and 4store[17]. We differ from these approaches in that we address parallel query processing as a process involving both the loading and computation over RDF data rather than a process occurring over a persistent storage system. DeWitt and Gerber[11] introduce a parallel hash-join algorithm and demonstrate its effectiveness in parallel join execution. We adapt this parallel hash-join algorithm to the specifics of our parallel RDF query infrastructure and use it in query evaluation.

BigOWLIM [6] reports reasoning and query times on large datasets. BigOWLIM 3.1 is a semantic web repository that supports OWL Horst reasoning. They report that loading and inferencing on LUBM90k (12.03 billion triples) took 290 hours. They also report that loading, inferencing, and query evaluation of the LUBM benchmark on LUBM8k (1.07 billion triples) took 15.2 hours.

Many non-cluster systems exist for indexing and query processing of RDF data. Two general purpose databases recently used for RDF processing are the column-stores C-store[1] and MonetDB[34]. These systems partition RDF data vertically for each predicate, i.e., they store the RDF triples in separate predicate tables and create subject-object indexes on each table. This greatly enhances the performance of queries with bound predicate terms, but has difficulty handling queries with unbound predicates. Systems like RDF-3X[26,27] and Hexastore[42] go one step further by fully indexing RDF data, producing all six possible index orderings over the subject, predicate, and object terms. RDF-3X additionally implements index compression and many other join optimization techniques like join selectivity estimation, sideways-information-passing, and bloom filters for hash-joins.

The compressed triplestore we employ differs from these systems in its processing of join queries. It follows a two-phase approach of query processing: pruning the candidate RDF triples by following a procedure similar to the concept of semi-joins[7], and generating the final results without building intermediate join tables by following a procedure similar to multi-way joins[32].

[6] http://ontotext.com/owlim/benchmarking/lubm.html, accessed 2010/05/02.

3. Motivation

In the following sections, we rely on a specific use case supported by the BTC2009 dataset. We consider a scenario in which a user desires to discover the people mentioned in the dataset, and to retrieve specific facts about them. This involves discovering the resources that represent people as well as extracting the properties of those people in which we are interested. As a simple example we restrict ourselves to extracting all the people in the dataset with both names and email addresses.

4. Methodology

The process of loading, reasoning over, and extracting data from a dataset is performed in several discrete steps. This process starts by mapping the various ways in which data is modeled in the dataset to a single model with which we can interact directly. We encode our mapping as an RDFS upper ontology, and we combine the upper ontology with other relevant ontologies to create our target ontology. We then generate RDFS inferences with respect to our target ontology to derive new triples that use the terms of our upper ontology. Following materialization, we perform a basic graph pattern query to extract the specific data of interest (in this case people with both full names and email addresses) and produce a reduced dataset. Finally, we take the reduced dataset and store it in a BitMat[4–6] compressed triplestore where it may be directly and efficiently queried. The flow of data for this process is shown in Figure 1 and described in more detail below.

4.1. *Schema Mapping*

Information about people is represented in different ways in the BTC2009 dataset, including the use of the FOAF [7], SIOC [8], DBpedia [9], and AKT [10]

[7] Friend of a Friend, http://xmlns.com/foaf/spec/, accessed 2010/02/01.
[8] Semantically-Interlinked Online Communities, http://sioc-project.org/ontology, accessed 2010/02/01.
[9] http://wiki.dbpedia.org/Ontology, accessed 2010/02/01.
[10] Advanced Knowledge Technologies, http://www.aktors.org/publications/ontology/, accessed

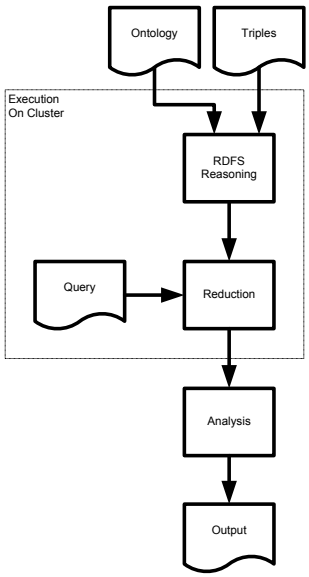

Fig. 1. The flow of data through our system

ontologies. We create a simple upper ontology to bring together concepts and properties pertaining to people. For example, we define the class up:Person which is defined as a superclass to existing person classes, e.g., foaf:Person. We do the same for relevant properties, e.g., up:full_name is a superproperty of akt:full-name. Note that "up" is the namespace prefix for our upper ontology. The upper ontology, along with other BTC2009-related ontologies, are provided alongside the BTC2009 dataset to allow inferencing of triples that use the terms in our upper ontology. Note that it may not always be necessary for the user to create an upper ontology if existing ontologies include concepts and properties that are well-connected to other concepts and properties.

The upper ontology we use during inferencing is included in full in the Appendix.

4.2. *Reasoning*

Weaver and Hendler[40] declare the following definitions and assumption:

Definition 1 *The **finite axiomatic triples** are the RDF and RDFS axiomatic triples [19] minus any triples that describe resources of the form rdf:_#.*

Definition 2 *The **finite RDFS rules** are the set of rules concerning literal generalization (lg, gl) and the RDF and RDFS entailment rules (rdf1-2, rdfs1-13)[19], and also the following rule which we call cmp:*

(?n rdf:type rdfs:Resource) ∧
?n is of the form rdf:_# ⟶
(?n rdf:type rdfs:ContainerMembershipProperty)

Definition 3 *The **finite RDFS closure** is defined the same way as the RDFS closure [19] with the following exceptions: (1) the finite axiomatic triples are used instead of all of the RDF and RDFS axiomatic triples; and (2) rule cmp is included in the last step of rule applications.*

Assumption 4 *Other than those mentioned in the axiomatic triples or those produced from the entailment rules, the resources in the RDF and RDFS vocabulary have no superclasses, subclasses, superproperties, subproperties, domains, or ranges.*

Assumption 4 further restricts the definition of finite RDFS closure given in definition 3 by not including entailments that result from "extending" the RDFS terms. For example, in complete RDFS reasoning, a triple like {`:mySubProp rdfs:subPropertyOf rdfs:subPropertyOf`} is a valid triple that leads to further inferencing by creating more rdfs:subPropertyOf triples. Weaver and Hendler preclude complete inferencing on such triples in their inferencing system. [11] Therefore, we define the following for reference in this article:

Definition 5 *We define the **substantive RDFS closure** as the finite RDFS closure (definition 3) excluding some or all entailments from "extending" RDFS terms (assumption 4).*

Definition 6 *We define the **partial, substantive RDFS closure** as the substantive RDFS closure excluding the following "less interesting" RDFS entailment rules:*

- *Rules lg/gl: assigning unique blank nodes to all literals and vice versa.*
- *Rule rdfs1: all literals are type rdfs:Literal.*
- *Rules rdfs4a/b: everything is an rdfs:Resource.*
- *Rule rdfs6: each property is a subproperty of itself.*
- *Rule rdfs8: each class is a subclass of rdfs:Resource.*
- *Rule rdfs10: each class is a subclass of itself.*

However, note that with the exclusion of rdfs4a/b, rule cmp no longer produces the intended inferences. Therefore, we redefine rule cmp more narrowly as:

2010/02/01.

[11] Some, but not necessarily all, inferences from such triples will actually be produced. However, the partitioning scheme presented in [40] does not ensure completeness of such inferencing.

(?n rdf:type rdf:Property) ∧
?n is of the form rdf:_# ⟶
(?n rdf:type rdfs:ContainerMembershipProperty)

Weaver and Hendler show that the substantive RDFS closure of a dataset can be produced in an embarrassingly parallel [12] fashion. Given a set of triples divided into ontology and assertions, each processor is given the entire ontology and a portion of the assertions; call such a partitioning of triples an "abox partitioning." (This is similar to the "independent abox partitioning" scheme presented by Guo and Heflin[15]. Such a division of triples is also employed by SAOR [20–22] to improve performance of sequential reasoning.) Certain rules, termed "Abox Partitioning Safe" (APS) rules, can be applied independently to each partition of an abox partitioning to produce all possible inferences. Based on [40], the finite RDFS rules are all APS for producing the substantive RDFS closure. Breaking down the BTC2009 dataset into separate chunks (such that the union of the chunks create the whole BTC2009 dataset), we individually compute the partial, substantive RDFS closure of these chunks. This allows us to efficiently materialize the closure with respect to the target ontology in parallel.

The greatest benefit of RDFS reasoning (for our purposes) is type inference from class hierarchy, assertion inference from property hierarchy, and type inference from domains and ranges of properties. This allows us to efficiently generate inferences that use terms in our upper ontology. (Note that a simple rewriting of terms does not suffice. For example, replacing foaf:Person with up:Person will not capture any individuals typed as a subclass of foaf:Person.)

Note that the output may contain duplicate triples between processors. The evaluation in [40] suggests that the percentage of triples that occur more than once decreases with the number of assertions and increases with the number of processors. Rules *rdfs4a/b* and *rdfs1* were responsible for most of the duplicate triples in [40], and those rules were excluded from the work presented herein. Duplicates are allowed in the input of the query system described in the following section.

[12] An embarrassingly parallel computation is "the ideal computation from a parallel computing standpoint—a computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor." [43]).

4.3. *Reduction*

Once the partial, substantive RDFS closure of the dataset is produced, we need to extract only the data in which we are interested. We employ a parallel RDF query approach described by Weaver and Williams[41], allowing us to efficiently perform a query over said closure of the BTC2009 dataset to extract just the information we need, greatly reducing the number of triples for later storage and analysis.

4.3.1. *Dataset Loading*

The reduction process makes use of the parallel nature of clusters to achieve much higher loading rates than would otherwise be possible. As described in [41], each processor in the cluster is responsible for loading a single, contiguous chunk of the input data and maintaining local, in-memory indexes over this data. Importantly, no global data structures are used to index the input data, allowing each processor to load data independently with no inter-process communication for index creation. This design dramatically improves loading time over traditional approaches which are slowed by the need to maintain globally unique identifiers and indexes over the input data.

Each processor in the cluster holds its local triple data as a set of 64-bit node identifiers and a mapping from these identifiers to full node values. Since node identifiers are not unique across the cluster, whenever a processor sends intermediate results to another processor it must send full node values as opposed to the node identifiers. This is done by converting the node identifiers into string representations of the node values before performing the send operation, allowing the receiving processor to assign its own local identifiers to the received node values. While it incurs a higher communication cost during query evaluation, this design allows for the (previously discussed) highly parallelized loading operation which is fast enough that we can afford to load data for every query and enables exploratory querying. By making data loading an inexpensive operation, we also allow our system to take advantage of the up-to-date state of the data without costly index maintenance.

We investigated an alternative approach involving the use of globally unique node identifiers, but the distribution, management, and lookup operations for these identifiers proved to be an extremely

time-consuming process, one which we found to be prohibitive.

4.3.2. *Query Evaluation*

During query evaluation, our system makes use of a parallel hash join to distribute intermediate results and spread execution cost across processors in the cluster. In the following discussion and algorithms we use the following notation from SPARQL [13]: a solution mapping μ is a partial function from variables to RDF terms, and a set of results from a query is a multiset, Ω, of solution mappings. Algorithm 2 uses "**pardo**" to mean "do in parallel".

During a two-way join, the two subqueries are executed independently on each processor i, regardless of the dataset held locally on each processor. The results of these subqueries ($\Omega_{1,i}$ and $\Omega_{2,i}$) are then redistributed among the processors in such a way as to ensure that the appropriate results for the join are colocated. This is done by hashing on the values of the variables shared between the two result sets. After distributing the results, each processor performs the join locally on the received results ($\Omega'_{1,i}$ and $\Omega'_{2,i}$). The redistribution is illustrated in Algorithm 1, while the overall parallel hash-join is illustrated in Algorithm 2 (assuming $\Omega_{1,i}$ and $\Omega_{2,i}$ are available as input after executing the two subqueries).

For the query evaluation of a basic graph pattern containing n triple patterns, the parallel hash-join algorithm is run $n-1$ times, joining the triple patterns in a so-called left-deep query execution plan. The union on line 6 of Algorithm 2 represents the logical, complete results of the join. However, instead of performing this union, each $\Omega'_{1\bowtie 2,i}$ can simply become the input $\Omega_{1,i}$ for the subsequent join with the results from the next triple pattern in the query execution plan. Further details are included in [41].

Algorithm 1: Distribute Solution Mappings (distmu)

Input: A (local) multiset of solution mappings Ω_i, a set of join variables V, and a number of processors p.
Output: A multiset of solution mappings Ω'_i from redistribution.

1 $\Omega'_i = \emptyset$
2 **foreach** $\mu \in \Omega_i$ **do**
3 $\quad \mu' = project(V, \mu)$
4 $\quad recvr = hash(\mu') \% p$
$\quad$ `// Send` μ `to` $recvr$`.`
5 $\quad send(recvr, \mu)$
$\quad$ `// Receive solution mappings from any`
$\quad$ `// processor.`
6 $\quad$ **while** $recv(*, \mu_r)$ **do**
7 $\quad\quad$ add μ_r to Ω'_i
8 $\quad$ **end**
9 **end**
10 **return** Ω'_i

Algorithm 2: Parallel Hash Join

Input: Two multisets of solution mappings $\Omega_1 = \bigcup_{i=0}^{p-1} \Omega_{1,i}$ and $\Omega_2 = \bigcup_{i=0}^{p-1} \Omega_{2,i}$, the set of join variables V, and a number of processors p.
Output: A multiset of solution mappings $\Omega_{1\bowtie 2} = join(\Omega_1, \Omega_2)$.

`// Loop indicates parallelism where` i `is`
`// the rank of the processor.`
1 **for** i *= 0 to* $p-1$ **pardo**
$\quad$ `// Ensure solution mappings that can`
$\quad$ `// join meet on same processor.`
2 $\quad \Omega'_{1,i} = distmu(\Omega_{1,i}, V, p)$
3 $\quad \Omega'_{2,i} = distmu(\Omega_{2,i}, V, p)$
$\quad$ `// Join local solution mappings.`
4 $\quad \Omega'_{1\bowtie 2,i} = join(\Omega'_{1,i}, \Omega'_{2,i})$
5 **end**
6 **return** $\bigcup_{i=0}^{p-1} \Omega'_{1\bowtie 2,i}$

4.4. *Analysis*

Having extracted only the data in which we are interested, we now wish to store it in such a way that it does not occupy much memory or disk space and such that we can analyze it easily. To do so, we use BitMat[4–6].

As illustrated in figure 2, BitMat constructs a 3-dimensional bit cube where each dimension of the cube represents unique subjects (S), predicates (P), and objects (O) occurring in the RDF data. Each unique S, P, and O value is mapped to a unique position (identified by an integer) along each dimension of the bit cube, except that common subject and object resources are mapped to the same

[13] http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/#initDefinitions

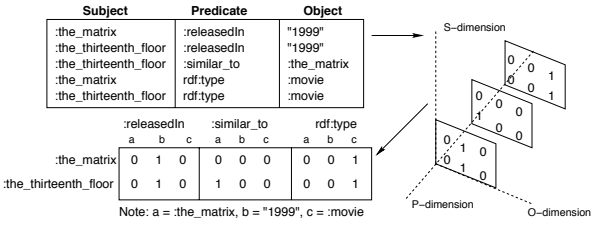

Fig. 2. Construction of a BitMat

integer identifiers. Since BitMat's query processing algorithm does not process predicate-subject or predicate-object joins, overlapping S, P, and O IDs do not pose a problem for query processing. If V_s, V_p, and V_o are the number of unique subjects, predicates, and objects in the RDF data, then the size of this bit cube is $V_s \times V_p \times V_o$. A bit is set to 1 if the corresponding triple exists in the underlying RDF data.

This bit cube can be stored in various ways. One way is to flatten the bit cube by slicing it along the predicate dimension and concatenating all the subject-object matrices along the object dimension, as shown in figure 2. This is the method we use in this work and is outlined in [4]. Additionally, as elaborated in the recent work by Atre et al. in [5], the bit cube can be sliced along differing dimensions, thus effectively creating multiple indexes. These additional indexes improve query performance at the expense of increased size.

Without compression, BitMats are typically very large and very sparse. Therefore, we apply "gap encoding" on each row of the matrix to compress it. For example, a bit-row of "0011000" would be represented as "[0] 2 2 3" (starting with the first bit value, we record alternating run lengths of 0s and 1s). This allows us to store very large amounts of data with a much smaller space requirement.

We define three primitive operations for BitMats:

– *Filter* takes as input a BitMat and a triple pattern and returns a new BitMat which contains only those triples that match the triple pattern. For example, a triple pattern of { `?s ex:p ?o` } would clear all the subject-object matrices not corresponding to the predicate `ex:p`.
– *Fold* takes a BitMat and a triple position (subject, predicate, or object) and returns a bit vector which contains a 1 in position i if there exists at least one triple in the BitMat with ID i in the specified position.
– *Unfold* takes a BitMat and a bit vector corresponding to a triple position (e.g. the result of a *fold* operation) and returns a new BitMat. For each 1 in position i of the bit vector, corresponding triples with resource i in the specified position are retained in the resulting BitMat.

These operations work directly on compressed BitMats using bitwise operations without requiring decompression at any point.

For processing join queries, we use a novel two-phase algorithm. In the first phase, we create a BitMat for each triple pattern in the query, filter each BitMat with its associated triple pattern, and propagate constraints between the BitMats based on join variables. In the second phase, instead of performing pairwise joins as is done in conventional join processing, we stitch the results together using a technique similar to multi-way joins [32]. The entire query processing takes place in memory without accessing the disk. Further details of this algorithm are beyond the scope of this paper but may be found in [4].

5. Evaluation

In this section, we provide the technical and performance details of using our system to extract data from the BTC2009 dataset and perform rudimentary analysis on the extracted dataset. Run times for the individual phases are summarized in Table 1, as reported at the Billion Triples Challenge .

Reasoning	349 sec
Extracting/Reducing	940 sec
BitMat Creation	25 sec
TOTAL	1,314 sec (≈22 min)

Table 1
Time required for individual phases of the reduction process over the Billion Triples Challenge 2009 dataset.

5.1. *Reasoning*

Inferencing was done on the Opteron blade cluster at Rensselaer Polytechnic Institute's Computational Center for Nanotechnology Innovations (CCNI). Each machine is an IBM LS21 blade server running Red Hat Enterprise Linux WS release 4 (Nahant Update 8) with two dual-core 2.6 GHz AMD Opteron processors with gigabit Ethernet and InfiniBand interconnects and system memory

of at least 8GB. We read and write files to/from the large General Parallel File System [14].

As mentioned, according to [40], the substantive RDFS closures of different sets of data (each including all ontological triples) is the same as the substantive RDFS closure for the union of those sets. The ontology is provided in a separate file from the assertional data, so no preprocessing is required to extract the ontology. Note, however, that in the case of the BTC2009 dataset, some processors may have ontological triples from the BTC2009 dataset that are not in the target ontology and that may not be shared with other processors. As a result, each processor produces the partial, substantive RDFS closure of its local dataset with respect to all of its ontological triples, but when the results are merged together, they create the partial, substantive RDFS closure with respect to only the target ontology. In other words, the final result is sound and complete (for the partial, substantive RDFS closure) with respect to the target ontology but only sound with respect to all ontological triples in both the target ontology and the BTC2009 dataset.

Since the BTC2009 dataset is provided as 116 separate files, we originally computed the partial, substantive RDFS closure (with respect to the target ontology) for each of these files separately to produce the closure for the entire dataset, using 32 processors for each file. Although we could have computed the closure of the entire BTC2009 dataset all at once, doing so requires a large number of processors. Due to queuing contention on the cluster, allocating such a large number of processors was infeasible at the time of the Billion Triples Challenge.

Had we run all of these jobs in parallel, it would have used 32×116 = 3,712 processors, each 32-processor job having exclusive access to its respective files. The materialization computation took on average 32.5 seconds per processor. However, high data skew in the BTC2009 dataset resulted in a minimum processor time of 5.3 seconds and a maximum processor time of 348.1 seconds. Interestingly, all processors except for one finished in less than 120 seconds. The closure of the BTC2009 dataset contains 1,620,437,279 (unique) triples. The original BTC2009 dataset contains 898,966,813 (unique) triples, meaning 721,470,466 inferences were materialized. This is the result reported at the time of the Billion Triples Challenge and which is shown in table 1.

However, since the Billion Triples Challenge, we have performed the same reasoning on the same dataset using 1,788 processors in parallel all sharing access to the same files. The detailed performance results are delineated in table 2. The total time for all processors to finish was nearly 30 minutes. As previously mentioned, each processor computes the partial, substantive RDFS closure of its local dataset with respect to all of its ontological triples. Therefore, as we reduce the number of processors, we increase the number of inferences because each partition of the data contains as many or more ontological triples than a smaller partition. Thus, ontological triples in the BTC2009 dataset that are not in the target ontology have more far-reaching effect, and the output of the 1,788-processor run has more triples than the output of the 3,712-processor run. Therefore, we do not report the time for the 1,788-processor run in table 1 because we do not use its output as input in the reduction phase.

PHASE	**Min**	**Avg**	**Med**	**Max**
Initialization	4.5 1	4.56	4.56	4.65
Load Target Ontology	13.21	13.76	13.39	47.68
Load BTC2009 Dataset	26.16	60.94	60.44	60.08
Produce Closure	2.02	33.69	34.65	1,628.50
Write Closure to File	1.87	122.50	131.92	228.69
Finalize	0.15	2.94	2.79	21.05
TOTAL	84.33	237.53	250.23	1,777.41

Table 2
Minimum, average, median, and maximum times (in seconds) for the individual phases of reasoning.

Again, the results show high skew in performance among the processors. While we expect some amount of skew from a random assignment of triples to each processor, the maximum reasoning time is surprisingly far from the average and median. This led us to investigate the cause of such skew. Figure 3 shows the fraction of total inferences produced over time. (Note that duplicate inferences between processors are included for this figure, and by "produced," we mean written to file at the end of each independent process.) At 379 seconds (6.32 minutes), 98.77% of the inferences were produced, and all processors but one had finished. The one remaining processor took 1,778 seconds (29.63 minutes) to finish producing its closure. Looking at the output for the processor, we found an abundance of

[14] http://www-03.ibm.com/systems/clusters/software/gpfs/, accessed 2010/02/01.

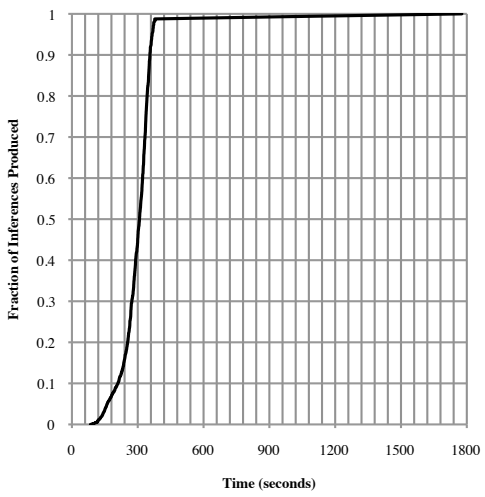

Fig. 3. Fraction of inferences produced over time

triples like { _:b rdf:type owl:Nothing }. Looking at the input for that processor, we discovered the following triples:

```
owl:Thing owl:equivalentClass owl:Nothing .
owl:equivalentClass rdfs:subPropertyOf
                                rdfs:subClassOf .
```

Therefore, owl:Thing was inferred to be a subclass of owl:Nothing.[15] Furthermore, owl:Thing is declared to be a subclass of 91 resources and a superclass of 23 resources which include common classes like rdf:List, rdf:Property, rdfs:Resource, xsd:string, geo:SpatialThing, and skos:Concept. As a result, the processor inferred an unusually high number of rdf:type statements from subclass hierarchy, causing it to lag behind other processors.

5.2. *Reduction*

Given our use case, we are interested in extracting just information about people to produce a reduced dataset. Using the previously discussed parallel query answering system, we can easily extract all people with both full names and email addresses using the following SPARQL basic graph pattern query:

```
SELECT ?p ?e ?n
WHERE {
  ?p a up:Person ;
    up:email ?e ;
    up:full_name ?n .
}
```

Although we use the SPARQL SELECT syntax to describe the basic graph pattern in this example, we actually write the results to disk as RDF, emulating a SPARQL CONSTRUCT query for the triples that are matched by the basic graph pattern.

We ran this query on the CCNI's IBM Blue Gene/L supercomputer using 8,192 nodes, each of which has two 700-MHz PowerPC 440 processors and 1GB of system memory. We utilize three of the Blue Gene/L's specialized hardware networks: a 175MBps 3-dimensional torus for point-to-point communication, a 350MBps global-collective network, and a global barrier network. Loading all 1.6 billion triples took 62 seconds (representing a rate of almost 26 million triples per second) while executing the query and writing the results to disk took 878 seconds for a total running time of roughly 16 minutes. We note that query execution time depends heavily on the selectivity of the triple patterns that comprise the query.

5.3. *Analysis*

After extracting data from the full dataset, the reduced dataset consists of 784,783 triples, a much more manageable size for use in further analysis without requiring a large cluster or supercomputer. Stored in the N-Triples serialization, this data occupies 113MB on disk. Using the BitMat system, the same data can be stored in an 8MB compressed index along with a 25MB uncompressed dictionary file mapping node values to integer IDs. We performed the conversion of triples to BitMat and all queries on the BitMat on a Lenovo ThinkPad T60 laptop with a 2GHz Intel T2500 Core Duo Processor having 1GB of RAM and running Ubuntu 9.04. Constructing the BitMat took roughly 25 seconds.

We chose a number of queries to run against the reduced dataset. Suppose we want to look up a person's email address by name. Using Libby Miller as an example, we formulate the following query:

```
SELECT ?p ?email
WHERE {
  ?p up:full_name "Libby Miller" ;
    up:email ?email .
}
```

This query returns 96 unique results with 3 unique email addresses in 0.101 seconds.

[15] As mentioned in a previous footnote, despite not being complete with regards to triples that "extend" RDFS terms, some such inferences may still be produced. This is one of those cases.

Next, suppose we want to look up URIs for persons by email address. Considering up:email to be an inverse functional property, such a lookup indicates that some URIs and/or blank nodes represent the same person. Again, using Libby Miller as an example, we formulate the following query:

```
SELECT ?uri
WHERE {
 ?uri up:email
  <mailto:libby.miller@bristol.ac.uk> .
}
```

This query returns 205 unique results in 0.030 seconds.

Finally, we generate statistics in order to analyze the frequency of common names. Figure 4 shows the 25 most commonly occurring names in the reduced dataset.

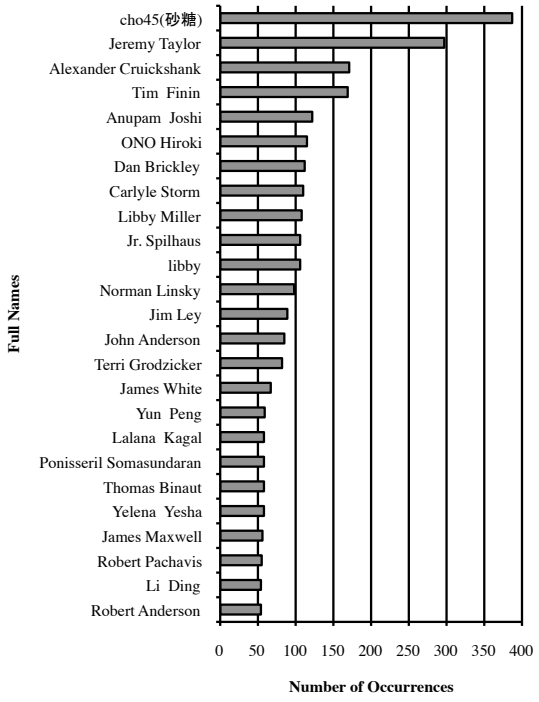

Fig. 4. Most Frequent Full Names in Reduced Dataset

6. Conclusion

In this paper we have shown that the strength of high performance clusters can be leveraged to provide rapid RDF inferencing and data extraction from large datasets. Using a combination of Opteron blade cluster and Blue Gene/L, we are able to load the Billion Triples Challenge dataset, fully materialize inferences, extract a specific subset, compress the subset, and perform rudimentary analysis all in the order of tens of minutes. To our knowledge no other system is able to provide similar performance.

We have shown that use of a compressed triplestore such as BitMat allows very efficient RDF storage while preserving the ability to directly query the data. This makes it well suited for use with large datasets on personal computers and other resource-constrained devices.

In the future, we hope to extend our system in several ways. First, we intend to integrate inferencing and query execution into a single process. This would allow query evaluation without requiring the preprocessing step of inference materialization, and reduce the overhead cost of writing the materialized data to disk. Such integration would also provide basic graph pattern query functionality to the inferencing engine allowing more expressive forms of reasoning that cannot be formulated as APS rules (e.g. many OWL2RL rules). We also intend to extend the supported query expressivity from basic graph patterns to support operations such as optional graph patterns and filtering, providing the user with greater latitude in choosing what data subset to extract. Finally, we hope to continue developing BitMat, increasing scalability to allow analysis of much larger data subsets.

7. Acknowledgement

Funding for this work was provided in part by a seedling grant from DARPA's Transformational Convergence Technology Office and gifts from Lockheed Martin Advanced Technology Laboratories and Fujitsu Laboratories of America. We thank Jacqueline A. Stampalia, Lindsay Todd, Luz Angelica Nunez, and Michael D. Sofka for providing support in performing our evaluation at Rensselaer Polytechnic Institute's Computational Center for Nanotechnology Innovations.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable Semantic Web Data Management using Vertical Partitioning, in: PVLDB, 2007.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, A. Silberschatz, Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for

analytical workloads, in: VLDB'09: Proceedings of the 2009 VLDB Endowment, 2009.

[3] G. Anadiotis, S. Kotoulas, E. Oren, R. Siebes, F. van Harmelen,
N. Drost, R. Kemp, J. Maassen, F. J. Seinstra, H. E. Bal, MaRVIN: a distributed platform for massive RDF inference, http://www.larkc.eu/marvin/btc2008.pdf, accessed 2010/02/01 (2008).

[4] M. Atre, V. Chaoji, J. Weaver, G. T. Williams, BitMat: An In-core RDF Graph Store for Join Query Processing, http://www.cs.rpi.edu/~atrem/papers/bitmat_techrep.pdf (August 2009).

[5] M. Atre, V. Chaoji, M. J. Zaki, J. A. Hendler, Matrix "Bit"loaded: A scalable lightweight join query processor for RDF data, in: Proceedings of the 19th International World Wide Web Conference, 2010.

[6] M. Atre, J. Hendler, BitMat: A Main Memory Bitmatrix of RDF Triples, in: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems, 2009.

[7] P. A. Bernstein, N. Goodman, Power of natural semijoins, SIAM Journal of Computing 10 (4).

[8] J. Bock, Parallel Computation Techniques for Ontology Reasoning, in: The Semantic Web - ISWC 2008, vol. 5318/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008.

[9] M. Cai, M. R. Frank, RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network, in: Proceedings of the 13th International World Wide Web Conference, 2004.

[10] P. Cudré-Mauroux, S. Agarwal, K. Aberer, GridVine: An Infrastructure for Peer Information Management, IEEE Internet Computing 11 (5) (2007) 36–44.

[11] D. J. DeWitt, R. H. Gerber, Multiprocessor Hash-Based Join Algorithms, in: Proceedings of the 11th International Conference on Very Large Data Bases, 1985.

[12] O. Erling, Toward web scale RDF, in: Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge Base Systems, 2008.

[13] O. Erling, I. Mikhailov, RDF Support in the Virtuoso DBMS, in: S. Auer, C. Bizer, C. Müller, A. V. Zhdanova (eds.), Proceedings of the 1st Conference on Social Semantic Web, vol. 113 of LNI, GI, 2007.

[14] Q. Fang, Y. Zhao, G. Yang, W. Zheng, Scalable Distributed Ontology Reasoning Using DHT-Based Partitioning, in: The Semantic Web, vol. 5367/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008.

[15] Y. Guo, J. Heflin, A Scalable Approach for Partitioning OWL Knowledge Bases, in: Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems, 2006.

[16] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, Web Semantics: Science, Services and Agents on the World Wide Web 3 (2-3) (2005) 158–182.

[17] S. Harris, N. Lamb, N. Shadbolt, 4store: The Design and Implementation of a Clustered RDF Store, in: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems, 2009.

[18] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, in: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference, 2007.

[19] P. Hayes, RDF Semantics, http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ (2004).

[20] A. Hogan, A. Harth, A. Polleres, SAOR: Authoritative Reasoning for the Web, in: ASWC, 2008.

[21] A. Hogan, A. Harth, A. Polleres, Scalable Authoritative OWL Reasoning on a Billion Triples, in: Proceedings of Billion Triple Semantic Web Challenge, 2008.

[22] A. Hogan, A. Harth, A. Polleres, Scalable Authoritative OWL Reasoning for the Web, International Journal on Semantic Web and Information Systems.

[23] Z. Kaoudi, I. Miliaraki, M. Koubarakis, RDFS Reasoning and Query Answering on Top of DHTs, in: The Semantic Web - ISWC 2008, vol. 5318/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008.

[24] E. Liarou, S. Idreos, M. Koubarakis, Continuous RDF Query Processing over DHTs, in: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference, 2007.

[25] T. Liebig, F. Muller, Parallelizing Tableaux-Based Description Logic Reasoning, in: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, vol. 4806/2007 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007.

[26] T. Neumann, G. Weikum, RDF3X: a RISC style Engine for RDF, in: PVLDB, 2008.

[27] T. Neumann, G. Weikum, Scalable join processing on very large RDF graphs, in: SIGMOD, 2009.

[28] E. Oren, C. Gueret, S. Schlobach, Anytime Query Answering in RDF through Evolutionary Algorithms, in: The Semantic Web - ISWC 2008, vol. 5318/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008.

[29] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, F. van Harmelen, MaRVIN: A platform for large-scale analysis of Semantic Web data, in: Proceeding of the WebSci'09: Society On-Line, 2009.

[30] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, F. van Harmelen, Marvin: Distributed reasoning over large-scale semantic web data, Journal of Web Semantics 7 (4) (2009) 305–316.

[31] A. Owens, A. Seaborne, N. Gibbins, mc schraefel, Clustered TDB: A Clustered Triple Store for Jena, http://eprints.ecs.soton.ac.uk/16974/1/www2009fixedref.pdf, accessed 2010/02/01 (2008).

[32] D. Papadias, N. Mamoulis, Y. Theodoridis, Constraint-based processing of multiway spatial joins, Algorithmica 30 (2) (2001) 188–215.

[33] S. Rudolph, T. Tserendorj, P. Hitzler, What is Approximate Reasoning?, in: Web Reasoning and Rule Systems, vol. 5341/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008.

[34] L. Sidirourgos, R. Goncalves, M. Kersten, et al., ColumnStore Support for RDF Data Management: not all swans are white, in: PVLDB, 2008.

[35] R. Soma, V. K. Prasanna, A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases, Tech. rep., University of Southern California (2008).

[36] R. Soma, V. K. Prasanna, Parallel Inferencing for OWL Knowledge Bases, in: ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing, IEEE Computer Society, Washington DC, USA, 2008.
[37] H. J. ter Horst, Combining RDF and part of OWL with rules: Semantics, decidability, complexity, in: International Semantic Web Conference, 2005.
[38] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H. Bal, OWL reasoning with WebPIE: calculating the closure of 100 billion triples, in: Proceedings of the 7th Extended Semantic Web Conference, 2010.
[39] J. Urbani, S. Kotoulas, E. Oren, F. van Harmelen, Scalable Distributed Reasoning Using MapReduce, in: Proceedings of the 8th International Semantic Web Conference, 2009.
[40] J. Weaver, J. A. Hendler, Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples, in: Proceedings of the 8th International Semantic Web Conference, 2009.
[41] J. Weaver, G. T. Williams, Scalable RDF query processing on clusters and supercomputers, in: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems, 2009.
[42] C. Weiss, P. Karras, A. Bernstein, Hexastore: Sextuple Indexing for Semantic Web Data Management, in: PVLDB, 2008.
[43] B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, chap. 3, 2nd ed., Prentice Hall, 2005.

Appendix

The upper ontology used during inferencing over the BTC2009 dataset is shown below in the Turtle [16] serialization.

```
@prefix rdf:
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:
 <http://www.w3.org/2000/01/rdf-schema#> .
@prefix akt:
 <http://www.aktors.org/ontology/portal#> .
@prefix dbpo:
 <http://dbpedia.org/ontology/> .
@prefix dbpp:
 <http://dbpedia.org/property/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix sioc: <http://rdfs.org/sioc/ns#> .
@prefix up:
 <http://www.cs.rpi.edu/~weavej3/btc2009#> .

up:Person a rdfs:Class .
up:email a rdf:Property .
up:email_sha1 a rdf:Property .
up:first_name a rdf:Property .
up:last_name a rdf:Property .
up:full_name a rdf:Property .
up:affiliated_with a rdf:Property .
up:web_page a rdf:Property .

sioc:User
 rdfs:subClassOf up:Person .
sioc:email
 rdfs:subPropertyOf up:email .
sioc:email_sha1
 rdfs:subPropertyOf up:email_sha1 .
sioc:first_name
 rdfs:subPropertyOf up:first_name .
sioc:last_name
 rdfs:subPropertyOf up:last_name .
sioc:name
 rdfs:subPropertyOf up:full_name .
dbpo:Person
 rdfs:subClassOf up:Person .
dbpp:name
 rdfs:subPropertyOf up:full_name .
akt:Person
 rdfs:subClassOf up:Person .
akt:has-email-address
 rdfs:subPropertyOf up:email .
akt:family-name
 rdfs:subPropertyOf up:last_name .
akt:full-name
 rdfs:subPropertyOf up:full_name .
akt:given-name
 rdfs:subPropertyOf up:first_name .
foaf:Person
 rdfs:subClassOf up:Person .
foaf:family_name
 rdfs:subPropertyOf up:last_name .
foaf:firstName
 rdfs:subPropertyOf up:first_name .
foaf:knows
 rdfs:subPropertyOf up:affiliated_with .
foaf:surname
 rdfs:subPropertyOf up:last_name .
foaf:mbox
 rdfs:subPropertyOf up:email .
foaf:mbox_sha1sum
 rdfs:subPropertyOf up:email_sha1 .
foaf:page
 rdfs:subPropertyOf up:web_page .
foaf:name
 rdfs:subPropertyOf up:full_name .
foaf:givenname
 rdfs:subPropertyOf up:first_name .
```

[16] http://www.w3.org/TeamSubmission/turtle/, accessed 2010/02/01.