Clause-Iteration with MapReduce to Scalably Query Data Graphs in the SHARD Graph-Store

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

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

ABSTRACT

Graph data processing is an emerging application area for cloud computing because there are few other information infrastructures that cost-effectively permit scalable graph data processing. We present a scalable cloud-based approach to process queries on graph data utilizing the MapReduce model. We call this approach the Clause-Iteration approach. We present algorithms that, when used in conjunction with a MapReduce framework, respond to SPARQL queries over RDF data. Our innovation in the Clause-Iteration approach comes from 1) the iterative construction of query responses by incrementally growing the number of query clauses considered in a response, and 2) our use of flagged keys to join the results of these incremental responses. The Clause-Iteration algorithms form the basis of our scalable, SHARD graph-store built on the Hadoop implementation of MapReduce. SHARD performs favorably when compared to existing "industrial" graph-stores on a standard benchmark graph with 800 million edges. We discuss design considerations and alternatives associated with constructing scalable graph processing technologies.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – *methodologies.*

General Terms

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

Keywords

Distributed Computing, Graph Data, MapReduce, Algorithms, Systems, Semantic Web, SPARQL, Performance Evaluation.

1. INTRODUCTION

Intensive computing over graph data has become increasingly important in diverse application domains from social networking and genomics to crime fighting and Semantic Web technologies. Unfortunately, advances in scalable information management technologies for intensive computing (cloud-based or otherwise) have not kept pace to support the processing of these increasingly large graph data applications. Highly scalable cloud-based approaches to traditional information storage, management and processing technologies, such as databases can only be partially leveraged for scalable graph data processing in the clouds. For example, current state-of-the-art industrial 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) [34]. This is fine when data is small, but current methodologies to design high-performance information systems with embedded graph processing capabilities are limited by data processing and analysis bottlenecks that consistently emerge with graphs on the order of a billion edges [17][34]. Even if an information system could manage data graphs with a billion edges, it would still be insufficient to address the ongoing explosion of graph data available in semantic formats [10] that can be used in a myriad of application areas. These scalability constraints are the greatest barriers to achieve the fundamental web-scale Semantic Web vision [2] and have hindered the broader adoption of Semantic Web technologies.

There have of course been several notable monolithic design approaches for query processing with similar if not the same functional design goals [34]. Several of these graph-stores have achieved very good performance on single compute-node systems by using designs based around memory mapping index information [19]. However, disk and memory limitations have limited scalability and driven the need for distributed computing approaches.

We introduce a scalable cloud-based approach to process queries on graph data based on iterating over clauses in graph queries to construct query responses using the MapReduce paradigm [5]. We call this approach the Clause-Iteration approach. We provide algorithms to use the Clause-Iteration approach to process SPARQL [38] queries over RDF data [33]. (SPARQL is a standard Semantic Web [10] graph data query language and RDF is a standard Semantic Web data format for representing data graphs.) Our innovation in the Clause-Iteration approach comes from parallelization techniques for the iterative construction of query responses by incrementally growing the number of query clauses considered in an incremental response, overlaid with our use of flagged keys to join the results of these incremental responses.

We implement the Clause-Iteration algorithms for SPARQL queries over RDF data in the SHARD (Scalable, High-Performance, Robust and Distributed) graph-store [37]. SHARD is built on top of Hadoop [9], a popular MapReduce implementation. We present initial experimental results evaluating the Clause-Iteration approach from an early version of SHARD that we deployed into an Amazon EC2 cloud [1]. We perform our evaluation with the standard LUBM benchmark for graph stores [7] with an 800-million edge graph data set. We find that SHARD, using the Clause-Iteration approach, performs better than non-parallel graph-stores currently used in industry [34].

The remainder of this paper is organized as follows. In Section 2 we provide a brief overview of relevant graph data query system design goals with respect to SPARQL-like query languages and data represented in formats similar to RDF. In Section 3 we present our Clause-Iteration approach that responds to SPARQL queries over RDF data. In Section 4 we describe our experimental results from the use of the Clause-Iteration approach in our SHARD graph-store to process SPARQL queries in an Amazon EC2 cloud. In Section 5 we review current approaches and related work for large-scale cloud-based graph data processing that can be run on parallelized commodity computing environments that address similar challenges as the Clause-Iteration approach and SHARD. In Section 6 we discuss design insight we gained from experimentation and ongoing and alternative designs for high-performance, massively scalable information systems. We discuss additional future work in Section 7.

2. GRAPH QUERY DESIGN GOALS

Our primary design goal is to enable rapid SPARQL-like querying over RDF-like graph data on very large data graphs using inexpensive hardware and parallel computation. We focus on algorithms which perform better when responding to queries that are more complicated than simple edge-lookups where a substantial subset of the stored graph data may be returned by the query.

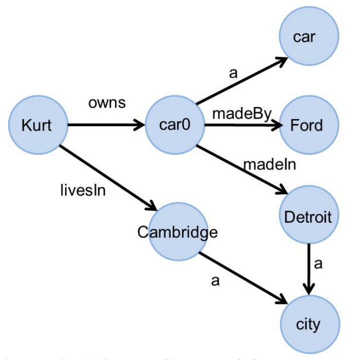

Figure 1: A Small Graph of Graph Data.

To align with general Semantic Web data standards, we consider graphs represented as subject-predicate-object edges [2][8], although the algorithms discussed here can be generalized to other situations. A small version of the kind of graph we consider can be seen in Figure 1 which we will use as part of a running example. This graph dataset contains 7 edges to represent that 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 take an approach to processing graph data based upon the popular MapReduce cloud computing paradigm [5] which enables robust parallel processing over large datasets on low-cost commodity hardware. In particular, we used the Hadoop [9] implementation of MapReduce. Graph data could be represented in several formats to be passed as input to the algorithms which run the SPARQL query processing, including general RDF. In order to best leverage popular MapReduce implementations like Hadoop to construct practical graph query systems, we assume data is stored directly on compute nodes in native file systems like the HDFS distributed file system. We assume that each line in the data file represents all edges from a single node. For example the input line,

```
Kurt owns car0 livesIn Cambridge
```

represents all edges with `Kurt` as the subject: the entity `Kurt` owns an entity `car0`, `Kurt` lives in `Cambridge`. Lists of such lines represent all of the graph data the algorithm runs over.

Although our approach to representing edge 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 in practice. For one, this approach, particularly when used with 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 graph-store. Most importantly, however, is that although this approach to storing edges is inefficient for query processing that requires the inspection of only a small number of edges, this approach is efficient in the context of the use of our query algorithm when used with Hadoop to scan over large sets of edges to respond to queries that will generate a large number of results. Hadoop natively scans over input data during the Map stage of its MapReduce operations.

The SPARQL-like queries we consider have semantics remarkably similar to also well-known SQL semantics. In particular, we focus on queries with multiple clauses, multiple variables and literals which are represented in the query clauses, and an identification of a subset of those variables which should be returned in response to the query. A SPARQL-like query for the above graph data that we use as a running example is the following:

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

The above query has three clauses and asks for all matches to the variable `?person` such that 1) `?person` owns an entity represented by the variable `?car`, 2) `?car` is a car and 3) `?car` was made in Detroit. Note that the above query can be represented as a directed graph as seen in Figure 2.

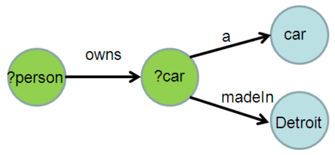

Figure 2: A Directed Graph Representation of a Query.

Processing of queries in the context of a data graph consists of identifying which variables in the query clauses can be mapped to subsets of nodes in the data graph such that the query clauses align with data edges. 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 where when `?person` is aligned with Kurt and `?car` is aligned with car0, the query clauses match corresponding edges in the data graph. In this instance, the query clauses align with the edges that indicate Kurt owns an object car0, car0 is a car and car0 was made in Detroit.

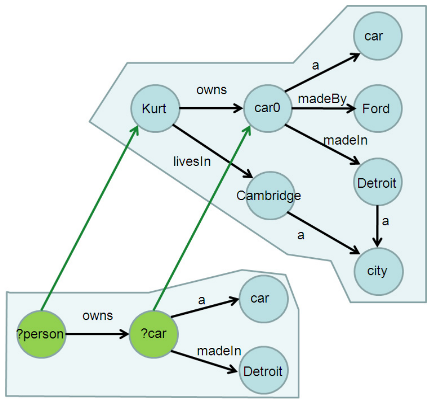

Figure 3: An Alignment of SPARQL Query Variables with Graph Data.

3. CLASUE-ITERATION DESIGN

3.1 Approach Overview

The basis of the Clause-Iteration approach is to iterate over clauses in queries to incrementally attempt to bind query variables to data nodes in the graph data while satisfying all of the query constraints. The goal of this approach is to utilize MapReduce-style operations for high parallelization on low-cost commodity hardware.

An Iteration algorithm coordinates the high-level operation of iterated MapReduce jobs with one iteration for each clause in the query. A schematic overview of the Iteration Algorithm for iterative data selection and query binding can be seen in Figure 4. We describe this algorithm in more detail below in the subsection immediately following, but we first give an overview of the algorithms used for the MapReduce operations called by the Iteration Algorithm.

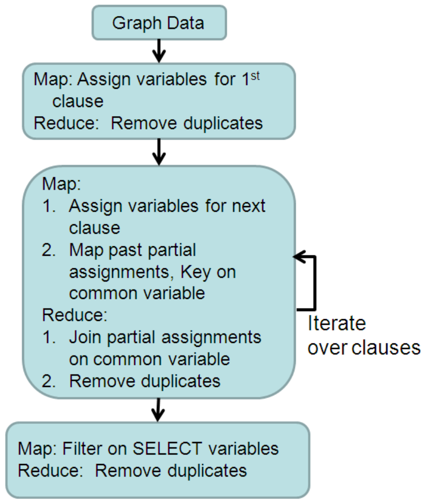
Graph Data
Map: Assign variables for 1st clause
Reduce: Remove duplicates
Map:
1. Assign variables for next clause
2. Map past partial assignments, Key on common variable
Reduce:
1. Join partial assignments on common variable
2. Remove duplicates
Iterate over clauses
Map: Filter on SELECT variables
Reduce: Remove duplicates

Figure 4: A Schematic Overview of the Iteration Algorithm in the Clause-Iteration Approach.

MapReduce operations iteratively select data matching a single query clause and joins that selected data to data subgraphs that align with previously query clauses based on common variable bindings.

The initial map step identifies all feasible bindings of graph data to variables in the first query clause. The output key of the initial map step is the list of variable bindings and the output values are set to null. The initial reduce step removes duplicate bindings without further modifying the output of the initial map step.

The intermediate MapReduce jobs continue to construct query responses by iteratively binding graph data to variables in later clauses as new variables are introduced and then joining these new bindings to the previous bound variables such that the joined bound variables align with iteratively increasing subsets of the query clauses. The intermediate steps perform MapReduce operations simultaneously over both the graph data and the previously bound variables which were saved to disk to perform this operation.

A final MapReduce step consists of filtering bound variable assignments to obtain just the variable bindings requested in the `SELECT` clause of the original 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.

3.2 Iteration Algorithm

A pseudo-code version of the MapReduce Iteration algorithm used for the Clause-Iteration approach to process queries on graph data is seen in Figure 5. This algorithm takes graph data and a query as input and uses three different types of MapReduce jobs (`firstClauseMapReduce`, `intermediateClauseMapReduce` and `selectMapReduce`) to identify subgraphs that match the query which are returned by the algorithm.

```
SPARQLEngine(TripleData triples, Query query):
mrInput = triples
run firstClauseMapReduce(mrInput, mrOutput,
   query.clause(0))
boundVars = query.clause(0).getVars()
for i= 1 to query.numClauses-1
   mrInput = union(triples, mrOutput)
   curVars = query.clause(i).getVars()
   comVars = intersection(boundVars,curVars)
   run intermediateClauseMapReduce(mrInput,
      mrOutput, query.clause(i), comVars)
mrInput = mrOutput
run selectMapReduce(mrInput, mrOutout,
   query.select())
return mrOutput
```

Figure 5. Iteration Algorithm to Iteratively Call MapReduce Jobs to Respond to Queries

The Iteration algorithm uses the `firstClauseMapReduce` MapReduce job is to identify which edges in the data graph match the first query clause. Hence the `firstClauseMapReduce` MapReduce job is passed the edge data (`mrInput`) and the first query clause (`query.clause(0)`) as input. The output of the job (`mrOutput`) is the set of all possible assignments to the variables in the first clause that are supported in the data graph. `boundVars` tracks which query variables have been bound by identifying edges which match the first clause.

For our running example query we introduced above, the SPARQL engine will iterate over the three clauses in the query. The variables `?person` and `?car` are bound and set to `boundVars` during the processing of the first clause (`?person :owns ?car`).

Edge data can be stored in a framework appropriate for the implementation context. For example, when the Clause-Iteration approach is used in context with Hadoop, the data can be stored in HDFS and filepaths can be passed as parameters to the algorithm implementations.

Note that we also assume that some small parameters (like individual query clauses) can be passed directly to the MapReduce jobs used in the Clause-Iteration approach. This assumption is valid for common MapReduce implementations like Hadoop where XML data can be used to pass parameters to the MapReduce jobs [41].

The Iteration algorithm uses the `intermediateClauseMapReduce` MapReduce job to identify which edges in the data graph match the successive query clauses and to join these edges with subgraphs that match previous clauses. Hence the `intermediateClauseMapReduce` MapReduce job is passed a union of the edge data and previous variable assignments identified by previous MapReduce jobs (`mrOutput`) that correspond to subgraph query matchings with the data. `intermediateClauseMapReduce` is also passed as input to the iterated query clauses (`query.clause(1)`) and `comVars`, a list of variable bindings that are common to the current clauses and the current one. The output of the job (`mrOutput`) is the set of all possible assignments to the variables in the first through ith clause that are supported in the data graph. `boundVars` tracks which query variables have been bound by identifying edges which match the first clause in order to determine `comVars`.

During the first iteration for the running example which processes the clause (`?car :a :car`) the `?car` query variable is used to set `curVars`. Consequently `comVars` is also set to `?car` as this variable is in the current clause which is already bound.

After the final iteration, `mrOutput` is set to `mrOutput` and identifies lists of variable assignments which satisfy all the query clauses and are supported by the data graph. This data is used as input to `selectMapReduce` along with an identification of the variables selected for return by the query. This final MapReduce job filters these complete variable bindings to list only sets of variable bindings which are returned by the Iteration Algorithm. In our running example the query processing filters for the query variable `?person.`

We assume without loss of generality that the list of common variables on every iteration is non-empty. If we cannot rearrange the order of the query clauses such that the common variable list is always non-empty, then query can be split into independent sub-queries that can be processed independently for performance reasons.

We also note that implementations of the algorithm could be greatly impacted by the ordering of the clauses. If larger intermediate results are returned by the intermediate MapReduce jobs, then the response of the algorithm will be slower. This topic of performance tuning via query reordering is an area of ongoing research that we discuss below in Section 7.

3.3 Binding Graph Data to Clauses

Before providing algorithms for the various MapReduce jobs used in the Clause-Iteration approach, we first describe how graph data is bound to variables. This algorithm, called the Variable Binding Algorithm, is shown in pseudo-code in Figure 6.

In practice we used a special prefix character (like the SPARQL variable prefix '?') to distinguish variable assignments from graph data when deciding which map operation to run. We attach this prefix to all variable bindings when listing variable bindings. For example, the bindings in Figure 3 for the clause `?person :owns ?car` would be represented as the following line:

```
?person Kurt ?car car0
```

This example line describes variable bindings simply as a list of pairs of variable names (such as `?person`) and literals bound to that variable (such as `Kurt`.) If the Map step sees a ? as the first character in the input text, then the second map operation is run and the first map operation otherwise.

The algorithm is given a list of edges in the format of a subject followed by a list of pairs of predicates and objects, as described above, and a query clause. The algorithm accumulates variable bindings in `bindingSetList`.

The clause can have one or two variables to bind in the subject, object, or both. If the subject of the query clause is a variable, then all subjects of the data edges can be bound to that variable and this possible binding is saved in `subjBinding`. If the query clause subject is not a variable and the query subject literal is not equal to the data subject, then the algorithm returns an empty list because no bindings are feasible.

```
variableBinding(linesOfTriples triples, Clause
   c):
bindingSetList = new List of Sets of Bindings
subjBinding = null
subject = triples.next
if c.subject is variable
subjBinding = new Binding(c.subject,subject)
else
   if subject != c.subject
      return null
while triples.hasNext
   objBinding = null
   bindingSet = new bindingSet(subjBinding)
   predicate = triples.next
   object = triples.next
   if predicate = C0.predicate
      if c.object is variable
         objBinding = new
      Binding(c.object,object)
         bindingSet.add(objBinding)
         bindingSetList.add(bindingSet)
      else
         if object = c.object
            bindingSetList.add(bindingSet)
            return
return bindingSetList
```

Figure 6. Algorithm to Identify Feasible Bindings from Variables in Query Clauses to Nodes in Individual Triple Data

If the clause subject is a variable or the clause subject matches the data subject, the algorithm iterates through the predicate-object pairs in the data to find any possible bindings that match. If the query and data predicates do not match, then the next pair is tested. If the predicates match and the query object is a literal that does not match, then the next pair is tested. If the predicates match and the query object is a literal that matches, then the subject binding is set to the return list and the algorithm terminates. If the predicates match and the query object is a variable, then the object binding is created and added to the return list with the subject binding. After iteration, the binding list is returned by the algorithm.

3.4 Initial MapReduce Operations

Pseudo-code versions of the InitialMap and InitialReduce algorithms used respectively for the map and reduce steps in the InitialMapReduce jobs in the Clause-Iteration approach can be seen in Figure 7.

```
initialMap(<keyIn,valIn> mapIn, Clause clause,
   <valIn,valOut> mapOut):
for lineOfTriples in mapIn.keySet()
   bindings = variableBindnings(lineOfTriples,
      clause)
   for binding in bindings
      mapOut.add(<binding,null>)

initialReduce(<keyIn,valIn> reduceIn, Clause
   clause, <valIn,valOut> reduceOut):
for key in reduceIn.keySet()
   reduceOut.add(<key,null>)
```

Figure 7. Map and Reduce Algorithms for InitialMapReduce Job in Clause-Iteration Approach

Note we treat the primary inputs and outputs for the Map and Reduce steps as being key-value pairs, as used in the Hadoop MapReduce implementation, as for all Map and Reduce steps of MapReduce jobs discussed herein. If, for example, several lines of text are passed to a Map algorithm as input, each line is treated as a key with a null value. In this manner, the `initialMap` algorithm takes several lines of text listing edge data, identifies what variable bindings align with the input clause, and then outputs those bindings as keys with null values assigned.

The reduce step takes those possible bindings and removes any duplicates.

For our running example, the query clause and graph data are mapped to the triple `Kurt owns car0`. This triple is passed through and output by the reduce step.

3.5 Intermediate MapReduce Operations

Pseudo-code versions of the IntermediateMap and IntermediateReduce algorithms used respectively for the map and reduce steps in the IntermediateMapReduce jobs in the Clause-Iteration approach can be seen in Figure 8.

The ith intermediate Map is actually one of two possible map operations selected by the "`if inputLine is tripleData`" line. This test is performed by seeing if an input line has a special leading character that we describe next.

The first of these operations, which runs when the if statement is satisfied, operates over input edge data to identify all alignments of variables in the ith query clause to nodes in the graph data. The output key of this map for edge data is a list of variable bindings which were encountered in previous clauses and the output values are the set variable bindings which have not been encountered in previous clauses. We loop over bindings because each line of triples may have multiple edges that map to the clause being processed.

The second map operation that results from the "else" condition takes lists of previous bindings as input, and rearranges the bindings such that the output key of the second map is a list of variable bindings which were encountered in the current clause. The output values are set to variable bindings which are not encountered in the current clause. Note that for both cases the output keys are always the list of variable bindings which are common to the current clause and previous clauses and the values are variable bindings which are not common to enable a join in the reduce step.

```
intermediateMap(<keyIn,valIn> mapIn, Clause
   clause, List commVars, <valIn,valOut>
   mapOut):
for inputLine in mapIn.keySet()
   if inputLine is tripleData
      bindings =
         variableBindnings(lineOfTriples,
         clause)
      for binding in bindings
        mapOut.add(<binding.in(commVars),
          binding.out(commVars)>)
   else
      mapOut.add(<inputLine.in(commVars),
          inputLine.out(commVars)>)

intermediateReduce(<keyIn,valIn> reduceIn,
   Clause clause, List commVars,
   <valIn,valOut> reduceOut):
for key in reduceIn.keySet()
   newBindings is new List
   oldBindings is new List
   for binding in reduceIn.values(key)
      if binding is currentBinding
         newBindings.add(binding)
      else
         oldBindings.add(binding)
   for newB in newBindings
      for oldB in newBindings
          combinedB = union(key, newB, oldB)
   reduceOut.add(<combinedB,null>)
```

Figure 8. Map and Reduce Algorithms for Initial MapReduce Job in Clause-Iteration Approach

For our running example the first call of the initial Map processes the clause (`?car :a :car`) the `?car` query variable is used to set `curVars`. Consequently `comVars` is also set to `?car` as this variable is in the current clause which is already bound. The map step processes all of the graph triples to output (`car0 :a :car`) as the sole binding match. The "if" conditional map operation outputs `(<(?car car0), null>)` because only the `?car` query variable is bound and no new variables are bound. The "else" conditional map operation outputs `(<(?car car0), (?person, Kurt)>)` because only the `?car` query variable is bound and `person` was previously bound to `Kurt`.

The ith Reduce step effectively runs a join operation over the intermediate results from the ith Map step by iterating over all pairs of output results with the same common variable binding key. If there is a list of common variable bindings that correspond to both the previous clauses and the new clause as determined by the output key of the ith Map step, then the Reduce step combines all variable bindings and outputs a key with the combined list of variable bindings and a value set to null.

For our running example the join operation of the `<(?car car0), null>` and `<(?car car0), (?person, Kurt)>` input to the reduce step results in the combined `<(?car car0 ?person, Kurt), null>)` variable binding output.

This iteration of MapReduce-join continues until all clauses are processed and variables are assigned which satisfy the query clauses. Intermediate results of the query processing can be saved for later reference to speed up the processing of similar subsequent queries.

3.6 Select MapReduce Operations

Pseudo-code versions of the selectMap and selectReduce algorithms used respectively for the map and reduce steps in the SelectMapReduce jobs in the Clause-Iteration approach can be seen in Figure 9.

```
selectMap(<keyIn,valIn> mapIn, List
   selectVars, <valIn,valOut> mapOut):
for varBinding in mapIn.keySet()
   outVars = varBindings.subset(selectVars)
   mapOut.add(<outVars,null>)

selectReduce(<keyIn,valIn> reduceIn, Clause
   clause, <valIn,valOut> reduceOut):
for key in reduceIn.keySet()
   reduceOut.add(<key,null>)
```

Figure 9. Map and Reduce Algorithms for SelectMapReduce Job in Clause-Iteration Approach

In the selectMap function, the bound variables are filtered so that only the select variables are returned. The reduce step removes duplicates.

3.7 Generalizations of the Clause-Iteration Algorithm

Although our discussion of the Clause-Iteration approach to respond to queries on graph data was focused on SPARQL and RDF edge data, the Clause-Iteration approach can be easily generalized to other query languages and data formats.

Many SQL-like query languages (including SPARQL) are composed of multiple query clauses and an identification of variables to return. It would be trivial to extend the Clause-Iteration approach to these alternative SQL-like query languages.

Even though we focused on conjunctive clause bindings where subgraphs also need to satisfy all query clauses, the Clause-Iteration approach extends to disjunctive clause bindings where only some of the clauses need to be satisfied by data subgraphs to be deemed a sufficient match to be returned. This generalization to disjunctive clauses would be performed by modifying the join operation in the intermediate MapReduce jobs such that the join operation is performed disjunctively rather than conjunctively.

The Clause-Iteration approach could also be generalized to handle other data representations such as quad data with SQL-like query languages instead of graph data. The initial and intermediate MapReduce jobs in the Clause-Iteration approach performs iterative variable bindings that are effectively agnostic to the data representation, as long as the lower-level mechanics of the variable binding is generalized.

4. EXPERIMENTATION

We implemented Clause-Iteration approach to SPARQL query processing over RDF data in the SHARD graph-store to evaluate the performance of our general algorithm design. In particular, 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 [9]. Hadoop stores data on compute nodes through the Hadoop Distributed File System (HDFS) and schedules the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. The Cloudera Hadoop implementation also provides an approach to pass query clauses to compute nodes based on XML files to parameterize the MapReduce operations based on the clauses.

Taken together, these parameterizations allowed us to focus our experimentation on the design and implementation of high-level functionality of algorithmic implementation using the MapReduce framework to construct high-performance and highly scalable applications.

The version of SHARD we deployed for evaluation supports basic SPARQL query functionality (without support for prefixes, optional clauses or results ordering) over the N3 form of full RDF data. The unimplemented SPARQL functionality can generally be handled by pre- or post-processing of queries. We do not expect implementations of this extra functionality to substantially detract from the performance exhibited by the Clause-Iteration approach currently implemented in 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 graph-stores. Also, the deployed version of SHARD does not yet take advantage of any potential query caching enabled by our design choices.

4.1 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 [28]. 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 edge data for 6000 universities which is approximately 800 million edges (and several GB of data) to align with scalable performance evaluations of graph stores used in industry [34].

After loading the graph data into the SHARD graph store, we evaluated the performance of SHARD with the query processing algorithm in responding to queries 1, 9 and 14 of LUBM as was done in [34]. 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).

4.2 Performance

SHARD achieved the following query response times for 6000 universities (approx. 800 million edges and several GB of data) 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 that SHARD performance increased with the number of compute nodes, but we found this performance increase to be sub-linear. This sub-linear increase was most likely due to the overhead of coordination with the NameNode in the MapReduce steps.

We compared the performance of SHARD on the LUBM benchmark to the DAMLDB graph-store. DAMLDB is a current monolithic industry-standard triple-store released as the open-source project Parliament [29]. DAMLDB is designed to run on a single server and represents a current widely-used industry standard triple-store. Using DAMLDB coupled with the Sesame [36] Semantic Web framework to aid query processing [34] we observed the following performance for the same queries from the LUBM benchmark:

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

We attempted to compare the query processing performance of SHARD to other triple-stores, but we were unable to load a sufficient amount of data in what we believe to be a reasonable time frame (i.e., within a few hours.) Therefore we did not compare the performance of SHARD to these other triple-stores. A complete list of these other triple stores with insufficient data loading performance can be found in [34]. Note that in [34], the version of DAMLDB we are comparing SHARD to is Sesame+DAMLDB.

Note that 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 graph-store could potentially be used for extremely large datasets (multiple billions of edges) without requiring any specialized hardware, as is required for similar scalability in monolithic graph-stores.

5. SCALABLE CLOUD APPROACHES TO PROCESSING GRAPH DATA

In this section we discuss current design approaches to scalable cloud-based graph-data processing. We decompose our discussion based on whether or not these technologies are primarily influenced by the MapReduce computing paradigm [5]. We make this distinction between MapReduce-based and non-MapReduce approaches primarily because a MapReduce approach greatly affects the design of cloud-based graph data systems, highlighting a dependence on relatively lower-level indexing and data partitioning functionality. Effective indexing and partitioning techniques are traditionally some of the most demanding technical challenges in building scalable distributed information systems and involve tracking the varying location of data over distributed compute nodes.

5.1 Cloud-Based Graph Data Technologies without MapReduce Implementations

Cloud-based graph technologies that do not utilize MapReduce include neo4j [25], VertexDB [40], Clustered TDB [27], BigData [3], FlockDB [32], Hypergraph [13] and InfiniteGraph [14].

Neo4j is probably the most widely known cloud-based graph database. It is designed with a relatively straightforward and reliable indexer to support graph traversals in cloud-based graph data. FlockDB is probably the most used cloud-based technology – it is part an open-source project used at Twitter. FlockDB uses a more traditional approach to data management via a clustered MySQL backend to store adjacency graphs.

VertexDB made some interesting design choices to provide high-performance garbage collection. VertexDB is composed of nodes which are folders of key/value pairs. Keys are stored in lexical ordering. One of the key features of VertexDB is its garbage collection. Keys beginning with an underscore tell the database that the value is a string, otherwise, the value is a pointer to another node. In this way, the database's garbage collector knows how to traverse the nodes. There is a root node that corresponds to the / path that serves as the root reference of garbage collection scan. Each node also tracks it's size (number of keys it contains) so this can be returned quickly.

Most relevant to the Semantic Web community is Clustered TDB. Clustered TDB is an outgrowth of the Jena project that employs distributed indexes to improve analytic performance. Bigdata is another clustered RDF store. It uses dynamically partitioned key-range shards with B+Trees to enable to easier dynamic addition of capacity. HypergraphDB also uses BTree indexing as implemented in BerkeleyDB. InfiniteGraph is a proprietary graph database that uses a peer-to-peer architecture. Its main architectural feature is its local caching of subgraphs.

Outside of the high-level indexing features we overviewed in the survey, another approach that shows promise has recently been demonstrated in OrientDB. OrientDB [26] uses a new indexing algorithm called MVRB-Tree, derived from the Red-Black Tree and from the B+Tree to combines their respective fast insertion and ultra-fast lookup properties.

5.2 MapReduce Based Implementations for Cloud-Based Graph Data Processing

Although MapReduce has been very successful in supporting many kinds of graph data processing [22] there have been fewer approaches to more complex and general graph data processing challenges including support for general graph query languages. Some of these approaches include [12][17][21][24][42][43].

Although most graph stores provide native support for link following, most do not provide more advanced cloud-based approaches for large-scale parallelized data processing unless they utilize some version of the MapReduce compute paradigm. This is most likely due to the concurrency abstraction issues discussed which is addressed in MapReduce frameworks for processing and generating large data sets [5]. 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. There have been a number of recent studies on MapReduce design patterns [22] that are useful for cloud-based graph data management systems.

The MapReduce software framework is useful for highly-scalable cloud-based processing because it is easily parallelizable for execution on large clusters of commodity machines. One of the more popular MapReduce implementations is Hadoop [9]. Hadoop manages data allocation on compute nodes with 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. These services allow for the design and implementation of high-level functionality using the MapReduce framework to construct highly scalable applications.

There are several benefits as well as drawbacks from using MapReduce to design high-performance information systems, irrespective of the specific designs for those information systems. 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. 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 and Reduce steps. The drawbacks of the Hadoop implementation of the MapReduce framework include that only Java implementations 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.

There have been a number of recent approaches to cloud-based graph data management beside SHARD that use MapReduce implementations. Among the most notable are LarKC [20] and WebPIE [42] which both focus on very large-scale reasoning over graph data. An alternative approach to query processing is also provided in [12] that relies on heuristic approaches.

Also related are graph-stores built on top of databases and high-level languages built using Hadoop. For example [16] provides an extension to a popular Semantic Web toolset to us HBase [10], a Hadoop-based database, as a backing graph store. Similarly, [39] discusses an approach to using the Pig [30] query language with Hadoop to process Semantic Web queries. The drawback of these approaches is that they put additional layers of processing between the original query and the source data, generally resulting in degraded performance. Admittedly, [16] does not provide performance measurements. [39] shows performance on the order of minutes for datasets of several megabytes, several order of magnitude slower than the performance of SHARD on datasets several orders of magnitude larger, although with admittedly different queries.

6. DESIGN INSIGHTS

The performance of the SHARD implementation of the Clause-Iteration approach as compared to previous graph-store implementations demonstrates the viability of our approach based on MapReduce. Admittedly, the previous implementations were run on monolithic hardware, but these previous approaches had strict scalability limitations no longer imposed by a cloud-based Clause-Iteration implementation using MapReduce. Because it is based on MapReduce, the Clause-Iteration design is easily distributed across many compute nodes for highly parallel and highly scalable operation. It is also lower-cost as it can run on commodity hardware rather than the high-memory and high-cost servers usually used to run graph-stores with monolithic architectures.

There are a number of areas for improvement in an alternative to the Hadoop implementation of the MapReduce software framework for the easier design of information systems in general and graph-stores in particular. Most notably, MapReduce and consequently our design are biased towards operations over large datasets without the search for individual key-value pairs. This could be improved upon with native indexing capabilities, possibly supported during pre-processing operations by generalizing beyond the Hadoop implementation of MapReduce. These pre-processing operations could also be used to reason over the data, so that a generalized Clause-Iteration approach could correctly respond to queries that require reasoning.

A more advanced modification to support information system design would be an enhanced MapReduce implementation that provides advanced data linking and indexing to more natively support operations on graph data. Instead of having to store lists of data in flat files in an HDFS-like construct, this enhanced software framework could provide a native linked-data construct that pairs data elements with pointers to related data.

As hinted in our survey above, indexing is a distinguishing feature of graph databases. This aspect is driven by the traditional graph database application of entity relationship searching and identification where a graph database should identify linked descendants of entity nodes. In this manner, link look-up performance becomes a key metric for identifying the optimal graph database for large-scale applications. This linked data framework would provide faster localized query processing without requiring exhaustive search of the data set on every query request.

A potential path toward greatly enhancing the current approaches to cloud-based graph processing is to improve graph partitioning. Splitting data across multiple compute nodes opens up the potential for large latencies when querying for connectivity or pattern-matching across this distributed data. Traditionally graph partitioning has been performed using high-level round-robin, hash partitioning or range partitioning techniques. Round robin is a simple distribution of graph edges in a round robin fashion to each compute node. Hash partitioning involves the distribution of edges by applying a hash function to an attribute value. Range partitioning: involves the distribution of edges based on data ranges, similar to collecting edges based on source nodes in SHARD.

Round robin is inefficient if there is a desire to access edges based on attribute values as is commonly done in graph databases since the location of a given edge is unknown. Hash partitioning is less effective for range queries commonly used for pattern matching. Range partitioning may put a disproportionate amount of access and computation load on some servers, thus losing some of the parallelism advantage of cloud computing.

Another technique that shows promise for improved scalability in cloud-based graph processing technologies is to represent sub-graphs with a limited number of arcs as trees compacted into contiguous storage. A compact tree representation can remove the need for individual child pointers by storing all sibling nodes in contiguous memory. Each tree node then needs to only store a pointer to its first child of each child type in order to access all of its children. Although this is a general technique, we have not yet seen it attempted for cloud computing graph processing technologies.

In order to represent a graph as sub-trees, each node will have three possible child types. A tree-node child is within the tree storage itself, and can be represented as an integer displacement in the array of tree nodes. A leaf-node child may be indicated by an index into an array of all leaf nodes of the tree. Finally, a link from one tree, back up to a point within the same tree or off into another tree would be a link node. This representation is sketched in Figure 10.

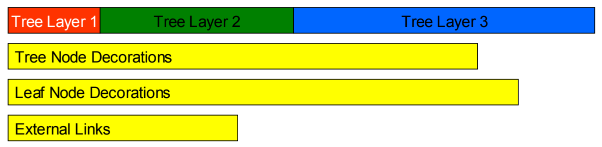

Figure 10: A schematic of subgraph representation.

Each tree node also requires an arc type for each child. Since some of the children of a tree node may reside outside of the tree, the simplest approach is to add the arc type information to the child-decorator information stored inside arrays. In this way, the decorator arrays can be searched linearly for arc types in one MapReduce step. The offsets of particular arc types can be used in a second MapReduce pass to locate connected nodes in the tree-structure array.

In a large graph that supports extremely high query load, disk reads and dram access can be planned according to the same types of arbitration schedules as are used for scheduling packet networks or operating system jobs. Assuming a system that is simultaneously processing hundreds or thousands of simultaneous tree walks for query processing, the number of pending traversals of a particular subgraph represents that subgraph's priority for disk-read, dram caching or even L2 caching in high speed storage close to the processor.

This technique allows a highly scalable system to be built and optimized dynamically on an arbitrary number of workstations. When traversals reach external links connecting one sub-graph to another, that pending traversal work is enqueued and increases the visitation priority of the linked sub-graph. Batching read and traversal access in this way can produce a graph processing system that exhibits maximal performance from data-cache and disk-access patterns.

Various techniques to improve the efficiency of graph data processing for Page-Rank style algorithms are discussed in [22]. These are also relevant to Clause-Iteration methods. Among these are simple approaches to using Combiners between the Map and Reduce steps, and in-Mapper combining to minimize data communications. The Cloud9 software toolbox discussed in [22] may also support enhanced Clause-Iteration performance. Cloud9 enables more ordered clause joining in the Clause-Iteration approach to reduce the needed iteration over joined edges. Cloud 9 may also provides more control over data partitioning so that edges which are likely to be joined in the queries can be placed on the same compute node.

7. ADDITIONAL FUTURE WORK

A trivial extension to the Clause-Iteration approach we are developing is the Subject-Iteration approach. In practice we identified that many queries often have clause subjects with multiple outgoing edges. When coupled with our subject-based graph representation format, rather than requiring a MapReduce job for each edge it would be more efficient to iterate and join over the subjects in a query. Unfortunately, this approach may not always be beneficial due to the extra overhead to perform the join operations. However, depending on the connectedness of the query graph, query graphs with many more edges than nodes would potentially benefit greatly from a Subject-Iteration approach.

We have also been investigating the impact of query ordering on the practical performance of the Clause-Iteration implementations. As alluded to above, query clauses may need to be reordered to ensure that there are common variables to join partial query response on every iteration. Query reordering for improved performance is an ongoing research topic in the general information management community [18], but we have found in practice that minimizing the scale of the join operation has a potentially large impact on the performance of the Clause-Iteration implementation in SHARD. The challenge is in estimating the scale of the join operations due to possible query reordering without actually performing the join operation. A feasible approach may be based on data sampling.

We are also investigating more effective methods to index data. This will most likely need to be supported by a modification to the Hadoop implementation of 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 becomes a deployment concern.)

8. 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] Bigdata Scale-out Architecture. (2010) Retrieved from http://www.bigdata.com/whitepapers/bigdata_whitepaper_10-13-2009_public.pdf

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

[5] 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.

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

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

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

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

[10] HBase (2011). Apache HBase. Retrieved from http://hbase.apache.org/

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

[12] Husain M., McGlothlin J., Masud M., Khan L., and Thuraisingham B. Heuristics Based Query Processing for Large RDF Graphs Using Cloud Computing. To appear in IEEE Transaction on Data and Knowledge Engineering Journal (TKDE).

[13] Hypergraph (2010) Retrieved from http://www.kobrix.com/hgdb.jsp

[14] InfiniteGraph (2010) Retrieved from http://www.infinitegraph.com/

[15] Jena (2010) Retrieved from http://jena.sourceforge.net/

[16] Kantarcioglu M. and Thuraisingham B. (2010) HBase Graph for Jena. Retrieved from http://cs.utdallas.edu/semanticweb/HBase-Extension/hbase-extension.html

[17] 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.

[18] Kolas, D., Query Rewriting for Semantic Web Information Integration, Proceedings of the Information Integration on the Web workshop. 2007.

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

[20] LarKC (2010) Retrieved from http://www.larkc.eu

[21] 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.

[22] Lin J. and Schatz M., Design patterns for efficient graph algorithms in MapReduce. In Proceedings of the Eighth Workshop on Mining and Learning with Graphs. 2010.

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

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

[25] Neo4j (2010) Retrieved from http://neo4j.org/

[26] OrientDB From http://www.orientechnologies.com/

[27] Owens, A., Seaborne, A., Gibbins, N., mc schraefel: Clustered TDB: A clustered triple store for Jena. In: WWW2009. (November 2008)

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

[29] Parliament (2010) Retrieved from http://parliament.semwebcentral.org/

[30] Pig (2011) Apache Pig. Retrieved from http://pig.apache.org/

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

[32] Robey Pointer, Nick Kallen, Ed Ceaser, John Kalucki. Introducing FlockDB. http://engineering.twitter.com/2010/05/introducing-flockdb.html

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

[34] Rohloff K., Dean M., Emmons I., Ryder D., Sumner J.. "An Evaluation of Triple-Store Technologies for Large Data Stores." SSWS '07, Vilamoura, Portugal, Nov 27, 2007.

[35] Rohloff K, Schantz R., High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store. International Workshop on Programming Support Innovations for Emerging Distributed Applications (PSI EtA), 2010.

[36] Sesame (2010). Retrieved from http://www.openrdf.org/

[37] SHARD (2011) Retrieved from http://www.dist-systems.bbn.com/people/krohloff/shard.shtml

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

[39] Sridhar R., Ravindra P. and Anyanwu K. RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web. ISWC 2009.

[40] VertexDB (2010) Retrieved from http://www.dekorte.com/projects/opensource/vertexd

[41] White T., Hadoop: The Definitive Guide. O'Reilly Media, Inc. June 2009.

[42] Urbani J., Kotoulas S., Maassen J., van Harmelen F., and Bal H. OWL reasoning with WebPIE: calculating the closure of 100 billion triples, In Proceedings of the ESWC '10, 2010.

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