SAND2011-8549 C

SAND2011-8549C

Sprinkle SPARQL: Optimizing Semantic Graph Database Queries

Eric L. Goodman, Edward Jimenez
Sandia National Laboratories
Albuquerque, NM 87185

Cliff Joslyn, David Haglin, Bob Adolf, Sinan al-Saffar
Pacific Northwest National Laboratory
Richland, WA 99352

ABSTRACT

We present Sprinkle SPARQL, an algorithm for performing complex SPARQL queries in an efficient, scalable, and graph-oriented manner on a shared-memory machine. The algorithm has two phases. During the first phase, called the *Sprinkle* phase, each triple pattern is processed in isolation. The set of RDF Terms satisfying each triple pattern is stored in hash tables for each variable. From these constrained sets of RDF Terms, we then perform join operations to create the result. The benefits of the preliminary Sprinkle phase are two fold: 1) the number of variable bindings is significantly reduced, minimizing the size and complexity of the expensive join operations, and 2) we obtain enough information to select a near optimal execution plan for the join phase. We evaluate our approach on two data sets: LUBM(8000) and a one billion edge R-MAT graph generated with Graph500[1] parameters and extended to have edge labels.

Keywords

Semantic Web, SPARQL, Query, High-performance Computing

1. INTRODUCTION

In this paper we analyze and evaluate empirically a new algorithm for performing SPARQL Queries[2] called *Sprinkle SPARQL*. An initial version was briefly presented previously [6]. Here, we present extensions that overcome limitations in the prior work. However, our key contribution in this paper is to show that Sprinkle SPARQL fulfills two key desiderata for query engines, namely

1. before execution of a query plan, removing RDF terms[3] from consideration that will not result in valid bindings for the variables in the query, and
2. selection of a near optimal execution path.

[1]http://www.graph500.org/
[2]http://www.w3.org/TR/rdf-sparql-query/
[3]http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/#defn_RDFTerm

To understand each point, consider the graph in Figure 1 and suppose we have the query.

```
SELECT ?X1 ?X2 ?X3
WHERE
   {?X1 p1 ?X2 .
    ?X2 p2 ?X3}
```

By inspection, we see that the optimal path is to first evaluate `?X1 p1 ?X2` followed by `?X2 p2 ?X3` resulting in five total edge examinations. The other order results in eight edges. However, even if we select the optimal path, for a naïve approach, 75% of the intermediate results created after evaluating `?X1 p1 ?X2` are later pruned from the final answer. This wasted computation is easily exacerbated with more complex queries. Sprinkle SPARQL addresses these two issues, using low cost operations we call *Sprinkling* to prune dead-ends and find a near optimal execution plan.

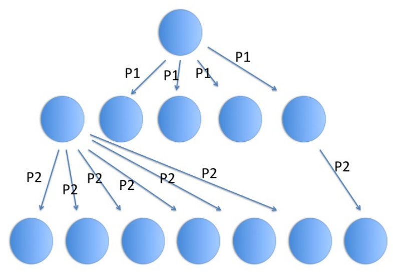


Figure 1: An example graph showing how the ordering of evaluating triple patterns is important. Also shows that even with the best ordering, work is wasted on intermediate results that are later pruned.

2. CONVENTIONS/DEFINITIONS

Data written in RDF can be thought of as a graph, where the subject and object are vertices and the predicate is a directed edge connecting them. As such, we use the following conventions and definitions to facilitate expressing various concepts about the RDF graph.

- V is the set of vertices in the graph. $|V|$ denotes the number of vertices in the graph, or the *order*.
- E is the set of edges. $|E|$ denotes the number of edges, or *size* of the graph.

- T is the set of edge types (predicate types). $|T|$ denotes the number of unique edge types.
- For $p \in T$, $|p|$ denotes the number of edges with edge type p.
- Let $\sim$ be the relation *has the same edge type as*. This relation defines an equivalence relation on the edges. We define max_T as the size of the largest equivalence class under $\sim$.
- We denote the total degree of a vertex v as $\delta(v)$, the number of incoming edges as $\delta^-(v)$, and the number of outgoing edges as $\delta^+(v)$.
- We refer to the maximum indegree as Δ^- and the maximum outdegree as Δ^+. The average degree is $\bar{\delta} = |E|/|V|$.

A *result* is the outcome of a SPARQL query. It has a set of columns, which is the set of selected variables in the SPARQL query, and number of rows, which are solutions of the SPARQL query. We also make use of the term *intermediate result*, which has the same tabular form as the result of a SPARQL query, but is the product of executing a subset of the triple patterns in the basic graph pattern of a query. Intermediate results can contain variables and columns not in the final result as not all variables used in a basic graph pattern need be selected by the query.

We use the standard convention of SPARQL to represent variables with alphanumeric identifiers prepended with a question mark, e.g. $?x1$. However, we also overload the notation in our algorithmic discussions to encode two separate notions: 1) the list of variable bindings in an intermediate or final result, and 2) the set of variable bindings within a hash table created during the Sprinkle phase of our algorithm. We discuss each in more detail below.

For the case when a variable, $?x1$, is referenced in terms of the result of a query, we imply all the variable bindings found within the column of the result for $?x1$. For example, take the query presented in Section 1. The final result (if non-empty) will be a three column structure. Below is an example. For conciseness, we use integers to represent the variable bindings instead of RDF Terms.

$?x1$	$?x2$	$?x3$
1	5	3
1	2	5
$\vdots$	$\vdots$	$\vdots$
2	5	3

So when we refer to $?x1$ in the context of the result, we imply the list of values, 1, 1, ..., 2. Notice that $?x1$ is a list, so values can be repeated.

For the Sprinkle SPARQL algorithm, we create hash tables containing variable bindings that fulfill the constraints specified by the triple patterns. In this instance, $?x1$ is a set of values and not a list.

We also use the $\delta()$ notation on lists and sets of variable bindings. For instance, if $?x$ is a variable, we define $\delta^+(?x)$ to be the summation of all the outdegrees of known variable bindings of $?x$ that are vertices, i.e.

$$\delta^+(?x) = \sum_{v \in ?x \wedge v \in V} \delta^+(v) \qquad (1)$$

We also use the term *triapp*:

Definition For a triple pattern tp with n variables where $1 \leq n \leq 3$, a *triapp* of tp is the application of tp to an intermediate result, R (possibly empty), producing a new intermediate result R', where R' has m additional columns, $m = |\{v|v \text{ a variable} \wedge v \in tp\} - \{v|v \text{ a column in } R\}|$

This paper uses k to refer to the number of triapps for a particular query.

To better understand triapps and their interaction with results and intermediate results, consider the following example basic graph pattern with three triple patterns:

$?x1$	*type*	*Person*
$?x1$	*knows*	$?x2$
$?x1$	*coworker*	$?x2$

If the ordering of evaluation was the same as the presentation above, the first triapp creates a one-column intermediate result containing all the nodes that are people. The second triapp expands the result by one column, where each row represents the fact of that the first element knows the second element of the row. The third triapp constrains the existing two-column result, limiting it to pairs of nodes where the first element both knows and is coworkers with the second element.

For the query algorithms we present, we make the following restriction on possible triapp evaluation orderings: Let X_i be the set of variables in the result after i triapps and let $triapp_i$ be the i^{th} triple application in a given ordering for the query. The intersection of variables $v \in triapp_{i+1}$ with X_i must be non-empty. For example, if we add the triple pattern stating $?x2$ to be a person to the example query:

$?x1$	*type*	*Person*
$?x2$	*type*	*Person*
$?x1$	*knows*	$?x2$
$?x1$	*coworker*	$?x2$

evaluating the first triapp followed by the second would not satisfy the restriction because $?x2$ is not in the intermediate result created after evaluating the first triapp.

3. DATA STRUCTURES

There are two fundamental data structures used by the Sprinkle SPARQL algorithm. These data structures are not specific to Sprinkle SPARQL; they could be used by other search engine strategies. Any data structure used in a search engine must support the following features in order for the engine to achieve efficiency:

1. Given a specific node, walk the incoming (outgoing) neighbors in time proportional to the indegree (outdegree) of the node.
2. Given an edge type, walk the edges of that type in time proportional to the frequency of that edge type.

In both cases—vertex degrees and edge types—it is imperative that a lookup of "size" can be done in constant time. That is, the indegree or outdegree and the number of edges of a certain edge type must be available to the search engine in constant time. Moreover, the "walk" must be allowed to progress over an array (or a specific range of an array) so that parallelism can be exploited. To achieve the first feature we could use two adjacency lists, one for the incoming neighbors and one for the outgoing neighbors. However, to reduce our memory footprint, we use a compressed sparse row representation adapted to support typed, directed edges. We use a multimap to achieve the second feature.

We use a *bidirectional* compressed sparse row (CSR) data structure as depicted in Figure 2. The vertex list is composed of two pointers, one as an entry into the incoming edge list, and the other to the outgoing edge list. In addition, the vertex entries contain a unique integer corresponding to the URI that is the name of the vertex. The two edge lists contain pointers to an external edge information array that hold an integer indicating the edge type along with information about the *source* and *target* of the edge.

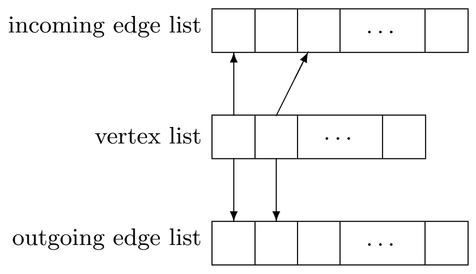


Figure 2: Compressed Sparse Row

This data structure provides the efficient access patterns required by our Sprinkle SPARQL search engine. Walking a neighbor list is essentially picking a range in the incoming or outgoing edge list and simply walking that array. The degrees of each vertex can be quickly computed by observing the distance between adjacent pointers from the vertex list. The size of this data structure is $7|E| + 2|V| + 5$, where the word size is 8 bytes.

Our multimap data structure is similar to a hash table. Hash tables can be thought of as associative arrays, storing key-value pairs. They also provide constant time lookup of a value for a given key. The multimap is the same, except instead of a single value, multiple values are stored. For our purposes, the key is an edge type and the values are the edges that have the given edge type. While very beneficial to our algorithm, this index structure does add to our space requirements. The multimap adds $4.5f \cdot |E|$ words, where the word size is 8 bytes and $1 < f \leq 2$ is a multiplier to obtain a load factor on the hash table conducive to performance.

4. CRAY XMT

For all of our experiments we employed a Cray XMT shared-memory supercomputer. The Cray XMT has a relatively lengthy history of performing well on graph-type problems [2], [16], [5], [13], as well as growing relevance and application to the Semantic Web [7], [6], [14]. As such, we believe it is an ideal platform to perform SPARQL Queries.

The Cray XMT is a unique shared-memory machine with multithreaded processors especially designed to support fine-grained parallelism and perform well despite memory and network latency. Each of the custom-designed compute processors (called *Threadstorm* processors) comes equipped with 128 hardware threads, called *streams* in XMT parlance, and the processor instead of the operating system has responsibility for scheduling the streams. To allow for single-cycle context switching, each stream has a program counter, a status word, eight target registers, and thirty-two general purpose registers. At each instruction cycle, an instruction issued by one stream is moved into the execution pipeline. The large number of streams allows each processor to avoid stalls due to memory requests to a much larger extent than commodity microprocessors. For example, after a processor/ has processed an instruction for one stream, it can cycle through the other streams before returning to the original one, by which time some requests to memory may have completed. Each Threadstorm processor of the machine under study has 8 GB of memory, all of which is globally addressable. The latest XMT2 can go up to 64GB per processor. The system we use in this study has 128 processors and 1 TB of shared memory.

Programming on the XMT consists of writing C/C++ code augmented with non-standard language features including generics, intrinsics, futures, and performance-tuning compiler directives such as pragmas. Generics are a set of functions the Cray XMT compiler supports that operate atomically on scalar values, performing either `read`, `write`, `purge`, `touch`, and `int_fetch_add` operations. Each 8-byte word of memory is associated with a full-empty bit and the read and write operations interact with these bits to provide light-weight synchronization between threads.

For our code, we achieve parallelism through what is called implicit parallelism, where the compiler automatically parallelizes `for` loops, and through the a more direct approach using the `#pragma mta for all streams i of n` construct. The `for all streams` allows programmers to be cognizant of the total number of streams that the runtime has assigned to the loop, as well as providing an iteration index that can be treated as the id of the stream assigned to each iteration.

5. NAÏVE IMPLEMENTATION

For comparision, we implemented a naïve approach for evaluating SPARQL queries. From a high level, the approach is straight-forward:

```
1: function Naïve(bgp, graph, permutation, result)
2:     result ← ∅
3:     for $i \leftarrow 1, length(bgp)$ do
4:         $triapp(graph, bgp[permutation[\mathrm{i}]], result)$
5:     end for
6: end function
```

In essence, we iterate through the triple patterns, blindly following the sequence specified by *permutation*, forming a series of intermediate results.

As the Naïve method requires an ordering, in our experiments we attempt to evaluate all possible permutations, though in some instances we resort to sampling. In this manner, we present the space of possible outcomes absent some mechanism for choosing an ordering. Also, even if the best ordering is chosen through some other approach, we show that not only does Sprinkle SPARQL largely select that ordering, but also reduces the amount of computation required via the Sprinkle preprocessing step.

6. SPRINKLE SPARQL

Here we present the algorithmic details for *Sprinkle SPARQL*. The algorithm performs two phases: a series of *Sprinkle* operations, called the *Sprinkle* phase, followed by a series of triapps over two-variable triple patterns, called the *Join* phase.

Figure 3 outlines a portion of the Sprinkle phase. There are a number of global variables used implicitly throughout the code that we describe below:

- *bgp* The basic graph pattern that we are evaluating.
- *g* The compressed sparse row graph data structure representing the RDF graph.
- *pi* The predicate index with the mapping from edge types to edges with the given edge type.
- *tables* An array of hash tables that store the valid variable bindings for each variable in the query.
- *odegrees* An array the same size as the number of variables in the query that stores the summation of the outdegrees for valid variable bindings.
- *idegrees* Similar to *odegrees* except a summation of indegrees instead.
- *bindings* The number of valid variable bindings for each variable.
- *occurs* Used to keep track of how many times a variable has been seen during processing.
- *seen* Tally on which triples have been processed.

The sprinkle phase iterates through all of the triple patterns of the basic graph pattern; however, the sequence of evaluation is greedily determined by picking the triple pattern with the least amount of work using the follow statistics as a guide:

- If the subject of the triple pattern is constant, the outdegree of the subject.
- If the subject is a variable, the summation of outdegrees for the bindings, found in *odegrees*.
- If the predicate is constant, the predicate index is queried to determine the number of edges with that predicate.
- If the predicate is a variable, the sum total of all edges with a predicate type that is recorded as valid for the variable, found in *bindings*.
- If the object is constant, the indegree of the object.
- If the object is a variable, the sum of all indegrees for all bindings of the variable, found via *idegrees*.

The triple pattern with the smallest statistic for one of its elements across all unseen triple patterns is selected next for evaluation. This selection is performed on line 3. The four elements returned by the call to *select_tp* are *tp*, the triple pattern index within the basic graph pattern, *pos*, signifying which element within the triple pattern had the smallest statistic, *tt*, the type of sprinkle operation to perform, and *m*, the value of the smallest statistic.

The rest of the sprinkle phase is a large switch statement which handles each possible triple pattern motif and what type of Sprinkling is to be performed. For example, Lines 4-20 handle the triple pattern motif of $?s\ p\ o$ where the subject variable has not been seen before. For this case, there is no information about the subject, so the only two options are exploring from the predicate or exploring from the object. For the case when the predicate occurs less frequently than the indegree of the object (lines 7-12), we use the predicate index to explore the edges with the proper type, and every time we find one of the edges with the correct object, we attempt to insert the subject into the variable's hash table. The hash table stores key-value pairs where the keys are RDF Terms and the value is a count of how many times the variable for the table has been encountered from processed triple patterns. Thus, when we attempt to insert a key *s*, it is only added to or updated in the hash table if the value

```
 1: function sprinkle_phase
 2:     for $i \leftarrow 1, length(bgp)$ do
 3:         $tp, pos, tt, m \leftarrow select\_tp()$
 4:         if tt == VCC then
 5:             $v \leftarrow var\_id(tp,$ SUBJECT)
 6:             $create\_table(v, m)$
 7:             if pos == PRED then
 8:                 for all $e \in pi.edges(bgp[tp].pred)$ do
 9:                     if e.object == bgp[tp].object then
10:                         $table[v].insert(e.subject, occurs[v])$
11:                     end if
12:                 end for
13:             else if pos == OBJECT then
14:                 for all $e \in g.in\_edges(bgp[tp].object)$ do
15:                     if e.type == bgp[tp].pred then
16:                         $table[v].insert(e.subject, occurs[v])$
17:                     end if
18:                 end for
19:             end if
20:             $update\_counts(var)$
21:         else if tt == MCC then
22:             $v \leftarrow var\_id(tp,$ SUBJECT)
23:             if pos == SUBJECT then
24:                 $subjects \leftarrow tables[v].get\_keys()$
25:                 for all $s \in subjects$ do
26:                     for all $e \in g.out\_edges(s)$ do
27:                         if e.type == bgp[t].pred $\wedge$
                             e.object == bgp[t].object then
28:                             $table[v].insert(e.subject, occurs[v])$
29:                         end if
30:                     end for
31:                 end for
32:             else if pos == PRED then
33:                 ...                        ▷ Same as VCC case.
34:             else if pos == OBJECT then
35:                 ...                        ▷ Same as VCC case.
36:             end if
37:             $update\_counts(var)$
38:             ...                            ▷ Other cases omitted.
39:         end if
40:     end for
41: end function
```

Figure 3: The Sprinkle Phase

associated with *s* equals $occurs[v]$ (assuming values in the table are initialized to be zero). If on the other hand the indegree of the object is less than the number of edges with the correct type (lines 13-19), we explore all the in-edges of the object. If the edges are of the correct type, we perform the insert function. In general, we use statistics from the compressed sparse row graph and the predicate index to take a greedy path of least resistance.

Also of note are lines 6, where we create a table for a previously unseen variable, and 20, where we update information about the variable bindings stored in the table. If a variable has not been seen before, we create a table on the order of the size of the smallest statistic reported for the triple (m). At the end of each case within the Sprinkle phase, we update counts for the affected variables. This includes the total outdegree, indegree, and number of bindings for all the variable bindings stored within the table.

After the Sprinkle phase, the algorithm switches to the Join phase, which is similar to the Naïve algorithm but with some key differences. Unlike the Naïve approach, Sprinkle SPARQL uses heuristics to select the order of the triapps.

Similar to the Sprinkle phase, the Join phase selects an ordering of evaluation based on statistics of each element of the triple patterns. We use the *idegrees*, *odegrees*, and *bindings* arrays populated during the Sprinkle phase; however, as variables are added to the result, the totals are updated to reflect the contents of the result instead of the hash table contents. Another key difference between the Naïve approach and Sprinkle SPARQL is the join phase of Sprinkle SPARQL does not need to consider the single-variable triple patterns. The variable bindings that arise from evaluating those are already stored in the hash tables for each variable. Also in contrast to the Naïve method, when a variable is added to an intermediate result, the result must be joined with a the contents of the table of the new variable.

7. EVALUATION

We evaluate Sprinkle SPARQL on two data sets, the Lehigh University Benchmark [8] and an R-MAT [4] graph augmented with edge labels.

The Lehigh University Benchmark is a synthetic data set that can be generated to arbitrary sizes. It produces triples related to a university setting, specifying students, faculty, courses, and other entities and the relationships between them. We generated LUBM(8000), where the number in the parentheses denotes the number of universities. The number of triples in the LUBM(8000) is approximately 1.1 billion. However, we materialize triples inferred via minimal RDFS [18] with an inference engine described in [7]. This expands the data set to ∼1.34 billion triples. Finally, we also added the rule stipulating that graduate student is a subclass of student, resulting in a final total of ∼1.35 billion triples. We converted the triples into integers for faster processing and for loading into our data structures.

Using this combination of minimal RDFS with the graduate student rule allows us to accurately answer LUBM standard queries 1-10.

R-MAT, or a Recursive Model for Graph Mining, is an approach for generating graphs that have similar characteristics to real-world graphs such as social networks, the Internet topology, and citation graphs. R-MAT mimics these examples by 1) generating a degree distribution that follows a power law, 2) exhibiting community structures, and 3) having a small diameter. There are six parameters to specify: $|V|$, $\bar{\delta}$, and the partition probabilities, a, b, c, and d, where $a + b + c + d = 1$ (for more information on the parameters, see [4]). We set $|V| = 2^{26}$ and $\bar{\delta} = 16$, resulting in a graph with approximately 2^{30} edges. For the partition parameters, we use what is specified by Graph500, $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$. We then add edge labels to graph, varying $|T|$ from 1000 to 10,000. We assigned edge labels in a uniform random way from T. This procedure of course does not follow real-world distributions. However, it is instructive to show that even in this relatively information-poor environment, Sprinkle SPARQL can continue to prove valuable for a large portion of the query space.

For both data sets, in comparing Sprinkle SPARQL with the Naïve approach, we use two metrics:

- We compare the overall time for an equal number of processors. As the Naïve approach has $n!$ possible execution paths for n triple patterns, we directly compare against the best and worst times. Also, for relatively large n, we plot the distribution to give a notion of how likely each outcome is.
- We also examine the sum of intermediate result sizes:

$$\sum_{i=1}^{n} num\_vars_i \cdot |result_i| \qquad (2)$$

where num_vars_i denotes the number of variables (number of columns) included in the i^{th} intermediate result and $|result_i|$ is the length of the result (number of rows). This gives a qualitative feel for the amount of work that is pruned during the Sprinkle phase.

7.1 LUBM

We do a rough categorization of the LUBM queries in terms of their complexity. Queries 1, 3, 5, 6, and 10 are simple and have only one variable in the basic graph pattern. Queries 4, 7, and 8 are also relatively simple, but do have more than one variable and several triple patterns. Queries 2 and 9 are the most complicated. The times reported below are computation only with no I/O.

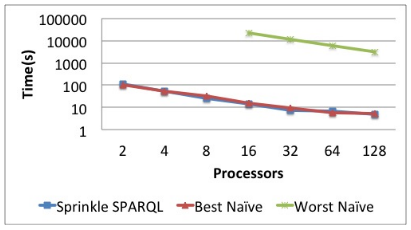
7.1.1 Single Variable LUBM Queries: 1, 3, 5, 6, 10

We first discuss LUBM queries 1, 3, 5, 6, and 10, all of which involve just one variable and no two-variable triple patterns. Some of these queries were designed to test the inferencing capability of SPARQL query engines. However, since we performed inferencing as a preprocessing step and fully materialized the resulting triples, these queries look the same as any other query that do not require inferencing.

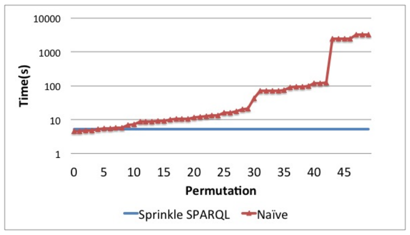
For 1, 3, 5, and 10 we present only the two processor results on the Cray XMT. Each of these queries have two triple patterns, one specifying the type of the variable `?X`, and the other specifying a constraint on `?X`. The trick for good performance on these queries is to select the latter triple, the constraint. Evaluating first the triple specifying the type of `?X` invariably leads to many matches, almost all of which are discarded. Sprinkle SPARQL has no troubles in selecting the constraint triple first, as the degree on the specified object is quite small in all cases. Table 1 compares the times of Sprinkle SPARQL versus Naïve for all the LUBM queries we examined. Table 2 compares the intermediate sizes during the join phase. However, it should be noted that Sprinkle SPARQL does not perform any joins when no two variable triple patterns are present. The result is extracted from the variable's hash table. As such, we report the final result size in Table 2.

Query	Sprinkle SPARQL	Naïve Best	Naïve Worst	Num Procs
1	0.066915	0.067022	45.722397	2
2	5.201755	5.565319	6007.190699	128
3	0.069094	0.066387	61.942575	2
4	0.316535	0.145286	540.875019	2
5	0.029899	0.010239	184.002888	2
6	4.400197	0.562214	0.562214	128
7	0.155892	0.127778	1427.017009	2
8	0.362543	0.192409	483.238118	2
9	13.761691	7.330358	66.298924	128
10	0.078558	0.06876	174.639052	2

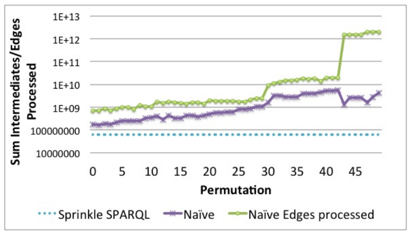
Table 1: This table compares times in seconds of Sprinkle SPARQL versus the best and worst Naïve times. How many processors were used in the runs is also specified.



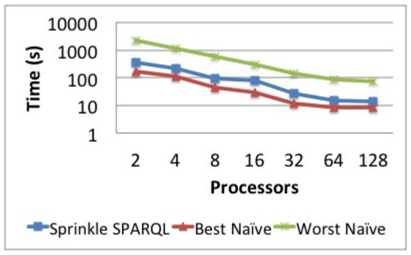
(a) LUBM Query 2



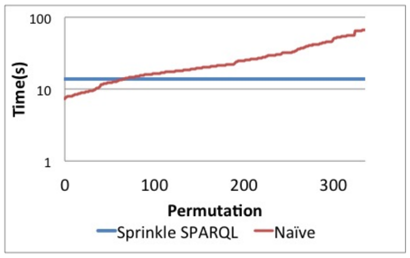
(b) Query 2 Naïve: permutation times



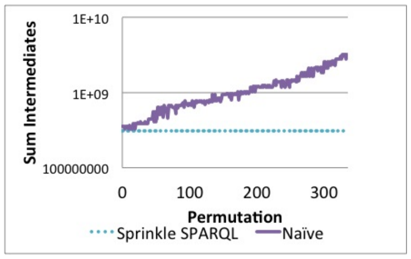
(c) Query 2 Naïve: permutaion sums



(d) LUBM Query 9



(e) Query 9 Naïve: permutation times



(f) Query 9 Naïve: permutation sums

Figure 4: Figure (a) compares Sprinkle SPARQL against the best and worst permutations of the Naïve approach. Figure (b) looks at the range of times the permutations took for 128 processors while Figure (c) examines the sum of intermediate sizes across permutations (and also the number of processed edges). Figure (d)-(f) display similar data for query 9.

Query	Sprinkle SPARQL	Naïve Best	Percent Change	Naïve Worst
1	4	8	100	20,157,123
2	60,798,953	161,272,120	165	5,539,651,344
3	6	12	100	64,478,867
4	306	381	19.7	986,917,868
5	719	1438	100	89,318,851
6	83557706	83557706	0	83,557,706
7	132	406	208	1,383,110,222
8	38950	55640	42.8	965,241,544
9	307234069	326,065,015	6.13	3,233,792,132
10	4	8	100	83557710

Table 2: This table compares the summation of intermediate join sizes of Sprinkle SPARQL versus the best and worst sums via the Naïve method. The *Percent Change* column specifies the percentage change from the sum of Sprinkle SPARQL to the best sum of Naïve.

Query 6 asks for all Students. Since we already performed inferencing, this query largely becomes an I/O task of returning the 83,557,706 students. However, query 6 does reveal a small weakness of Sprinkle SPARQL in comparison to the naïve method. Since there is only one triple pattern, the ordering for both methods is obviously the same, so the information we keep track of during the Sprinkle phase reaps no benefit. Also, we don't prune any results during Sprinkle from updates to variables across multiple triple patterns. Sprinkle SPARQL in essence becomes a large batch insert of all the students into a hash table, and then the very next step is extracting them back out of the hash table. This is naturally going to be much slower than simply grabbing the list of students from the Student node in the graph, and creating a result set. This weakness can easily be averted by adding some special case logic for when only one triple pattern is present, and executing the Naïve method in such a situation.

7.1.2 *Queries 4, 7, 8*

Queries 4, 7, and 8 involve two variables, but are still relatively simple. Table 1 shows that Sprinkle SPARQL results in times that are competitive with the best permutation of Naïve. However, there are many more possible permutations for these queries, and Figure 5 shows the range of times. The times vary over many orders of magnitude.

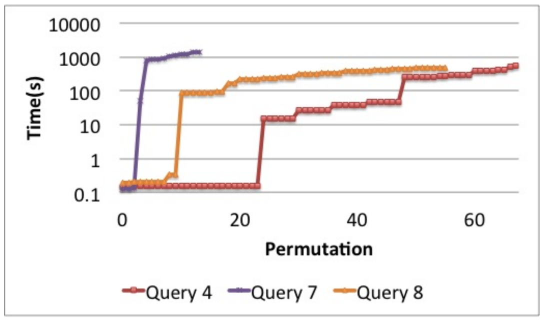


Figure 5: Sorted times for permutations of LUBM Queries 4, 7, and 8 using the Naïve method.

7.1.3 *Queries 2, 9*

Queries 2 and 9 are the more complicated LUBM queries. Figure 6 shows the subgraph of query 2, which is at its core a triangle. Query 9 has a similar structure.

For query 2, the naïve implementation has a large variance, ranging from 5.56 to 6007 seconds. Due to some permutations taking such inordinate amounts of time, we did not run every possible valid permutation. We ran the Naïve approach on 50 different randomly sampled permutations.

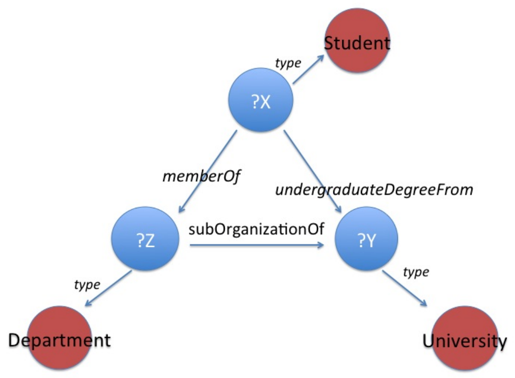


Figure 6: A graphical representation of LUBM Query 2.

The runs with 128 processors, sorted by time, are presented in Figure 4(b). We also present the summation of intermediate sizes for each query in Figure 4(c). However, the determining factor in run time appears to be the number of processed edges. Some orderings amass a large number of high-degree vertices which results in over a trillion edges processed! Based on this sample of 50, we found the best and worst permutations for the naïve implementation and ran between 2 and 128 processors. Sprinkle SPARQL and the best permutation are about even for all processor counts.

For query 9, Sprinkle SPARQL didn't fare as well against the best Naïve permutation as query 2. From Table 2 we see that the difference in intermediate sizes is only off by about 6%. As such, Sprinkle SPARQL couldn't overcome its extra overhead and it took roughly twice the time of the best Naïve permutation. However, Sprinkle SPARQL did select the best ordering.

7.2 R-MAT

As outlined previously, we generated an R-MAT graph with a billion edges and then added edge types in a uniform random way. We must also discuss the type of queries we ran against this data. The below table outlines the 1 and 2 variable triple patterns and the expected number of triples that will match. We excluded from consideration zero-variable triple patterns, which match one triple, and three-variable triple patterns, which match everything when evaluated in isolation.

			Triples Matching
$?s$	p	o	$\bar{\delta}/\lvert T\rvert$
s	$?p$	o	$\bar{\delta}/\lvert V\rvert^2$
s	p	$?o$	$\bar{\delta}/\lvert T\rvert$
$?s$	$?p$	o	$\bar{\delta}$
$?s$	p	$?o$	$\lvert E\rvert/\lvert T\rvert$
s	$?p$	$?o$	$\bar{\delta}$

As expected, the triple patterns with two variables match the most triples. Also, if we make the reasonable assumption that $|T| < |V|$, it follows that $\frac{|E|}{|T|} > \frac{|E|}{|V|} = \bar{\delta}$. Thus, the pattern $?s\ p\ ?o$ is expected to match the most triples and in some respects will be the most complicated. We selected sequences of that pattern to be the focus of our study. More precisely, we evaluated path queries of length k:

$$?x_1\ p_1\ ?x_2\ p_2\ ?x_3\ \dots\ ?x_k\ p_k\ ?x_{k+1}$$

With this setup, we came to the following conclusions:

- Due to how we generated the edge types, the ordering of triapps is largely irrelevant with negligible impact on performance. Thus, the only benefit Sprinkle SPARQL offers is the ability to reduce intermediate result sizes with pruning due to the Sprinkle phase.
- For path queries, there are two general regions, determined by $|T|$, one where the result size increases with increasing numbers of triapps, k, and another where the result size contracts with increasing k.
- We show that Sprinkle SPARQL provides the most benefit to the region where the result sizes contract with increasing k.
- Finally, we present some initial analysis of this graph to better understand the bifurcation we witnessed.

For the rest of this section, we will discuss these four items.

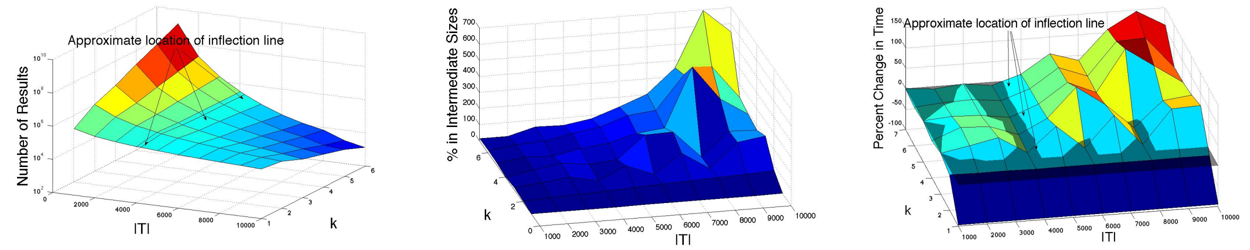
The following table shows the mean and standard deviation of the runtime for performing 10 queries with 5 triapps (16 permutations each) for each of the values $|T|$ listed. The runs were with 64 processers. We can largely conclude that there is no significant difference in the ordering of the triapps. The lack of difference in performance is due to how the edges were assigned. For the first triapp, regardless of the edge type, the expected number of matches is $\frac{|E|}{|T|}$. Then, again for the next triapp, each 2-gram of connected edge types is equally likely due to the independence assumption, and so forth. Thus, ordering does not significantly impact performance and Sprinkle SPARQL loses one of its key advantages: the ability to select an efficient path.

$\lvert T\rvert$	2000	4000	6000	8000	10000
mean	130.6	18.1	7.75	4.68	3.37
stddev	2.81	0.30	0.10	0.084	0.048

We varied $|T|$ from 1000 to 10,000 and k from 1 to 7 and found a curious result. Figure 7(a) shows the results. We found that for small $|T|$, the size of the query result grows with increasing k. For large $|T|$, the size of the query contracts with increasing k. Experimentally, we found the inflection line to be around $|T| = 4210$. We will present later some analysis of why this occurs, but first we will discuss the impact that this line has on the performance of Sprinkle SPARQL relative to the Naïve approach.

Figure 7(b) shows the percentage change in the summation of intermediate result sizes from Sprinkle SPARQL to the Naïve approach. Thus, positive numbers indicate larger intermediate sizes for the Naïve approach relative to Sprinkle SPARQL. As expected, for one triapp, the size is exactly the same for both methods. However, for every other data point, Sprinkle SPARQL reduces the sum. The relative increase ranges from a 4.7% to 696%. The effect is most dramatic for large $|T|$ and large k.

Some of the savings in intermediate sizes translates into improved performance, as shown in Figure 7(c). The figure shows the percent change in time from Sprinkle SPARQL to Naïve. There is a transparent plane marking 0%. Again, similar to LUBM query 6, Sprinkle SPARQL performs significantly worse than the naïve approach for single triapps. However, for $k > 2$, the performance difference ranges from -16.9% to 145%. The region where Sprinkle SPARQL performs the best is where the number of edge types is greater than the inflection line, $|T| > 4210$.



(a) Contraction/Growth of Result Size (b) % Change in Intermediate Sizes (c) % Change in Time

Figure 7: (a) Shows the growth in result size of the query with increasing k for $|T| < 4210$ and contraction for $|T| > 4210$. (b) This shows the percent change in summation of intermediate sizes from Sprinkle SPARQL to Naïve. For all $|T|$ and number of triapps, k, Naïve produces intermedate results that are greater than or equal to what is produced by Sprinkle SPARQL, though the effect is most dramatic with large $|T|$ and large k. (c) Displays the percent change in run time from Naïve to Sprinkle SPARQL. The transparent plane marks 0% change. Sprinkle SPARQL does best in the region $|T| > 4210$ and where $k > 1$.

We now present some analysis of our experimental findings that show a bifurcation of the result sizes for path queries dependent upon $|T|$, where one region has expanding result sizes with increasing k, and another contracts with increasing k. For simplicity in presentation, we will assume the query progresses from the first node, x_1, to the last, x_{k+1}, in sequence. Other query execution plans are possible, but our assumption of uniform random distribution of edge types affords us no loss of generality.

To understand how the query result size grows or contracts with k, we will examine the behavior of how the list of vertices, list of edges, and set of edge types on the frontier of the query change over time. We define $V^{(k)}$ to be the list of vertices in the intermediate result after k triapps within the column for variable x_{k+1}. $E^{(k)}$ is a list of all edges coming from the list of $V^{(k)}$ vertices. Finally, $T^{(k)}$ is a set containing all the edge types of edges in $E^{(k)}$. For $k = 0$, we define $V^{(0)} = V$, $E^{(0)} = E$, and $T^{(0)} = T$. This analysis assume the elements of $T^{(k)}$ are uniformly distributed in $E^{(k)}$. Despite our constructed graph having the degree approximately power law distributed, this analysis makes no assumptions about the degree distribution.

Determining the size of $V^{(1)}$ and $E^{(1)}$ for the first triapp ($?x_1\ p_1\ ?x_2$) is straightforward. Since the elements of T are uniformly distributed in E, we have $|V^{(1)}| \approx \frac{|E|}{|T|}$ To find $|E^{(1)}|$, we use the probability that vertex v in V is the target of a randomly selected edge is: $p(v) = \frac{\delta^-(v)}{\sum_{u \in V} \delta^-(u)} = \frac{\delta^-(v)}{|E|}$ Therefore, the expected total outdegree (i.e. $|E^{(1)}|$) from $|V^{(1)}|$ random vertices is:

$$|E^{(1)}| = |V^{(1)}| \sum_{v \in V} \frac{\delta^-(v)}{|E|} \delta^+(v) \tag{3}$$

$$= \frac{1}{|T|} \sum_{v \in V} \delta^-(v) \delta^+(v) \tag{4}$$

Concerning $|T^{(1)}|$, and in general for $|T^{(k)}|$, we find that the size of the set is unlikely to change. Given $|E^{(k)}|$ attempts drawing from T, the probability that edge type p_i is drawn t_i times for all $i \in \{1, 2, ..., |T|\}$ is given by the multinomial distribution:

$$p(t_1, t_2, ..., t_{|T|}; |E^{(k)}|) = |E^{(k)}|! \prod_{j=1}^{|T|} \frac{\left(\frac{|T|}{|E|}\right)^{t_j}}{t_j!} \tag{5}$$

for $\sum_{j=1}^{|T|} t_j = |E^{(k)}|$. Marginalizing the multinomial distribution above over all random elements not equal to a particular p_i yields the equation

$$\sum_{q=0}^{|E^{(k)}|} p(t_i = q, t_j \geq 0 | j \neq i) = 1 \tag{6}$$

where $\sum_{j=1}^{|T|} t_j = |E^{(k)}|$. The probability that one particular element in T is not drawn in $|E^{(k)}|$ trials is

$$p(t_i = 0, t_j \geq 0 | j \neq i) \tag{7}$$

where $i \in \{1, ..., |T|\}$. The expression above can be expressed as a binomial probability where the first state is $q = 0$ and the second state is $q \neq 0$. Therefore 6 can be rewritten as:

$$\begin{aligned} p(t_i = 0, t_j \geq 0) &= 1 - \sum_{q=1}^{|E^{(k)}|} p(t_i = q, t_j \geq 0 | j \neq i) \\ &= 1 - p(t_i > 0, t_j \geq 0 | j \neq i) \\ &= 1 - \left(1 - \binom{|E^{(k)}|}{0} \left(\frac{1}{|T|}\right)^0 \left(1 - \frac{1}{|T|}\right)^{|E^{(k)}|}\right) \\ &= 1 - \left(1 - \left(1 - \frac{1}{|T|}\right)^{|E^{(k)}|}\right) \\ &\approx 0. \end{aligned}$$

If $|E^{(k)}| \gg 1$, then the innermost parenthesis on the second to last line of the derivation approaches 0. Therefore, it is almost certain that that the entire set T is preserved throughout the intermediate joins.

From equation 4, we can make a rough estimate of where the inflection line is. If $|E^{(0)}| \approx |E^{(1)}|$, then given the assumption that edge types are uniformly distributed in $E^{(k)}$, each triapp should result in approximately the same number of vertices matching. Thus the intermediate results should stay relatively constant. Using 4, the estimated inflection line is 6538, contrasting to what we found empirically to be around 4210.

For the subsequent triple patterns $?x_s\ p_s\ ?x_{s+1}$, $1 < s \leq k$, the probability that vertex v in $V^{(s-1)}$ contains edge type p_s n_v times can be modeled with a binomial distribution where $\frac{1}{|T|}$ is the probability of success and $\delta^+(v)$ are the number of trials:

$$p(n_v) = \binom{\delta^+(v)}{n_v} \left(\frac{1}{|T|}\right)^{n_v} \left(1 - \frac{1}{|T|}\right)^{\delta^+(v)-n_v}. \quad (8)$$

This uses the fact that $|T^{(s-1)}| = |T|$ in most circumstances. Thus, the expected number of times p_s is present for each node is: $\bar{n_v} = \delta^+(v)/|T|$. Since no assumption is made about the distribution of the indegrees or outdegrees other than the summation of the degrees being bounded by $|E^{(s-1)}|$, we generalize by using the expected value of the outdegree: $\frac{\delta^+(V^{(s-1)})}{|V^{(s-1)}|}$, hence

$$\bar{n_v} \approx \frac{\delta^+(V^{(s-1)})}{|V^{(s-1)}||T|}. \quad (9)$$

Therefore, the expected number of vertices in $V^{(s)}$ is

$$|V^{(s)}| \approx \frac{\delta^+(V^{(s-1)})}{|T|} = \frac{|E^{(s-1)}|}{|T|}. \quad (10)$$

Figure 8 lists charts that are the results of experimental runs with the statistics defined above. These charts show the behavior of various graphs with varying number of edge types. It also shows the relative error between $|V^{(s)}|$ and an estimate obtained by equation 10. As predicted, the size of $|T|$ has a significant impact whether or not the result size grows or shrinks as the number of triapps grow.

8. RELATED WORKS

The subject of optimizing database queries is well studied and most techniques are based on identifying and reducing the work through pruning the search space. When a triple store and semantic query engine are implemented on top of a traditional database system with conventional hardware, the easiest method to optimize SPARQL queries is to push as much as possible of the optimization to the underlying database system. For example the work in [23] represents a PHP implementation of the SPARQL standard directly interacting with an underlying RDBMS. Fundamental aspects related to the efficient processing of the SPARQL query language for RDF are examined in [21] which also presents a complete complexity analysis for all operator fragments of the SPARQL query language. [11] proposes the SPARQL query graph model (SQGM). They defined transformations rules to simplify and to rewrite a query. Heuristics were employed to achieve an efficient query execution plan. Query optimization is performed through avoiding join operations, and eliminating redundant or contradicting restrictions. For instance, rules aim at simplifying complexly formulated queries by merging graph patterns. Techniques for optimizing instance retrieval in DL systems are described in [9]. [22] formalized the problem of Basic Graph Pattern (BGP) optimization for SPARQL queries and main memory graph implementations of RDF data. They also analyzed the characteristics of heuristics for selectivity-based static

k	$\lvert T^{(k)}\rvert$	$\lvert V^{(k)}\rvert$	$\lvert E^{(k)}\rvert$	$\frac{\lvert\lvert\hat{V}^{(k)}\rvert-\lvert V^{(k)}\rvert\rvert}{\lvert V^{(k)}\rvert}$
0	1,000	2^{26}	2^{30}	
1	1,000	1,023,200	6,655,566,698	-1.8e-3
2	1,000	6,680,090	22,364,529,263	-3.7e-3
3	1,000	22,428,939	92,198,766,029	-2.9e-3
4	1,000	91,973,501	358,445,924,114	2.4e-3
5	1,000	358,163,896	1,397,187,013,557	7.9e-4

k	$\lvert T^{(k)}\rvert$	$\lvert V^{(k)}\rvert$	$\lvert E^{(k)}\rvert$	$\frac{\lvert\lvert\hat{V}^{(k)}\rvert-\lvert V^{(k)}\rvert\rvert}{\lvert V^{(k)}\rvert}$
0	4,210	2^{26}	2^{30}	
1	4,210	242094	1,573,597,321	2.1e-3
2	4,210	374710	1,223,731,326	-2.5e-3
3	4,210	289136	1,072,606,424	5.3e-3
4	4,210	254519	1,044,346,250	1.0e-3
5	4,210	248195	941,999,621	-5.3e-4
6	4,210	223640	880,275,993	5.0e-4

k	$\lvert T^{(k)}\rvert$	$\lvert V^{(k)}\rvert$	$\lvert E^{(k)}\rvert$	$\frac{\lvert\lvert\hat{V}^{(k)}\rvert-\lvert V^{(k)}\rvert\rvert}{\lvert V^{(k)}\rvert}$
0	10,000	2^{26}	2^{30}	-8.2e-4
1	10,000	102,221	666,609,361	5.0e-2
2	10,000	66,073	240,200,189	8.9e-3
3	10,000	24,363	98,240,256	-1.4e-2
4	10,000	9,601	31,304,486	2.3e-2
5	10,000	3,166	16,665,314	-1.1e-2
6	10,000	1,835	8,758,491	-9.2e-2

Figure 8: This list of tables contains some example experimental data obtained from varying $|T|$ and k. The last column shows the relative errror between $|V^{(k)}|$ and an estimate of $|\hat{V}^{(k)}|$.

BGP optimization. The heuristics included variable counting and selectivity estimation techniques. Summary statistics for RDF data enable the selectivity estimation of joined triple patterns and the development of heuristics. Our own previous work in collecting summary statistics and analysis techniques showed that even when hundreds of thousands of different predicate and node types are present in the common large semantic datasets including those crawled from the web, only a small subset dominates the data [1].

Refer to [20] for a formal study of SPARQL where they examine a fragment without literals and discuss some optimizations procedures. The cost model in [19] is tailored for a heterogeneous grid of SPARQL processors and represents query plans as SPARQL Query Graph Models. Costs rely on recursive cost and cardinality functions. A prototype for SPARQL query optimization based on triple pattern selectivity estimation is built in [3]. That work also demonstrates how triple pattern reordering according to their selectivity affects the query execution performance. [17] identifies query fragments for which minimal query computation is always possible and investigate the complexity of related decision problems while [15] studies the problem of SPARQL query optimization on top of distributed hash tables.

Indexing has not been exclusive to traditional relational database systems as graph databases can also benefit from indexing frequent graph patterns. For example [10] describes optimized index structures for RDF and process and evaluate queries based on that index structure. [12] indexes graph patterns and shows how to find those indexes whose graph-patterns are contained in the query pattern, and derive formulas for estimating index selectivity. They also study the problem of finding optimal sets of indexes for a given query.

9. CONCLUSIONS AND FUTURE WORK

We have shown that Sprinkle SPARQL fulfills the two key desiderata we outlined previously, namely 1) it removes invalid variable bindings with low cost Sprinkle operations before expensive join operations, and 2) it selects a near optimal path for executing the joins. For all of our experimental studies, Sprinkle SPARQL does select the optimal execution plan. While in some cases the best permutation for the Naïve approach fared better than Sprinkle SPARQL, our algorithm presents an efficient method to discover that permutation.

For future work we are interested in the notion of Sprinkling to completion. The Sprinkle Phase currently only processes each triple pattern once. However, it may be useful to keep Sprinkling until there are no new updates to the variable bindings. In this manner, we prune every invalid variable binding. This may pose problems, as the number of possible Sprinkle operations may approach $k!$ where k is the number of triple patterns. If the amount of work dramatically decreases with each Sprinkle, this may still be a viable option, or perhaps the point of diminishing returns can be determined at run time.

We also believe the Sprinkle phase can be used as low cost estimator of query size and complexity. The Sprinkle operations give an estimate of the number of bindings per variable, the question then becomes how large does the joined representation become. Keeping track of the indegree and outdegree distributions for each variable may be sufficient to create an good estimator of query size.

Acknowledgments.

This work was funded under the Center for Adaptive Supercomputing Software – Multithreaded Architectures (CASS-MT) at the Dept. of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

10. REFERENCES

[1] S. al-Saffar, C. Joslyn, and A. Chappell. Structure discovery in large semantic graphs using extant ontological scaling and descriptive semantics. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 1:211–218, 2011.

[2] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proc. of the 2006 Inter. Conference on Parallel Processing*, ICPP '06, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[3] A. Bernstein, C. Kiefer, and M. Stocker. Optarq: A sparql optimization approach based on triple pattern selectivity estimation. Technical report, University of Zurich, Department of, 2007.

[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *SDM*. SIAM, 2004.

[5] G. Chin, A. Marquez, S. Choudhury, and K. Maschhoff. Implementing and evaluating multithreaded triad census algorithms on the cray xmt. In *IPDPS*, pages 1–9, Washington, DC, USA, 2009. IEEE Computer Society.

[6] E. L. Goodman, E. Jimenez, D. Mizell, S. al-Saffar, B. Adolf, and D. Haglin. High-performance computing applied to semantic databases. In *Proceedings of the 8th extended semantic web conference on The semanic web: research and applications - Volume Part II*, ESWC'11, pages 31–45, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] E. L. Goodman and D. Mizell. In-memory rdfs closure on billions of triples. In *Workshop on Scalable Semantic Web Knowledge Base Systems*, November 2010.

[8] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[9] V. Haarslev and R. Moller. Optimization strategies for instance retrieval. In *In Proc. of the International Workshop on Description Logics*, 2002.

[10] A. Harth and S. Decker. Optimized index structures for querying rdf from the web, 2005.

[11] O. Hartig and R. Heese. The sparql query graph model for query optimization. In *In Proc. of the European Semantic Web Conference (ESWC*, pages 564–578, 2007.

[12] R. Heese, U. Leser, B. Quilitz, and C. Rothe. Index support for sparql. In *ESWC*, 2007.

[13] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. *Parallel and Distributed Processing Symposium, International*, 0:1–7, 2010.

[14] C. Joslyn, B. Adolf, S. al-Saffar, J. Feo, E. Goodman, D. Haglin, G. Mackey, and D. Mizell. High performance descriptive semanic analysis of semantic graph databases. In *High-Performance Computing for the Semantic Web (HPCSW)*, 2011.

[15] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. Sparql query optimization on top of dhts. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 418–435, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] K. Madduri, D. Bader, J. Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.

[17] M. Meier, M. Schmidt, F. Wei, and G. Lausen. Semantic query optimization in the presence of types.

[18] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and Efficient Minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, Sept. 2009.

[19] L. Nixon and P. Obermeier. A cost model for querying distributed rdf-repositories with sparql. In *In Proc. of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense Tenerife, Spain, June 2, 2008.*, 2008.

[20] J. PÃľrez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *Proc. 5th International Semantic Web Conference (ISWC06*, pages 30–43. CommunityProjects/LinkingOpenData, 2006.

[21] M. Schmidt, M. Meier, and G. Lausen. G.: Foundations of sparql query optimization. *CoRR*, 2008.

[22] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[23] C. Weiske and S. Auer. Implementing sparql support for relational databases and possible enhancements. In *Proceedings of the first Conference on Social Semantic Web, CSSW.*