Undefined 0 (0) 1
IOS Press

A Scalable RDF Data Processing Framework based on Pig and Hadoop

Yusuke Tanimura [a,*], Steven Lynden [a], Akiyoshi Matono [a] and Isao Kojima [a]
[a] *National Institute of Advanced Science and Technology (AIST)*
1-1-1 Umezono, Tsukuba Central 2, Tsukuba, Ibaraki 305-8568, Japan
E-mail: {yusuke.tanimura, steven.lynden, a.matono, isao.kojima}@aist.go.jp

Abstract. In order to effectively handle the growing amount of available RDF data, scalable and flexible RDF data processing frameworks are needed. While emerging technologies for Big Data, such as Hadoop-based systems that take advantages of scalable and fault-tolerant distributed processing, based on Google's distributed file system and MapReduce parallel model, have become available, there are still many issues when applying the technologies to RDF data processing. In this paper, we propose our RDF data processing framework using Pig and Hadoop with several extensions to solve the issues. We integrate an efficient RDF storage schema into our framework and then show the performance improvement from Pig's standard bulk load and store operations, including the schema conversion cost from conventional RDF file formats. We also compare the performance of our framework to the existing single-node RDF databases. Furthermore, as reasoning is an important requirement for most RDF data processing systems, we introduce the user operation for inferring new triples using entailment rules and show the performance evaluation of the transitive closure operation as an example of the inference, on our framework.

Keywords: RDF, Parallel and distributed processing, Storage schema, Pig, Hadoop

1. Introduction

Metadata describing a variety of contents and contexts is recognized as being an important aspect of knowledge based processing. RDF (Resource Description Framework), a W3C standard for describing metadata [39], is becoming widely adopted in the Semantic Web world, where it is increasingly being used to publish large amounts of information. The amount of the available RDF data is rapidly increasing, and it will be necessary to support RDF databases in the petabyte scale with trillions of RDF triples in the future. However, the scalability of RDF databases is an unresolved issue. The data size requires distribution over multiple nodes, and graph pattern matching for RDF queries usually requires a large amount of processing power. Using a relational database in the back end of the RDF database might not perfectly solve the problem, because RDF databases require RDF-specific query processing operations and rule-based inference support. Moreover, storing RDF data as metadata and the contents written in the RDF together and analyzing them on the same data processing infrastructure will be one of the practical RDF applications.

In order to solve the scalability problem, we proposed a framework that exploits parallel database processing over the distributed file system and the MapReduce model [35]. The framework design was inspired by the recent achievements of Google and Yahoo for handling petabyte scale Web data on commodity hardware clusters. The distributed file system proposed as Google File System (GFS) [18] provides a functionality to store a large file over multiple storage nodes by dividing it to fixed-size chunks, with fault-tolerance to node crashes achieved using the chunk replica on other nodes. MapReduce [16] is a programming model to compose a parallel job by defining sub-tasks as an arbitrary map operation processing the chunk and a reduce operation merging the outputs of the maps, and also an efficient and fault-tolerant execution model allowing

*Corresponding author. E-mail: yusuke.tanimura@aist.go.jp.

retries of the sub-tasks on a distributed environment running the GFS. Hadoop [3] is based on GFS and MapReduce, and it is an open-source software aimed at providing a similar functionality. We use Hadoop as the basic infrastructure and also use Pig [28], which provides a general data processing platform on top of MapReduce and allows users to write a script incorporating database operations (e.g. filter, join) in a procedural programming style. Pig compiles the script and generates the MapReduce code to run on the Hadoop installed system. The scalability of this software stack is shown by other works [17,29]. However, because the RDF data model is a labeled, directed graph and complicated analytical queries are applied, more efficient and fine-tuned methods for RDF data processing should be integrated into the software stack.

In our previous work [35], we studied a storage schema to shorten the execution time of the select-projection-join (SPJ) query, which was implemented using MapReduce and executed on the Hadoop system, taking into account characteristics of the RDF data. Then we proposed an initial RDF data processing framework in the paper [36]. In this paper, the complete architecture design and extended methods for RDF data processing are presented with comprehensive evaluation, as follows:

- Definition of the RDF storage schema combined with vertical partitioning and the key-value data format of the Hadoop.
- Development of the schema conversion tool using MapReduce and the RDF data loader of Pig, for the above storage schema.
- Basic performance evaluation of our proposed framework using SP^{2}Bench [32].
- Definition of the inference command in Pig Latin.
- Implementation of the transitive closure operation on Pig, and its performance improvement by dynamic parameter tuning and parallel execution.

In the rest of the paper, we describe about the scalability problem of the RDF data in Section 2, a design overview and advantages of our proposed framework and extensions for efficient RDF data processing in Section 3, and then present implementation of each extension and evaluation result in Section 4. The related works are summarized in Section 5 and the paper is concluded in Section 6.

2. Scalability Requirements of RDF/RDF-annotated Data Repositories

A scalability problem involving storing and querying large RDF data is rapidly being recognized to be important. First, the number of the RDF repositories continues to increase as RDF becomes more widely adopted within applications. We can see the amount of the public data and their interlinks in the report from the W3C SWEO (Semantic Web Education and Outreach) Linking Open Data community project [7]. In September, 2011, 295 datasets consist of over 31 billion RDF triples, which are interlinked by around 504 million RDF links. There is another report that 973 million pages contained Microformat, Microdata, RDFa data in 3.2 billion HTML pages [21], and there is an effort to map Microdata to RDF [22]. These facts indicate that many data on the Web are being structured, or can be represented as RDF, and they would become an important application for search engines or data warehouses.

Second, the size of each RDF repository is also growing. One of the most prominent repositories, the DBpedia [2] knowledge base, describes more than 2.6 million things with 274 millions' RDF triples whose size is about 67GB. Other sources of large amounts of RDF data also exist, for example "ucode" is an identifier used to assign potentially billions of real-world objects so that the object becomes uniquely identifiable and manageable by computers [8]. Ucode is written in RDF, which means a large amount of RDF data might be produced as ucode becomes widely adopted.

Finally, in addition to the repository having only RDF data, there is an RDF-annotated data repository whose size can be much larger. For example, scientific data related to geosciences or physics now needs to be handled on the petabyte scale. Data analysis with a set of queries using metadata about scientific data is common and RDF is used for the metadata in those applications, too. In this scenario, providing a unified data processing infrastructure for both RDF and annotated data with efficient access on such a large scale is an important challenge.

Our proposed framework, explained in the next section, aims to support both types of repositories, the RDF-only type and the other type having both application data and RDF. Both data can be stored on the same distributed file system and both analyses can be performed on the same MapReduce execution system, but this paper focuses on the techniques for only handling RDF data.

3. Proposed RDF Data Processing Framework

3.1. Design concept

Our proposed RDF data processing framework is built on top of the distributed file system (DFS) and MapReduce provided by Hadoop, and based on Pig with several extensions to specially support RDF data processing. The reasons why we chose this layered architecture are as follows:

- **Scalable architecture**: Scalability of the DFS and the MapReduce was shown by Google's, Yahoo's and now many other activities, in storing and analyzing petabyte scale Web data over thousands of nodes.
- **A common set of data processing tools and a general query optimization framework**: Pig on top of the Hadoop provides a general data processing platform and supports a common set of database operations: `SELECT`, `FOREACH`, `FILTER`, `JOIN`, and etc. The operations are also useful for RDF data processing, which means that RDF and other generic data processing can be executed on the same infrastructure. This benefits users to perform efficient integrated data processing using RDF annotation.
- **Flexible interface for applying custom processing**: The MapReduce programming model supports the user defined function (UDF) in the map and reduce tasks. This capability is kept in Pig and the customized operation can be programmed in Pig Latin, a programming language of Pig. The UDF is extremely useful for rule-based processing and data analysis using third-party tools.

While the architecture has these advantages, Pig does not support various optimization techniques that are traditionally provided by database management systems. Pig implements the optimization techniques that do not require the data schema [27,17]. In addition, RDF queries usually require multiple joins, but Pig assumes that performing more than one join is rare and does not optimize the join ordering. Without optimization well-suited to applications, the query execution time would be very long. Therefore we need to define the storage schema for the RDF data and more effectively optimize the query. We keep the original system design of Pig and do not force non-RDF data to have the schema, so that the flexibility for the other data will not be lost. Instead the other data's schema can be described with the RDF data.

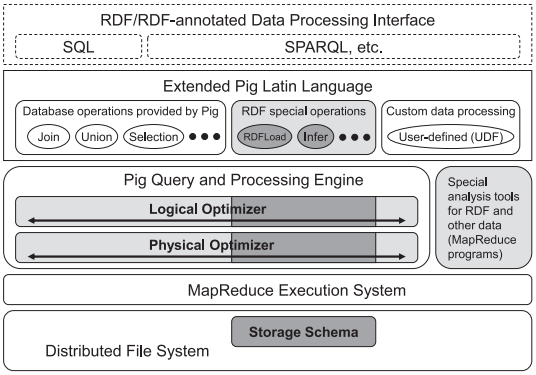


Fig. 1. An architecture overview of our proposed RDF data processing framework

For the rule-based processing with RDF, users would give rules as inputs to Pig, and then Pig needs to generate corresponding MapReduce operations to apply rules to the RDF data. Additional operations and/or built-in functions of Pig Latin should be designed for users to do this with ease.

3.2. Architecture overview

Figure 1 shows an architecture overview of our proposed RDF data processing framework. The extended Pig Latin language provides three types of the processing interface; a general data processing interface provided as Pig Latin standard commands, an RDF data processing interface that we extend, and a custom data processing interface given by users as UDF. Our extensions, which are shown in grey in Figure 1, are integrated into each layer, and they are to make RDF data processing more efficiently on this framework. The storage schema is defined at the bottom layer and the optimization using the schema is implemented in Pig's query engine. RDF-specific `LOAD`, `INFER` and other operations would be implemented in the Pig processing engine, or developed as special analysis tools using MapReduce. A SPARQL interface can be implemented on top of the extended Pig Latin by leveraging the three interfaces, if necessary, but it is out of the scope of this paper. The details of our extensions are described in the next subsection.

3.3. Extensions for RDF data processing

3.3.1. Storage schema

We design the RDF storage schema under the consideration of the following three aspects;

```
a = LOAD 'DBpedia' USING RDFLoader(
        '?predicate = <http://dbpedia.org/property/relatedInstance>');
b = LOAD 'WordNet' USING RDFLoader(
        '?subject = <http://www.w3.org/2006/03/wn/wn20/schema/tagCount>');
```

Fig. 3. Example of the LOAD operation using RDFLoader()

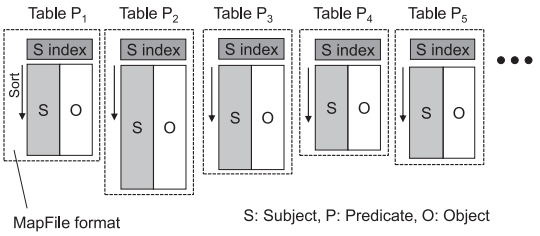


Fig. 2. Vertical partitioning combined with MapFile (VP-MapFile)

- An internal file structure of both RDF triples and intermediate results of queries and analysis.
- Data partitioning, which relates to a file tree structure and file distribution on the DFS.
- Structure of indices including inferred data.

and have implemented the vertical partitioning approach, which is one of the three data partitioning methods studied in our previous work [35].

The vertical partitioning [10] approach (VP) uses the characteristic that predicates rarely become a variable in most RDF queries. Because, in our previous experiments, VP outperforms other methods in terms of performance and less disk space usage, we adopted VP for integrating into our proposed framework. In VP, all triples are divided into predicate tables which correspond to files on the DFS, and each file name relates to the predicate value. In Section 4.1, we implement VP combined with the MapFile of the Hadoop. This implementation, which is called VP-MapFile, uses the MapFile as the internal file format and leverages lookup capability of key-value pairs, by using index and sort, as shown in Figure 2. Thus, when the query specifies the predicates and/or subjects, the matching triples can be quickly loaded.

In order to load the RDF data into the Pig runtime, we developed RDFLoader() that is a built-in load function of Pig. The example use of the RDFLoader() is shown in Figure 3. The first argument of `LOAD`, 'DBpedia,' is a repository name of the data to be loaded. The argument of the RDFLoader() accepts expressions based on the syntax of the SPARQL filter clause and applies the filter when loading data so that unnecessary data will never be read from the DFS. In the 'DBpedia' example, the RDF data whose predicate is 'relatedInstance' will be loaded. Once the data is loaded, the standard Pig operations can be applied to the data.

In addition, we developed a helper tool that converts the RDF data on existing RDF file formats to our VP-based storage schema. The tool can be applied to files imported into the DFS and the conversion is performed by the MapReduce execution.

3.3.2. Query optimization

The RDF query execution is optimized for our storage schema described in the previous subsection. The filter operations would be applied when loading data, which allows the system to read only the required RDF data from the DFS and achieves significant performance improvement. We evaluate this improvement, and the performance of the query execution by using SP^2Bench. The details are described in Section 4.

In addition to this, any other extended optimizations for RDF data processing can be applied to the original Pig query engine [17] in Figure 1. The logical optimizer interprets the statements in Pig Latin with/without RDF-specific operations, and constructs an optimized logical plan. The logical optimizer may change the order of the operations, combine more than two operations into one equivalent operation or make a decision to use materialized data (e.g. join indices) if it improves the performance. Then the physical optimizer compiles the logical plan into several MapReduce jobs.

In particular for RDF data processing, a sequence of self-join operations to find new triples of the RDF data graph is required by rule-based processing. Then multiple join operations are expected to be optimized by reordering, reuse of the past results, choosing faster join algorithms against the specific data distributions, and so on. As one of such techniques, we have proposed an adaptive multi-join method by a single MapReduce execution, which is presented in the paper [24], though the method has not been integrated into our proposed framework yet.

3.3.3. Reasoning support

Our proposed framework provides reasoning capability in two aspects, using given rules to infer new information and using the rules for performing faster query execution. In this paper, we focus on the former

Table 1
Specification of the cluster

Cluster specification	9 nodes – 1 node for client and management, and 8 nodes for processing and storage – 111GB disk space is allocated for Hadoop on each node. – CentOS 4.5, Hadoop release 0.20.2, Pig release 0.7.0
Hardware of each node	Intel Xeon 2.8GHz×2, 1GB memory, 3-drive RAID-5 disk array, 1 Gigabit Ethernet

Table 2
Specification of the virtualized cluster

Cluster specification	13 nodes, each of which runs on a different host machine – 1 node for client and management, and 12 nodes for processing and storage – 8 cores, 16GB memory, and 300GB disk space for Hadoop are allocated on each node. – CentOS 5 with KVM, Hadoop release 0.20.2, Pig release 0.7.0
Hardware of each host machine	Intel Xeon E5220 2.27GHz Quad Core×2, 24GB memory, 2TB disk, 1 Gigabit Ethernet

```
b = RDF-INFER a USING RDFRuleBase('myrule-1');
```

Fig. 4. Example of the RDF-INFER operation using RDFRuleBase()

one and propose the `RDF-INFER` command, which can be used to infer new RDF triples based on a set of rules, as an addition to Pig Latin. In Figure 4, the `RDF-INFER` finds new triples that satisfy the rule 'myrule-1.' The RDFRuleBase() is an example, new function for the `RDF-INFER` and it is implemented as a built-in function of Pig to interpret the rules for 'myrule-1' and generate a sequence of the Pig operations such as select, projection, join, etc. to apply the rules. The supported rules and the rule syntax for 'myrule-1' depends on each built-in function. Providing the `RDF-INFER` command is not only aimed at providing a high level interface to users but also aims to add the necessary functionality to Pig in order to execute the inference operation efficiently. Use of specific algorithms, reordering or merging operations specific to the RDF inference, and use of the materialization can be managed by the Pig query engine.

4. Implementation and evaluation

This section describes about implementation and evaluation of our extensions. The analytical query performance of our proposed framework was evaluated by two aspects, performance comparison against existing RDF databases by using the RDF benchmark suite and performance evaluation of our own implementing the transitive closure operation. Through the evaluations, scalability of our proposed framework and possibility of further performance improvement are presented.

Evaluations were performed on two environments each of whose specification is shown in Table 1 and 2 respectively. While Table 1 is a normal Hadoop-installed cluster, Table 2 is a cluster which consists of virtualized servers. Because each virtualized server occupies a single physical server, we only need to consider virtualization overheads on each node, in particular to I/O performance degradation compared to the physical server, when looking at results.

4.1. Schema-aware RDF data loader

4.1.1. Implementation

We implemented a prototype of the schema-aware RDF data loader, which is described as RDFLoader() here. The filter clause of the RDFLoader() understands simple string matching of the subject, the predicate or the object and takes advantages of the VP and MapFile schemas. Future work will examine the possibility of adding support for more advanced filtering operations on various RDF data types. Currently the RDFLoader() supports both text and binary formats of the predicate file. The binary format combined with VP was implemented with the MapFile of the Hadoop, which is a sorted data format with an index to permit lookups by key using a binary search. In this implementation, which is called VP-MapFile, random and partial access by subjects becomes possible by storing subjects as key. In addition, splittable block compression using the LZO [5] library natively installed on the server is available.

Table 3
Stored size of the WordNet 2.0 Full data in the HDFS

	Size [MB]	Ratio [%]
The original N-Triples format	318.2	100.00
N-Triples based format - Tab-delimited text file (*nt*)	317.9	99.89
VP format - Text without compression (*vp-txt*)	219.5	68.97
VP format - MapFile without compression (*vp-mf*)	236.8	74.41
VP format - MapFile with LZO compression (*vp-mfc*)	36.9	11.59

Table 4
Execution time of loading the RDF data with specifying a subject and a predicate

	Execution time [sec]
N-Triples based format - Tab-delimited text file (*nt-1m*)	59.6
N-Triples based format - Tab-delimited text file (*nt-5m*)	23.8
VP format - Text without compression (*vp-txt*)	26.4
VP format - MapFile without compression (*vp-mf*)	14.3
VP format - MapFile with LZO compression (*vp-mfc*)	12.4

4.1.2. Evaluation

Table 3 shows stored data size of WordNet 2.0 Full [38] consisting of 1,942,887 triples, on the Hadoop distributed file system (HDFS) by using our storage schemas, and the ratio to the original size. The N-Triples based format (*nt*) is almost same as the original format, but removes '.' and converts the space delimiter among the subject, the predicate and the object to the tab. In this result, the VP-based formats saved the disk space because it did not contain predicates in each file. Instead, a predicate map between the file names of the predicate group and actual predicates was separately stored and its file size was 3.5 [KB]. The MapFile used more space than the text format (*vp-txt*) but when the compression was enabled in *vp-mfc*, space usage was significantly reduced.

Figure 5 compared the performance results of the RDFLoader() and the standard Pig load function, PigStorage(). In this experiment, we measured the execution time, starting when the WordNet data in Table 3 is loaded, specifying a predicate, until it is completely stored in another file on the HDFS using the default `STORE` operation. In the case of using PigStorage(), the `FILTER` operation was applied to the loaded data before the `STORE` operation, in order to exclude most triples containing unspecified predicates.

The experiment was performed with 5 different predicates so that each output became a different data size. All executions with RDFLoader() ran as a single map task on 1 machine shown as in Table 1. However, *nt-5m* of the execution with PigStorage() ran as 5 map tasks on 5 machines in parallel because the *nt* file consists of 5 blocks due to the default block size parameter (64MB) of the HDFS. In order to eliminate the effects of access locality, we created replicas of the inputs on each machine. Furthermore, we prepared a single block *nt* file and performed the sequential execution with PigStorage() as a single map task (*nt-1m*). Figure 5 shows that loading the data using RDFLoader() was faster than using PigStorage() even in most cases of *nt-5m*. Although MapFile was slow for the data load, MapFile with the LZO compression (*mf-mfc*) was competitive to the text based VP format (*vp-txt*). In this environment, reducing an amount of read data from the HDFS overcame negative impact of the decompression cost.

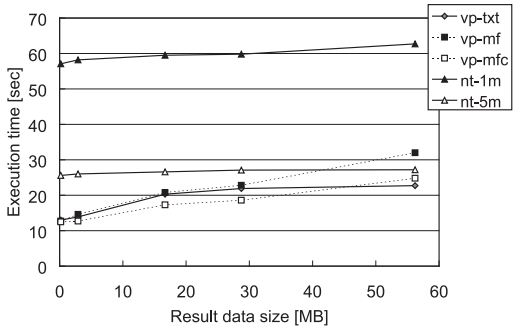


Fig. 5. Execution time of loading the RDF data while specifying a predicate

Table 5
Performance of the schema conversion tool

# of triples	Size in N-Triples [GB]	Exec. time [sec]
100 million	16	4,713
1 billion	158	51,002

Table 4 shows an extended experiment result where the output data was 56.2 [MB] in Figure 5. Here, we specified the subject in addition to the predicate in the arguments of the RDFLoader(). Thus the final output was 121 [bytes]. While *vp-txt* needs to execute the `FILTER` operation after loading the data, MapFile takes an advantage of random access function. Therefore the MapFile based format finished the execution in shorter time as expected.

In these experiments, the advantages of the RDFLoader() was shown, and particularly, *vp-mfc* was superior to others in disk usage and the performance of loading data.

4.2. Schema conversion tool

4.2.1. Implementation

The schema conversion tool, which transforms the RDF data into the data stored with our VP-based schemas described in Section 4.1, was implemented as a MapReduce program. Each map task reads one block of the input RDF file stored in the HDFS, parses triples and outputs subject-object pairs to each predicate group file. Then each reduce task collects the specific predicate files, merges them with sort by the subject and outputs them into the HDFS. At the end of the MapReduce job, the predicate map file is generated. The advantage of this implementation is that high workloads of parsing and sorting are executed in parallel. For parsing, the tool supports our own developed parser for the N-Triples format and the parser of the Sesame 2.3.2 [30]. By using Sesame, the tool can read any RDF formats that Sesame supports, such as N-Triples, Notation3 and so on.

4.2.2. Evaluation

First, two parsers were compared by the conversion performance. The N-Triples formatted file of the WordNet 2.0 Full was converted to *vp-mfc*. Each execution launched 5 maps and 7 reducers on the cluster in Table 1. While the conversion with our own parser took 83 [sec], the one with Sesame took 98 [sec]. The reason is that Sesame's parser starts parsing after reading a whole block, which stalls streamlined processing of the map task suitable to line-oriented inputs.

Then the execution time of the schema conversion was measured with larger datasets which were generated by the *sp2b_gen* program of the SP^{2}Bench [32]. Table 5 shows the experiment result which was performed on the cluster in Table 2. Each dataset on the HDFS was converted from the N-Triples to *vp-mf* with 128 reducers. For data increase, abrupt increase of the execution time was not seen from the result.

Finally, 25 million of triples (4GB in the N-Triples format) generated by *sp2b_gen* was imported to the cluster in Table 2, for the RDF query benchmark using SP^{2}Bench which is described in Section 4.3. The data import into the HDFS took 217 [sec] and the data conversion to *vp-mfc* with the Sesame parser took 524 [sec] with 128 reducers.

4.3. Query performance evaluation

4.3.1. Test method

Performance of our proposed framework was evaluated by SP^{2}Bench, which is a comprehensive RDF data processing benchmark along with the DBLP scenario [23]. Because the original test queries of SP^{2}Bench were written in SPARQL, we translated them to Pig Latin. In addition, we applied the RDFLoader(), which is described in Section 4.1, to the test queries.

The benchmark test was executed on the cluster in Table 2, and *vp-mfc*/*vp-mf* formatted dataset that has been described in Section 4.2.2 was used. Each dataset consists of 25 millions of triples and this is the largest dataset whose experiment result is shown in the SP^{2}Bench paper [32].

4.3.2. Test result

In Table 6, we compared the results obtained using our proposed framework with the results shown in Table IV of the SP^{2}Bench paper [32]. The experiment of the SP^{2}Bench paper was conducted under a single machine that ran Linux ubuntu v7.10 gutsy, on top of an Intel Core2 Duo E6400 2.13GHz CPU, 3GB memory, and 250GB Hitachi P7K500 SATA-II hard drive with 8MB cache, which is different from our environment. However, the comparison indicates that our proposed framework could process the amount of data that could not be handled on single-node RDF databases, due to the processing cost, the lack of the memory and so on.

Table 6
Comparison between our proposed framework and single-node RDF databases by the SP^{2}Bench result

	Query 5a	Query 5b	Query 6
Proposed framework (Used *mfc* format)	301 sec.	240 sec.	350 sec.
Proposed framework (Used *mf* format)	447 sec.	368 sec.	842 sec.
ARQ v2.2 / Jena 2.5.5	Timeout (30 min.)	Timeout (30 min.)	Timeout (30 min.)
Sesame v2.2 beta2 (In-memory)	Memory Exhaustion	Timeout (30 min.)	Memory Exhaustion
Sesame v2.2 beta2 (Mulgara SAIL v1.3 beta1)	Timeout (30 min.)	Success within 30 min.	Timeout (30 min.)
Virtuoso v5.0.6	Loading Failure	Loading Failure	Loading Failure

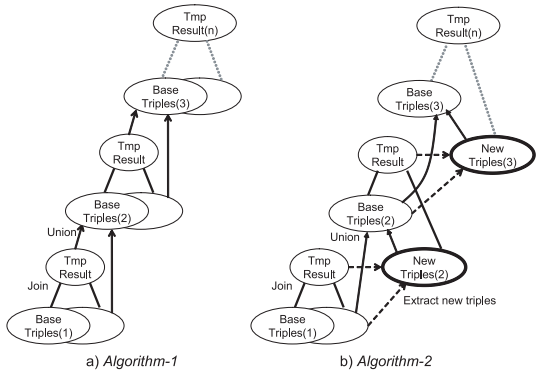


Fig. 6. Processing trees of each algorithm

$i \leftarrow 1$
repeat
  if $i = 1$ **then**
    $BaseTriples_i \leftarrow Input$
  end if
  $TmpResult \leftarrow$ SELF-JOIN $BaseTriples_i$
  $BaseTriples_{i+1} \leftarrow$ UNION $BaseTriples_i, TmpResult$
  $i \leftarrow i + 1$
until $Size\_Of\_BaseTriples_{i+1} = Size\_Of\_BaseTriples_i$

Fig. 7. *Algorithm-1*: Iterate self-join simply

$i \leftarrow 1$
repeat
  if $i = 1$ **then**
    $BaseTriples_i \leftarrow Input$
    $NewTriples_i \leftarrow Input$
  else
    $BaseTriples_i \leftarrow$ UNION $BaseTriples_{i-1}, NewTriples_i$
  end if
  $TmpResult \leftarrow$ JOIN $BaseTriples_i, NewTriples_i$
  Extract $NewTriples_{i+1}$ from $TmpResult$ and $BaseTriples_i$
  $i \leftarrow i + 1$
until $Size\_Of\_NewTriples_i = 0$

Fig. 8. *Algorithm-2*: Detect new triples in every loop iteration

4.4. Transitive closure operation over Pig

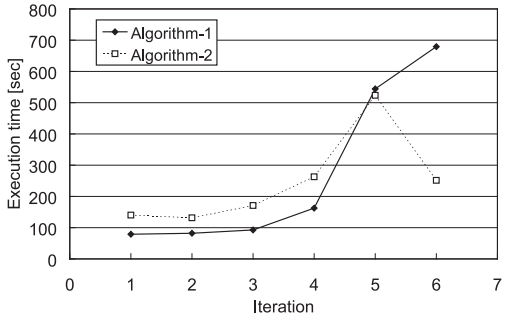
4.4.1. Implementation

RDF-based reasoning based on various languages (e.g. RDF Schema, OWL, SWRL[1]) is currently an active research topic and initially we have chosen to focus on one very common and significant problem, that of inferring new RDF data based on transitive properties, for example, the rdfs:subClassOf property. Such properties can be used to generate new triples by generating a transitive closure, which can be implemented by applying a sequence of self-joins to a table.

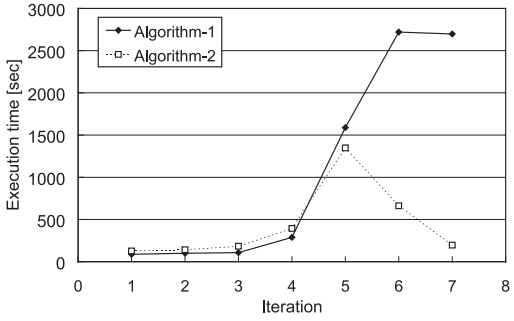
For support of the optimized execution to generate a transitive closure, we implemented two algorithms in Pig Latin and Shell script. Both algorithms are based on squaring evaluation of the iterative algorithms [12], which means that a previous join result become the next join input, as shown in Figure 6. Each detailed algorithm is shown in Figure 7 and 8.

Algorithm-1 in Figure 7 simply iterates self-join to compute the closure. The joined result is combined with each input data and becomes the input of the join at the next iteration. While *Algorithm-1* does not detect new triples at each iteration, *Algorithm-2* finds them out and always keeps two datasets. One stores all triples, a set of original triples and new triples, and the other stores only new triples. At each iteration, *Algorithm-2* performs join of all triples and new triples. Because the number of new triples is fewer than the number of base triples and hence the cardinalities of relations involved in the join are reduced. An obvious drawback is that extracting new triples requires additional processing and our concern is that it might take more time. We compare the two algorithms through several experiments in the next subsection.

[1] Semantic Web Rule Language, http://www.w3.org/Submission/SWRL/



a) knows-1



b) knows-2

Fig. 10. Execution time of the Pig Latin script at each loop iteration

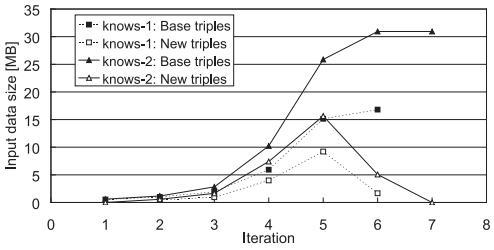


Fig. 9. Input data size of each join operation

4.4.2. Basic evaluation

In our evaluation, two programs implementing each algorithm were executed on 4 machines shown in Table 1. A small set of the Billion Triple Challenge dataset in 2009 [1] was used as the experiment data, and closure of the 'foaf:knows' property was obtained. The small set is approximately 1.13 [GB] in size of *vp-txt*, containing 26 [MB] of the 'foaf:knows' triples as its predicate table. Because, even with the input size, the generated intermediate data would fill the capacity of the temporary space in our experiment environment, we sampled the data uniformly and generated two sets of input data, knows-1 and knows-2. The knows-1 dataset is 2.00% of the original size and knows-2 is 2.26% of the original. Figure 9 shows the input data size of each join operation. *Algorithm-1* reads base triples two times at each iteration, and *Algorithm-2* does the same at the first iteration and reads the base triples and new triples at the rest of the iterations. Figure 9 indicates that the size of new triples was smaller than the size of base triples. The loop lasted 6 iterations for knows-1 and 7 iterations for knows-2.

Figure 10 shows execution time of each loop which corresponds to one execution of the Pig Latin script. For both inputs, *Algorithm-1* finished the execution faster than *Algorithm-2* until the 5th iteration. However, *Algorithm-2* became faster after the 5th iteration because the input data of join in *Algorithm-1* became large and then much larger intermediate data was generated. The intermediate data size reached to 8.3 [GB] at the 5th and 6th iterations in *Algorithm-1* while it was at most 2.3 [GB] in *Algorithm-2*. As shown in Table 7, the total execution time of *Algorithm-2* was shorter than *Algorithm-1*. Table 7 also shows the overhead ratio in the total execution time. Here we observed overheads such as internal processing in Pig between MapReduce jobs, generation of the Pig code, and the termination test of the loop. The overhead ratio depends on the number of MapReduce jobs. Thus the ratio was higher in *Algorithm-2* than in *Algorithm-1* because more jobs were generated to identify new triples. When each MapReduce execution became longer, the ratio became relatively smaller, which was seen in the comparison between the knows-1 and knows-2 results.

Comparing the results produced by the two algorithms, *Algorithm-2* ran faster than *Algorithm-1* with those inputs. Moreover, from the results, it indicates that *Algorithm-2* has an advantage over *Algorithm-1* as the input data size increases.

4.4.3. Improvement by join optimization

In order to execute the closure operation using *Algorithm-2* faster, we considered join optimization and parallel execution. First, we tried to use the most suitable join algorithm according to the properties of the input data. In the Pig release 0.7.0, 'replicated,' 'skewed' and 'merge' `JOIN` are available under several conditions upon the input data. Here, the 'replicated' `JOIN` was applied to *Algorithm-2* because it satisfied the condition that one of the input data size was small enough. In the 'replicated' `JOIN`, Pig performed `JOIN` in the map task without launching the reduce task. Second, we examined increasing the num-

Table 7
Total execution time and the number of the generated MapReduce jobs

	knows-1		knows-2	
	Alg.-1	Alg.-2	Alg.-1	Alg.-2
Total execution time [sec]	1,647	1,487	7,597	3,061
Total MapReduce execution time [sec]	1,500	1,276	7,445	2,807
Overheads ratio [%]	8.91	14.2	2.00	8.30
Total number of the MapReduce jobs	18	29	21	34
Total number of map tasks	90	121	312	200
Total number of reduce tasks	11	18	14	21

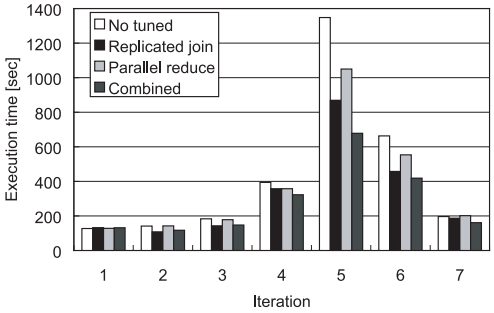


Fig. 11. Execution time of the Pig Latin script at each loop iteration with optimization

ber of reduce tasks, as the input data size and the number of map tasks increased like at the latter iterations in the previous experiment in Section 4.4.2. Note that the number of the reduce tasks was one at default setting, and at the previous experiment as well. This parallel execution was applied to not only the `JOIN` operation but also the `DISTINCT` and `GROUP` operations in this experiment.

Figure 11 shows the execution time of each iteration to process the knows-2 dataset, including the above improvement. Each replicated join iteration was shorter than the corresponding 'No-tuned' iteration, and sometimes much shorter (see the 5th and 6th iterations). However, we also observed that some reduce tasks took time to write a large join result to a file for the next MapReduce job. In the parallel execution, we set the number of the reduce tasks to be the division of the total input size of the reduce phase by 64 [MB], or the maximum number of 4. In this case, the `JOIN` operation was not performed in parallel when the input data size was small enough. The effect of the parallel execution was appeared in the 4th, 5th and 6th iterations but unexpectedly it was less effective than the replicated join. 'Combined' in Figure 11 mixed both optimizations. It basically used the replicated join but when the output size of the join was supposed to be large, it enabled parallel execution. Parallel execution of other operations was also enabled based on the input size of the reduce phase. 'Combined' achieved the best improvement in this experiment and the total execution time was reduced 35% from the execution of *Algorithm-2* in Table 7.

A problem of this approach is that the input and output size of the join is unknown before the execution, and not able to be estimated accurately. This means that it is difficult to select an appropriate join algorithm and an appropriate level of the parallel execution. One possibility is to incorporate an adaptive strategy within the Pig query engine to configure these parameters during query execution.

4.4.4. Effect of parallel execution

There is another advantage of using our proposed architecture for the `RDF-INFER` command execution by rule processing parallelism. Rule processing parallelism can be achieved by processing independent rules in parallel, for example, 'foaf:knows' and 'rdfs:subClassOf' are independent and can be executed in parallel for generating separate transitive closures. The significant aspect here is that generating multiple closures will need to take place at the same time, as the `RDF-INFER` command may need to process a number of rules at the same time. In this scenario, Hadoop is capable of automatically scheduling multiple concurrent joins in order to efficiently utilize the computational power of available nodes and minimize the overall time required to execute the `RDF-INFER` command. We experimented such parallel execution to get two transitive closures of 'foaf:knows' (knows-1 dataset) and 'rdfs:subClassOf' on 8 machines shown in Table 1, and compared with two individual executions. The 8 machines had a 16 map tasks capacity and another 16 reduce tasks ca-

Table 8
Parallel execution of the two transitive closure operations

	Individual execution		Parallel execution	
	knows-1	subClassOf	knows-1	subClassOf
Total time [sec]	1420	742	1443	775
Time increase [%]	-	-	101.65	104.46

pacity. While both closure operations were executed in parallel, the capacities afforded the submitted tasks. Table 8 shows that the parallel execution took about 66% of sum of the two individual executions, with almost no overhead for knows-1 and 4.46% overhead for subClassOf. This shows that multiple transitive closures can be generated by our proposed framework in parallel faster than if they are processed sequentially.

5. Related Work

The RAPID and RAPID+ systems have been developed based on Pig and Hadoop [33,31] and their designs are similar to our proposal. Filtering and aggregation at data load, which are implemented as GFD/GBD in the RAPID and loadFilter in the RAPID+, have the same goal of our RDFLoader(), though an internal storage schema in the HDFS of the RAPID and RAPID+ is not disclosed. In addition, the RAPID implements the Multi-dimensional Join [14] and the RAPID+ does the grouping-based star join processing with a look-ahead method to reduce I/O cost. Such join optimization techniques could be used in conjunction with our approach. Unlike the RAPID and RAPID+ systems, our presented storage schema is the integration of the logical partitioning and the underlying file system format, and our study is extended to the reasoning support and the performance evaluation of the transitive closure operation.

The paper [26] presents the framework that uses a Pig-based RDF stores as a back end of Sesame [11]. The paper mentions about the extended load and store functions to convert RDF to Pig's data model, and the reasoning support using a forward-chaining algorithm. However, the details of the back end implementation are not presented. Our framework does not limit the implementation to provide the repository abstraction layer of Sesame on top of itself, either.

The paper [20] summarizes various cloud-based Linked Data Management systems using recent large-scale data processing technologies, and many of them support RDF. In the paper, our proposed framework is categorized under the Pig/Hadoop based system but HBase [4] can be a replacement of the HDFS. As is obvious, the advantage of the replacement should be considered, because we see the primary performance bottleneck on our framework is currently not in the HDFS but the MapReduce execution system.

As a non-Hadoop system, SYSTAP's Bigdata™ [34] provides a scale-out storage and computing fabric for RDF data. Bigdata builds a B+tree architecture on a distributed system like Bigtable [13] and HBase, and has multiple indices of the mapping between term and ID, and all access paths (subject-predicate-object, predicate-object-subject, object-subject-predicate). The SPARQL and OWL Lite interfaces are supported, and declarative rules such as the forward closure or backward chaining can be dynamically added. The overall architecture is similar to our proposed framework but our framework aims at providing capability of flexibly customizable rule-based processing and integrated processing with non-RDF data. Therefore our framework is based on Pig, allowing procedural programming of the database and user-defined operations, and the technical approach presented in this paper is for extension of Pig in order to speed-up processing large RDF datasets.

As an optimization technique of the `RDF-INFER` operation, there are many existing studies for parallel algorithms to compute transitive closure. The algorithm using a pipeline method [15] cannot be applied to the MapReduce model, but the algorithms proposed in the papers [19,25] or other direct algorithms [12] can be applied. Implementation of these algorithms would not be a sequence of Pig commands as we implemented, but be a program directly using the MapReduce library. For the iterative algorithm, multi-way joins using MapReduce [9,24] seems to speed-up the closure execution.

Another approach towards efficient RDFS reasoning with MapReduce is presented in [37], where optimization for encoding the RDFS ruleset is performed in a higher level than the level of our discussed optimization for multi-joins. The approach is possibly used

for implementing the RDF-INFER operation in our framework.

6. Conclusion

This paper presented an approach towards scalable RDF data processing based on the general data processing platform of Pig and Hadoop. Two extensions for RDF, the RDFLoader() and the RDF-INFER command for allowing further optimization in Pig were proposed. Neither of these requires any changes of the Pig and Hadoop architectures and keeps the advantages of their scalable and flexible data processing capabilities. RDFLoader() was successfully implemented with MapFile provided by Hadoop, which allows users to load the data, specifying a subject and/or a predicate much faster than the original Pig load function and save disk space significantly. Moreover the schema conversion cost by the MapReduce execution is presented, and it is shown that our proposed framework having the extensions could handle 25 million triples which could not be successful in query executions on existing single-node RDF databases. It would be possible to handle larger size of the RDF data in the larger environment. Users of our framework only have to add more Hadoop nodes for storing the data and processing more MapReduce tasks that would simply increase against the data size, as usually other Pig and Hadoop users do for scaling-up their general data processing.

For reasoning support with the RDF-INFER, efficient implementation of the transitive closure operation on this framework was studied. Two squaring-based algorithms were compared, and *Algorithm-2*, which reduces the cardinalities of a join by additional processing, was faster. Because the intermediate data could be much larger in the closure operation against data like 'foaf:knows' in the full data set of the Billion Triple Challenge, *Algorithm-2* seems to be more practical for most applications. Improvement of *Algorithm-2* by enhanced functions of Pig, use of the most suitable join implementations and parallel execution, was examined. As a result, it is possible to improve the performance using them but only when the input and output data size was known in advance. In our future implementation of the RDF-INFER, it may be beneficial to have an adaptive or prediction based query optimization in the Pig query engine. Use of join indices and multiway joins that we introduced in Section 5 should also be applied to the implementation.

As future work, we would like to improve further about the RDF-INFER operation, have scalability experiments using publicly available RDF data [7,6] on larger environments, and then evaluate our proposed framework, in terms of the combined processing capability of the RDF and RDF-annotated data.

Acknowledgment

This work is supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Japanese Ministry of Internal Affairs and Communications.

References

[1] Billion Triple Challenge 2009 Dataset. http://vmlion25.deri.ie/.
[2] DBpedia. http://dbpedia.org.
[3] Hadoop. http://hadoop.apache.org/.
[4] HBase. http://hadoop.apache.org/hbase/.
[5] LZO. http://www.oberhumer.com/opensource/lzo/.
[6] Semantic Web Challenge. http://challenge.semanticweb.org/.
[7] The W3C SWEO Linking Open Data community project. http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData/.
[8] Ubiquitous ID Center. http://www.uidcenter.org/.
[9] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, pages 99–100, 2010.
[10] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web*, pages 1–13, 2001.
[11] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In *Semantics for the WWW*. MIT Press, 2001.
[12] F. Cacace, S. Ceri, and M. Houtsma. A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs. *International Journal of Distributed and Parallel Databases*, 1(14):337–382, 1993.
[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage for Structured Data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, pages 205–218, 2006.
[14] D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. The MD-join: An Operator for Complex OLAP. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 524–533, 2001.
[15] J.-P. Cheiney and C. de Maindreville. A Parallel Strategy for Transitive Cloosure using Double Hash-Based Clustering. In *Proceedings of VLDB*, pages 347–358, 1990.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004.

[17] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. In *Proceedings of VLDB*, 2009.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.

[19] A. Gibbons, A. Pagourtzis, I. Potapov, and W. Rytter. Coarse-Grained Parallel Transitive Closure Algorithm: Path Decomposition Technique. *The Computer Journal*, 46(4):391–400, 2003.

[20] M. Hausenblas, R. Grossman, A. Harth, and P. Cudré-Mauroux. Large-scale Linked Data Processing: Cloud Computing to the Rescue? In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER)*, pages 246–251, 2012.

[21] I. Herman, P. Mika, T. Berners-Lee, and Y. Raimond. Microdata, RDFa, Web APIs, Linked Data: Competing or Complementary? Panel Discussion slides of the 5th Linked Data on the Web Workshop (LDOW2012), 2012. http://events.linkeddata.org/ldow2012/slides/Bizer-LDOW2012-Panel-Background-Statistics.pdf.

[22] I. Hickson, G. Kellogg, J. Tennison, and I. Herman. Microdata to RDF. http://www.w3.org/TR/2012/NOTE-microdata-rdf-20121009/, October 2012.

[23] M. Ley and et al. The DBLP Computer Science Bibliography. http://www.informatik.uni-trier.de/ ley/db/.

[24] S. Lynden, Y. Tanimura, I. Kojima, and A. Matono. Dynamic Data Redistribution for MapReduce Joins. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 717–723, 2011.

[25] Z. Miao, J. Wang, B. Zhou, Y. Zhang, and J. Lu. Method for Extracting RDF(S) Sub-Ontology. In *Proceedings of the International Conference on Cyberworlds*, 2008.

[26] P. Mika and G. Tummarello. Web Semantics in the Clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[27] C. Olston, B. Reed, A. Silbersten, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *Proceedings of USENIX*, pages 262–273, 2008.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of ACM SIBMOD*, pages 1099–1110, 2008.

[29] O. O'Malley and A. C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. http://sortbenchmark.org/Yahoo2009.pdf, 2009.

[30] openRDF.org. Sesame. http://www.openrdf.org/.

[31] P. Ravindra, V. V. Deshpande, and K. Anyanwu. Towards scalable RDF graph analytics on MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud (MDAC)*, 2010.

[32] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 222–233, 2009.

[33] R. Sridhar, P. Ravindra, and K. Anyanwu. RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web. In *Proceedings of the 8th International Semantic Web Conference (ISWC)*, pages 715–730, 2009.

[34] SYSTAP. Bigdata. http://www.systap.com/bigdata.htm.

[35] Y. Tanimura, A. Matono, I. Kojima, and S. Sekiguchi. Storage Scheme for Parallel RDF Database Processing Using Distributed File Ssytem and MapReduce. In *Proceedings of HPC Asia*, pages 312–319, 2009.

[36] Y. Tanimura, A. Matono, S. Lynden, and I. Kojima. Extensions to the Pig data processing platform for scalable RDF data processing using Hadoop. In *Proceedings of the 1st International Workshop on Data Engineering meets the Semantic Web in conjunction with the 22nd International Conference on Data Engineering (ICDE)*, pages 251–256, 2010.

[37] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable Distributed Reasoning Using MapReduce. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2009.

[38] M. van Assem, A. Gangemi, and G. Schreiber. RDF/OWL Representation of WordNet. W3C Workding Draft, June 2006. http://www.w3.org/TR/2006/WD-wordnet-rdf-20060619/.

[39] World Wide Web Consortium (W3C) Recommendation. Resource Description Framework (RDF). http://www.w3.org/RDF/, 2004.