High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store

Kurt Rohloff
BBN Technologies
Cambridge, MA, USA
krohloff@bbn.com

Richard E. Schantz
BBN Technologies
Cambridge, MA, USA
schantz@bbn.com

ABSTRACT

In this paper we discuss the use of the MapReduce software framework to address the challenge of constructing high-performance, massively-scalable distributed systems. We discuss several design considerations associated with constructing complex distributed systems using the MapReduce software framework, including the difficulty of scalably building indexes. We focus on Hadoop, the most popular MapReduce implementation. Our discussion and analysis are motivated by our construction of SHARD, a massively scalable, high-performance and robust triple-store technology on top of Hadoop. We provide a general approach to construct an information system from the MapReduce software framework that responds to data queries. We provide experimental results generated of an early version of SHARD. We close with a discussion of hypothetical MapReduce alternatives that can be used for the construction of more scalable distributed computing systems.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design – *methodologies.*

General Terms

Design, Algorithms, Software Engineering, Performance, Design, Experimentation.

Keywords

Distributed computing, MapReduce, Programming, Systems, Semantic Web, Graph Data, SPARQL, Performance Evaluation.

1. INTRODUCTION

Lately there have been a number of advances in software frameworks, such as MapReduce [4], that can be used to address the challenges inherent to the construction of highly parallel, high-performance and highly scalable distributed computing systems much easier. Although MapReduce has been very successful as a distributed computing substrate for relatively simple highly parallel applications like search and data storage, there have been few more complex systems such as scalable data management systems built using this or similar frameworks. We speculate that this is primarily because these frameworks (and MapReduce in particular) are too low-level [5].

In this paper we focus on the design aspects inherent to using the MapReduce software framework to construct highly-parallel, high-performance and scalable information management systems. We also suggest alternative software frameworks for the easier design and development of highly-scalable information management systems. Our insight and discussions in these areas are motivated by our experience designing, constructing and evaluating our initial implementation of the SHARD (Scalable, High-Performance, Robust and Distributed) triple-store. A triple-store is an information storage and retrieval environment for graph data, traditionally represented in RDF formats [17]. (RDF is a standard data format for representing triples which are edges in data graphs.) SHARD persists graph data as RDF triples and responds to queries over this data in the SPARQL query language. We use the Hadoop implementation of MapReduce to construct SHARD. We discuss lessons learned and insight for future revisions of our design and implementation. To support these claims, we present our initial experimental results evaluating SHARD with the standard LUBM benchmark for triple-stores [6].

The problem context driving our information system design is the need for web-scale information systems. For example, one of the singular advancements over the past several years in the Semantic Web domain has been the explosion of graph data available in semantic formats [10]. Unfortunately, Semantic Web data processing technologies, which rely on graph data information systems, are designed for deployment on a single (or a small number of) machine(s). This is fine when data is small, but current methodologies to design high-level information systems for graph data are limited by data processing and analysis bottlenecks with graphs on the order of a billion edges [10][18]. These scalability constraints are the greatest barriers to achieve the fundamental web-scale Semantic Web vision and have hindered the broader adoption of Semantic Web technologies. Other scalable approaches to triple-store design based on key-value and column stores (such as Cassandra [3] and Project Voldemort [16], among many others) are feasible. However, these other technologies do not provide the native data processing capabilities supported by MapReduce implementations like Hadoop that enable more efficient query processing.

We discuss work-in-progress to address these scalability limitations in the Semantic Web by designing and developing SHARD using the MapReduce software framework. In particular, we describe initial results from deploying an early version of SHARD into an Amazon EC2 cloud [1] and running the standard LUBM triple-store benchmark. We find that SHARD already performs better than current industry-standard triple-stores for datasets, on the order of a billion triples.

The remainder of this paper is organized as follows. In Section 2 we provide an introduction to MapReduce, Hadoop, and their relevant properties for the design of information management systems. In Section 3 we provide a brief overview of relevant SHARD design goals and graph data processing concepts. In Section 4 we describe the design of information management systems such as SHARD using the MapReduce software framework. In Section 5 we describe our experimental results from the deployment of an early version of SHARD into an Amazon EC2 cloud. In Section 6 we discuss design insight we gained from experimentation. We conclude in Section 7 with a

discussion of ongoing and alternative designs for high-performance, massively scalable information systems.

2. MAPREDUCE AND HADOOP

MapReduce is a software framework for processing and generating large data sets [4]. Users specify a map function that splits data into key/value pairs and a reduce function that merges all key/value pairs based on the key. Many real world low-level tasks are expressible in this model including word counting and the Page-rank algorithm.

The MapReduce software framework is easily parallelizable for execution on large clusters of commodity machines. This enables the construction of high-performance, highly-scalable applications. One of the more popular MapReduce implementations is Hadoop [8]. Hadoop takes care of the details of managing data on compute nodes through the Hadoop Distributed File System (HDFS), scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows for the design and implementation of high-level functionality using the MapReduce framework to construct high-performance and highly scalable applications.

A key aspect of the MapReduce software framework, as expressed in the Hadoop implementation, is the use of a special, centralized compute node, called the NameNode. The name node directs the placement of data onto compute nodes through HDFS, assigns compute jobs to the various nodes, tracks failures and manages the shuffling of data after the Map step completes.

There are several benefits as well as drawbacks from using MapReduce to design high-performance information systems, irrespective of how those information systems are designed. These benefits include that MapReduce implementations such as Hadoop are generally easy to set up and debug, and applications are easy to write efficiently in several programming languages. The drawbacks of the Hadoop implementation of the MapReduce framework include that only Java programs can be used natively for more complex applications, it is difficult to run Java code on compute nodes that need runtime customization, NameNode creates a bottleneck for HDFS access, and NameNode failures can be catastrophic.

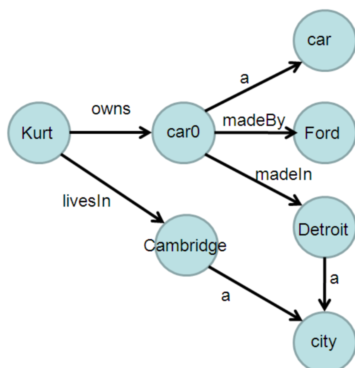

Figure 1: A Small Graph of Triple Data.

3. DESIGN GOALS

Our primary information system design motivations are the ability to persist and rapidly query very large data graphs. To align with Semantic Web data standards, we consider graphs represented as subject-predicate-object triples [2][7]. A small example graph can be seen in Figure 1 that contains 7 triples – Kurt lives in Cambridge, Kurt owns an object car0, car0 is a car, car0 was made by Ford, car0 was made in Detroit, Detroit is a city and Cambridge is a city.

We use SPARQL [19] as a representative query language - it is the standard Semantic Web query language. SPARQL semantics are general purpose and similar to the more well-known SQL. An example SPARQL query for the above graph data is the following:

```
SELECT ?person
WHERE  {
   ?person :owns ?car .
   ?car :a :car .
   ?car :madeIn :Detroit .
  }
```

The above SPARQL query has three clauses and asks for all matches to the variable `?person` such that `?person` owns an entity represented by the variable `?car` which is a car and was made in Detroit. Note that the above query can be represented as a directed graph as seen in Figure 2.

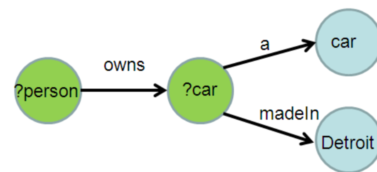

Figure 2: A Directed Graph Representation of a Query.

Processing of SPARQL queries in the context of a data graph such as the one above consists of identifying which variables in the query clauses can be bound to nodes in the data graph such that the query clauses align with the data triples. This alignment process for query processing is fairly general across many data representations and query languages. An example of this alignment for our example query and data can be seen in Figure 3.

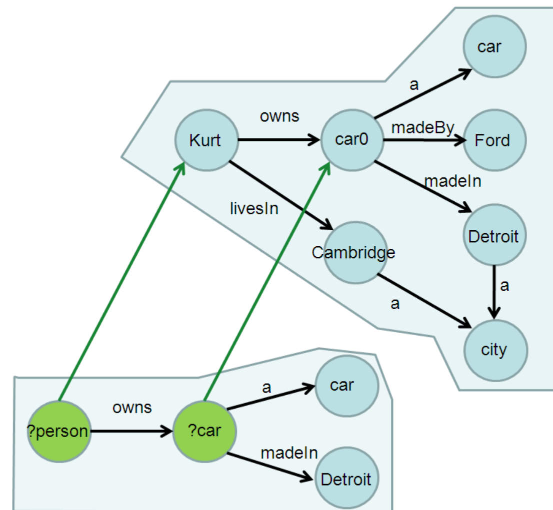

Figure 3: An Alignment of SPARQL Query Variables with Triple Data.

Our functional design goals for the SHARD triple-store are to:

1. Serve as a persistent store for triple data in RDF format.
2. Serve as a SPARQL endpoint to process SPARQL queries.

There have been a number of other design approaches for triple-stores with similar if not the same functional design goals [18]. Several of these triple-stores have achieved very good performance on single compute-node systems by using designs based around memory mapping index information [11]. However, disk and memory limitations have driven the need for distributed computing approaches to triple-stores [12][14]. There have been a number of recent attempts to develop designs of triple stores using distributed computing frameworks [20].

4. SYSTEM DESIGN

4.1 Data Persistence

In order to use a distributed computing approach to information management system design, it is generally infeasible to pass large volume input data directly to and from the user. This data passing would involve the coordinated transfer of data onto and off of the compute nodes when data needs to be processed. The large scale of data makes this approach impractical due to data churn. Consequently, large input (data, queries) and output (query results) data sets need to be stored directly on the compute nodes. In order to best leverage the MapReduce software framework and its Hadoop implementations to construct an information management system, we made this design decision with the understanding that the input data and output results are generally very large and not feasible to output directly to the user. The data storage directly on compute nodes is done natively using the Hadoop implementation of MapReduce by placing data in the HDFS distributed file system.

We persist SHARD data in flat files in the HDFS file system such that each line of the triple-store text file represents all triples associated with a different subject. Consider the following exemplar line saved in SHARD from the LUBM domain that represents three triples associated with the entity subject `Pub1`:

```
Pub1 :author Prof0 :name "Pub1" a :Publication
```

This line represents that the entity `Pub1` has an author entity `Prof0`, `Pub1` has a name “Pub1” and that `Pub1` is a publication.

Although this approach to persisting triple data as flat text files is rudimentary as compared to other information management approaches, we found that it offers a number of important benefits for several general application domains. For one, this approach, particularly in the HDFS implementation, brings a level of automated robustness by replicating data and MapReduce operations across multiple nodes. The data is also stored in a simple, easy to read format that lends itself to easier, user focused drill-down diagnostics of query results returned by the triple-store. Most importantly, however, although this approach to storing triples is inefficient for query processing that requires the inspection of only a small number of triples, this approach is efficient in the context of Hadoop for scanning over large sets of triples to respond to queries that will generate a large number of results, as Hadoop natively scans over input data during the Map stage of its Map-Reduce operations.

4.2 Query Processing

MapReduce provides only simple data manipulation techniques by splitting data into key-value pairs, and accumulating all values with the same keys. In order to provide more advanced query processing that can leverage highly scalable implementations of the MapReduce software framework, information systems would need to iterate over clauses in queries to incrementally attempt to bind query variables to literals in the triple data while satisfying all of the query constraints. Each iteration consists of a MapReduce operation for a single query clause. This iteration is non-trivial because results of previous clauses would need to be continually joined with the results of more recent clauses over the iterations of the MapReduce steps. We describe here how we designed these iterations with a focus on joining intermediate results as an approach to constructing more complex systems.

We use our SHARD context of graph data and SPARQL queries to concretely discuss a design for this iterative query processing using the MapReduce software framework. A schematic overview of this iterative query binding process for the graph data and SPARQL context can be seen in Figure 4. This schematic consists of multiple MapReduce operations.

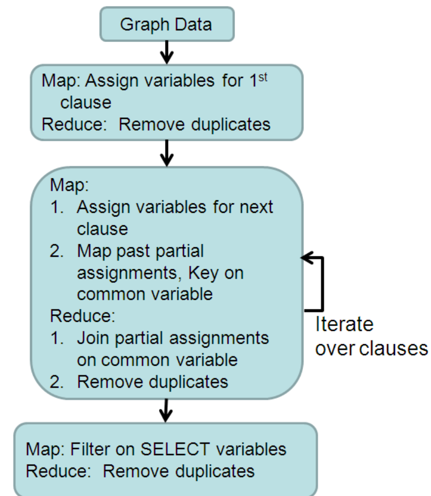

Figure 4: A Schematic Overview of the Iterative Algorithm to process SPARQL queries with Triple Data.

The first map MapReduce step maps the triple data to a list of variable bindings which satisfy the first clause of the query. The key of the Map step is the list of variable bindings. The Reduce step removes duplicate results and saves them to disk with the variable bindings as the key.

The intermediate query binding steps continue to iteratively bind variables to literals as new variables are introduced by processing successive query clauses and/or filtering the previous bindings which cannot fit the new clauses. The intermediate steps perform a MapReduce operation over both the triple data and the previously bound variables which were saved to disk.

The ith intermediate Map step identifies all variables in the triple-data which satisfy the ith clause and saves this result with the key being any variables in the ith clause which appeared in previous clauses. The value of this Map step is the bindings of other variables not previously seen in the query clauses, if any. This iteration of the Map set also rearranges the results of the previous variable bindings saved to disk to the same name of a variable key in the ith clause that appeared in previous clauses. The value of this key-value pair is the list of variable bindings which occurred in previous clauses but not in the ith clause.

The ith Reduce step runs a join operation over the intermediate results from the Map step by iterating over all pairs of results from the previous clause and the new clause with the same key assignment.

This iteration of map-reduce-join continues until all clauses are processed and variables are assigned which satisfy the query

clauses. SHARD is designed to save intermediate results of the query processing to speed up the processing of similar later queries. The storage of intermediate results is a byproduct of the Hadoop MapReduce implementation.

The final MapReduce step consists of filtering bound variable assignments to satisfy the `SELECT` clause of the SPARQL query. In particular, the Map step filters each of the bindings, and the Reduce step removes duplicates where the key value for both Map and Reduce are the bound variables in the `SELECT` clause.

5. EXPERIMENTATION

To test the performance of our general design of a scalable information management system based on the MapReduce framework, we developed an early version of SHARD using the Cloudera version of the Hadoop implementation that we deployed onto an Amazon EC2 cloud environment of 20 XL compute nodes [1] running RedHat Linux and Cloudera Hadoop. The version of SHARD we deployed for evaluation supports basic SPARQL query functionality (without support for prefixes, optional clauses or results ordering) over full RDF data. This unimplemented functionality is generally associated with the pre- or post-processing of queries and we don't expect that adding this extra functionality will substantially detract from the performance exhibited by the current implementation of SHARD. Although possible to implement, the deployed version of SHARD does not perform any query manipulation/reordering/etc… normally done for increased performance by SPARQL endpoints in mature triple-stores. Also, the deployed version of SHARD does not yet take advantage of any possible query caching made possible by our design choices.

5.1 LUBM Benchmark

We used the LUBM benchmark to evaluate the performance of SHARD. The LUBM benchmark creates artificial data about the publishing, coursework and advising activities of students and faculty in departments in universities.

The LUBM code natively generates OWL ontology files [15]. OWL ontology files represent relationships between properties, but because our early version of SHARD takes N3 (an RDF serialization format) data as input, we provided functionality to convert the generated LUBM data into N3 format over many universities and automatically store this generated data in the SHARD HDFS backend using Hadoop. We used code from the LUBM benchmark to generate triple data for 6000 universities which is approximately 800 million triples to parallel the performance evaluations made in a previous triple-store comparison study [18].

After loading the triple data into the SHARD triple store, We evaluated the performance of SHARD in responding to queries 1, 9 and 14 of LUBM as was done in the previous triple-store study. Query 1 is very simple and asks for the students that take a particular course and returns a very small set of responses. Query 9 is relatively more complicated query with a triangular pattern of relationships - it asks for all teachers, students and courses such that the teacher is the adviser of the student who takes a course taught by the teacher. Query 14 is relatively simple as it asks for all undergraduate students (but the response is very large).

5.2 Performance

SHARD achieved the following query response times for 6000 universities (approx. 800 million triples) using the LUBM benchmark when deployed on an Amazon AWS cloud with 20 compute nodes:

Query 1: 404 sec. (approx 0.1 hr.)
Query 9: 740 sec. (approx 0.2 hr.)
Query 14: 118 sec. (approx 0.03 hr.)

We generally found the SHARD performance increased with the number of compute nodes, but we found this performance increase to be sub-linear. This sub-linear increase was mostly likely due to the communication overhead of the MapReduce steps.

For comparison, in the triple store study [18] the industrial single-machine DAMLDB triple-store (released as the open-source project Parliament [1]) was able to achieve the following performance on the same queries coupled with the Sesame[2] and Jena[3] Semantic Web frameworks to aid query processing.

Sesame+DAMLDB took:

Query 1: approx 0.1hr.
Query 9: approx 1 hr.
Query 14: approx. 1 hr.

For Jena+DAMLDB we have no data on performance over 550 million triples due to the difficulty of loading triples into this dataset, but based on observed trends this triple-store probably would of taken the following:

Query 1: approx 0.001 hr.
Query 9: approx 1 hr.
Query 14: approx. 5 hr.

Note that the only query where SHARD performed noticeably worse than DAMLDB was on query 1. Query 1 returns a very small subset of literals bound to variables. Although MapReduce is traditionally used to build indices, its implementations (e.g. Hadoop) provide little native support for accessing data stored in HDFS files. Conversely, DAMLDB has some special indexing optimizations for simple queries like that for Query 1, that are not yet implemented in SHARD. We discuss how this aspect of MapReduce may be improved upon below. Except for this one exception, SHARD performed better than other known technologies due to the highly parallel implementations of the MapReduce framework that we leverage in our design of SHARD. Also, due to the inherent scalability of the Hadoop and HDFS approach to the SHARD design, the SHARD triple-store could potentially be used for extremely large datasets (more than billions of triples) without requiring any specialized hardware, as is required for other monolithic triple-stores.

6. DESIGN INSIGHTS

The performance of SHARD over previous triple-store implementations demonstrate the viability of our information system design approach based on MapReduce. Our design enables the efficient search through data to find matches that satisfy queries. This design is easily distributed across many compute nodes for highly parallel and highly scalable operation. It is also low-cost as it can run on commodity hardware.

There are a number of areas for improvement in an alternative to the MapReduce software framework and its implementations for the easier design of information systems in general and triple-stores in particular. Most notably, the MapReduce, and consequently our design, are biased towards operations over large

[1] http://parliament.semwebcentral.org/

[2] http://www.openrdf.org/

[3] http://jena.sourceforge.net/

datasets without the search for individual key-value pairs. This could be improved upon with native indexing capabilities, possibly supported during pre-processing operations. These pre-processing operations could also be used to reason over the data, so that SHARD could correctly respond to queries that require reasoning.

A more advanced modification to support information system design would be an alternative software framework that provides better data linking. Instead of having to store lists of data in flat files in an HDFS-like construct, a software framework could provide a native linked-data construct that pairs data elements with pointers to related data. This linked data framework would provide faster localized query processing without requiring exhaustive search of the data set on every query request.

7. ONGOING WORK

Development work is ongoing with our information system design based on the MapReduce software framework. Based on our experience with the initial SHARD deployment, we have several short- and long-term activities to further improve performance and applicability from both a design and software framework perspective.

First among the improvements is a more effective method to index data. This will most likely need to be supported by an alternative to the MapReduce framework that supports native indexing instead of basic Map operations over all data elements.

Additional performance improvement of our design in a targeted production environment could be provided by using cached partial results both locally for high-performance parallel operations and globally by a NameNode-like entity that tracks local caching of partial results.. This will require additional capability in a software framework to track partial results that were previously cached and possibly to track which cached results could be thrown out to save disk space in the cloud (if this is a deployment concern.)

There is a general need for improved tools for high-level highly-parallel computations in clusters. The MapReduce tools we explore operate at a low level. It is difficult to coordinate the storage and use of intermediate results in parallel without frequent, temporally expensive disk access. An enhanced software framework might provide this by caching data in memory and persisting data to disk as a backup for failure recovery.

8. ACKNOWLEDGMENTS

The authors would like to thank Gail Mitchell, Doug Reid and Prakash Manghwani from BBN, Philip Zeyliger from Cloudera and Hanspeter Pfister from Harvard University for their assistance.

9. REFERENCES

[1] Amazon. (2010) Amazon EC2 Instance Types. Retrieved from http://aws.amazon.com/ec2/instance-types/

[2] Berners-Lee, Tim; James Hendler and Ora Lassila (May 17, 2001). "The Semantic Web". Scientific American Magazine.

[3] Cassandra. (2010) Retrieved from http://cassandra.apache.org/

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

[5] DeWitt D., Stonebraker M. MapReduce: A major step backwards. databasecolumn.com. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/. Retrieved 2010-08-29.

[6] Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics 3(2) (2005) 158–182

[7] Grigoris A., van Harmelen F. A Semantic Web Primer, 2nd Edition. The MIT Press, 2008.

[8] Hadoop. (2010). Apache Hadoop. Retrieved from http://hadoop.apache.org/

[9] Hendler J., Web 3.0: The Dawn of Semantic Search. In IEEE Computer, Jan. 2010.

[10] Kiryakov A., Tashev Z., Ognyanoff D., Velkov R., Momtchev V., Balev B., Peikov I. "Validation goals and metrics for the LarKC platform." LarKC Report FP7 – 215535. Retrieved from http://www.larkc.eu/wp-content/uploads/2008/01/larkc_prefinal-version_d552_validation-goals-and-metrics-for-the-larkc-platform.pdf. 2009.

[11] Kolas D., Emmons I. and Dean M., Efficient Linked-List RDF Indexing in Parliament. In the Proceedings of the Scalable Semantic Web (SSWS) Workshop of ISWC '09, 2009.

[12] Li P., Zeng Y., Kotoulas S., Urbani J., and Zhong N., "The Quest for Parallel Reasoning on the Semantic Web," in Proceedings of the 2009 International Conference on Active Media Technology, LNCS, 2009.

[13] LinkingOpenData. (2010) Retrieved from http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

[14] Mika, P. and Tummarello, G. 2008. Web Semantics in the Clouds. IEEE Intelligent Systems 23, 5 (Sep. 2008), 82-87.

[15] OWL. (2010) Web Ontology Language (OWL.) Retrieved from http://www.w3.org/TR/owl2-overview/

[16] Project Voldemort. (2010) Retrieved from http://project-voldemort.com/

[17] RDF. (2010) Resource Description Framework (RDF) Retrieved from http://www.w3.org/RDF/

[18] Rohloff K., Dean M., Emmons I., Ryder D., Sumner J.. "An Evaluation of Triple-Store Technologies for Large Data Stores." 3rd International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS '07), Vilamoura, Portugal, Nov 27, 2007.

[19] SPARQL. (2010) SPARQL Query Language for RDF http://www.w3.org/TR/rdf-sparql-query/

[20] Urbani J., Kotoulas S., Oren E., and van Harmelen F., "Scalable Distributed Reasoning using MapReduce," In Proceedings of the ISWC '09, 2009.