ORNL/TM-2015/342

Query Optimization for Graph Analytics on Linked Data Using SPARQL

Seokyong Hong
Sangkeun Lee
Seung-Hwan Lim
Sreenivas R. Sukumar
Ranga R. Vatsavai

07/08/2015

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2015/342

Computational Sciences & Engineering Div (50159093)

Query Optimization for Graph Analytics on Linked Data Using SPARQL

Authors
Seokyong Hong
Sangkeun Lee
Seung-Hwan Lim
Sreenivas R. Sukumar
Ranga Raju Vatsavai

Date Published: 07/08/2015

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-BATTELLE, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

Triplestores that support W3C standards such as Resource Description Framework (RDF) and SPARQL, have emerged as one of the preferred solutions to store and analyze heterogeneous graphs. However, there is a gap in being able to express graph-theoretic operations in SPARQL over the RDF data model. This report describes the process of optimizing performance of the SPARQL-based implementation of such popular graph algorithms by reducing the space-overhead, simplifying iterative complexity, and removing redundant computations by analyzing query plans. Our optimized approach shows significant performance gains on triplestores hosted on stand-alone workstations as well as supercomputers specialized in graph-processing such as the Cray's uRiKA. In this report, we evaluate our optimization with three popular graph operations, counting triangles, finding eccentricity, and testing connectivity over real world graph data.

1. INTRODUCTION

Resource Description Framework (*RDF*) [22] has gained popularity for representing and linking data. The minimal unit of data in RDF is called a *triple*: *subject*, *predicate*, and *object*. In an RDF triple, subject and object can map into a source and destination vertex, respectively. With the predicate representing an edge between two vertices, a set of RDF triples logically denote a graph *G* with vertices and edges. *SPARQL* is the standard language to perform queries for RDF triples. For RDF triples that represent a graph *G*, SPARQL has a set of primitives capable of managing graph data ranging from creating/dropping graphs to inserting/modifying/deleting RDF triples. Using SPARQL, users can describe subgraph-patterns of interest. Hence, database systems for RDF triples, or triplestores are powerful tools for performing graph operations on graph data.

Recently, there is a growing demand for graph-theoretic analysis such as *degree distribution*, *node eccentricity*, *finding triangles*, and *finding connected components*. Addressing the demand, recent research efforts [1, 9] proposed several graph analysis operations for data stored in triplestores. Those operations give a general idea for building graph operations using SPARQL. However, those efforts are restricted in the following ways: (1) delivering only a single non-iterative graph operation that is optimized for directed graphs [1], and (2) focusing only on a generic programming abstraction for building iterative operations on triplestores [9]. The lack of research on systematic way of optimizing SPARQL queries for graph analysis offsets the attractiveness of RDF triplestores for large-scale graph analysis, compared with alternatives such as property graph based graph databases [28] and graph processing systems [14, 16, 27].

This study presents a systematic-way of developing highly optimized graph analysis operations using SPARQL queries, while being cognizant of resulting query plans and underlying SPARQL query processing engines. Our main contributions are: (1) we deliver three graph analysis operations via the SPARQL interface covering both non-iterative and iterative operations, (2) we present our optimization process based on query plan analysis on a popular open-source triplestore, *Apache Jena* [21], and (3) we evaluate those optimized graph analysis operations on a standalone workstation and a Cray XMT supercomputer-based graph analytics platform, *uRiKA* [26].

2. RELATED WORK

Several existing graph processing systems provide a set of graph operations for data analytics [14, 16, 27, 28]. For example, [27] enables users to access its graph operations in python and [14] provides Java-based operations in MapReduce computation model [15]. While supporting a broad range of graph operations, they are based on their own computation platforms and dedicated programming environments rather than triplestores and SPARQL.

In [1], the authors introduced a triangle counting algorithm for triplestores written in SPARQL. The proposed algorithm defines three patterns which represent all possible triangle structures constructed from a set of directed edges. Those patterns efficiently eliminate duplicate triangles, resulting in slow increase of time complexity as graph volumes become larger. However, this work has several limitations. First, the proposed triangle counting operation only works over directed graphs. It cannot process undirected graphs since the representative patterns fail to be defined without edge directions. Secondly, the algorithm represents only non-iterative graph algorithms while many graph algorithms have an iterative nature of computation.

[9] provided a programming abstraction for building efficient iterative graph operations in terms of SPARQL and introduced three iterative graph operations. The abstract programming structure is a useful basis for designing extended graph operations. They focused on a structural view of iterative graph algorithms rather than optimization of individual graph operations. We followed the programming abstraction in order to build iterative graph operations in SPARQL. However, our work is different in that we explored opportunities to optimize individual graph operations ranging from non-iterative to iterative ones rather than developing general programming abstraction for developing iterative graph operations. [20] highlighted the necessity of graph mining algorithms based on linked data and proposed a few initial graph analysis operations for triplestores. However, those operations did not consider resulting query plans and were not optimized with understanding its underlying query processing engine, resulting not comparable performance to alternatives even for moderate size of graphs [19].

3. GRAPH ANALYSIS AND TRIPLESTORES

This section explains graph analysis with three representative graph analysis operations that we focus on. Then, we present a description of triplestores and describe the query processing mechanism of a popular open-source triplestore, *Apache Jena* [21] which is the basis platform for our graph operation optimization and also the query processing engine of Cray's shared-memory graph processing appliance, *uRiKA* [26].

3.1 Graph Analysis and Operations

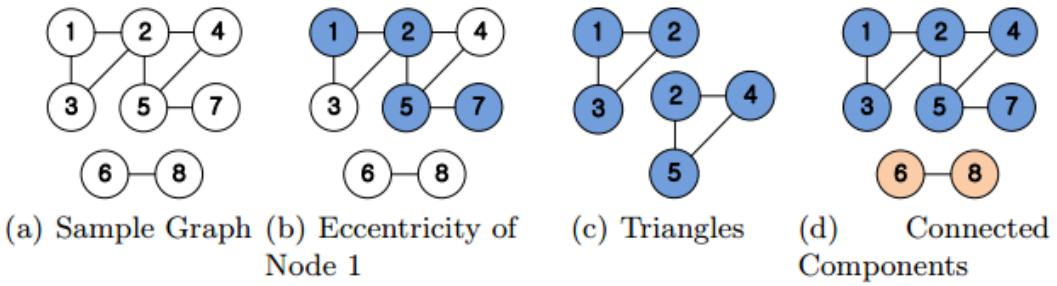


Figure 1. Examples of Graph Operations

Given a graph $G = (V, E)$ where V and E are the sets of nodes and edges, respectively, graph analysis aims at finding obscure properties and patterns from G. It consists of a broad range of analysis operations each of which reveals different aspects of G [1-5]. *Node eccentricity calculation* [2] computes the maximum distance from a node v to any other node in G. For example, in Figure 1(a), the eccentricity of *node 1* is three since the maximum distance from the node is the one to *node 7* (Figure 1(b)). Node eccentricity is an important topological attribute of a graph in that it can be extended to compute other attributes such as radius and diameter of graphs [6, 12]. Another important graph operation is *triangle counting* [1]. This operation finds all the occurrences of any three nodes in G which are connected with edges forming triangles. The sample graph in Figure 1(a) contains two triangles as shown in Figure 1(c). For triangle counting, it is especially critical to design an efficient processing algorithm since it can produce a huge amount of intermediate results during computation [7]. *Connected Components* (CC) is another important graph operation in that many real-world applications necessitate the decompositions of large graphs into connected components [8]. This operation retrieves all subgraphs from G where each node is reachable from any other nodes in a same subgraph. Figure 1(d) shows two connected components in the sample graph one of which consists of *<1, 2, 3, 4, 5, 7>* and the other of *<6, 8>*.

3.2 Triplestores

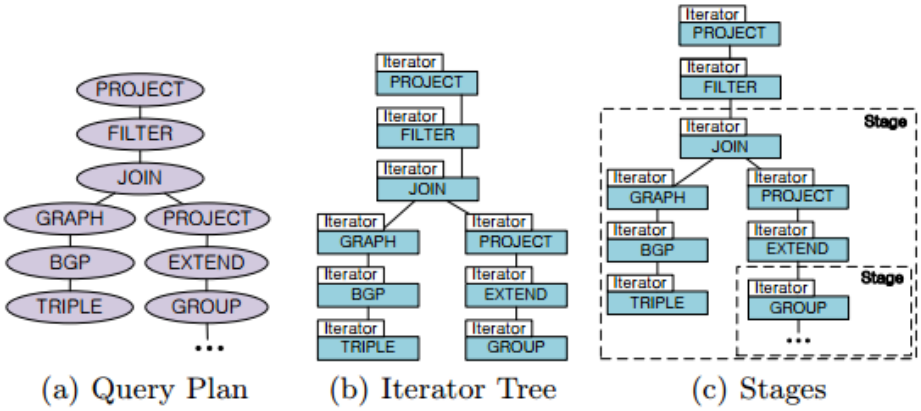


Figure 2. A Sample Query Plan and The Corresponding Iterator Tree

Triplestores [21, 24-26] are special database systems for managing data in *RDF* (*Resource Description Framework*) [22]. In this data model, data are represented as *RDF triples*. A RDF triple consists of *subject*, *predicate*, and *object* where subject denotes a unique resource on the Web and object denotes another resource or an attribute of the resource. Predicate describes a relationship between two resources or an attribute type. By the nature of the RDF data model, triples form directed graphs where predicates are correspondent to directed edges. Triplestores provide a standard query language called *SPARQL* [23]. Users can define interesting patterns and constraints by combinations of provided primitives. SPARQL queries are parsed and transformed into query plans and executed by query execution engines.

Apache Jena [21] is a popular open-source triplestore [1, 10]. *uRiKA*, a Cray's supercomputer-based graph analysis appliance, also employs Jena for its SPARQL query processing engine [26]. Jena executes query plans in an iterator-based manner [11]. In this approach, each operator in a query plan is mapped into a corresponding iterator as shown in Figure 2(a) and Figure 2(b). At query execution time, the query engine calls the root iterator in a query plan and the calling propagates through the iterator tree, returning one record per call. It is notable that some iterators such as ones for GROUP BY and JOIN make the query

plan staged as shown in Figure 2(c). That is, those iterators require all input data to be ready before they are performed. This staged query execution can make a huge impact on the overall query processing performance if a query plan contains several stages which generate excessively large amount of intermediate data. Jena provides a staging-free counterpart operation of JOIN, named *SEQUENCE*. This operation is far more efficient since it returns an intermediate record immediately without waiting for the entire intermediate data are processed and its physical implementation is much more efficient than that of JOIN. It retrieves one record at a time from one child operation tree and iterates the entire data of another child operation tree to find and return matched records. Hence, even if it is more efficient than JOIN, the SEQUENCE operation can become costly when several large child iterator trees exist in a plan or the degree of vertices is high.

4. GRAPH ANALYSIS OPERATIONS IN SPARQL

We designed and implemented three SPARQL-based graph analysis operations: node eccentricity calculation (NE), triangle counting (TC), and connected component computation (CC), which are highly optimized on triplestores. Each operation consists of three steps: (1) an initialization step for performing initial tasks such as creating an intermediate graph and generating an initial state, (2) a computation step, and (3) a finalization step for removing intermediate states and producing output. Each step issues one or more SPARQL queries. When designing each query, our preference is toward generating a query plan which consists of less number of physical operations, in general. In this section, we provide details on the graph analysis operations along with principal guidelines for designing graph operations using SPARQL.

4.1 Graph Representation in RDF Model

As explained in the previous section, directed graphs can be natively represented in RDF data model. For undirected graphs, however, there is an obvious discrepancy between them and the RDF model. In order to overcome such discrepancy, two general approaches are come up with. First, each undirected edge can be stored twice. For example, an edge $N_i \leftrightarrow N_j$ is represented as two triples < N_i <*edge*> N_j > and < N_j <*edge*> N_i > which mean both directions. Hence, undirected graphs require twice as much storage space as directed graphs. The second approach is that undirected edges are stored as if they are directed. When graph operations are performed, their "undirect" properties are reproduced by using a union of two patterns: { *?s* <*edge*> *?o* } UNION { *?o* <edge> *?s* }. Although it imposes an additional computation for such reproduction, the latter approach is preferred in this report to reduce the memory space used for storing graph datasets.

4.2 Node Eccentricity (NE)

The NE operation performs a *breadth-first-search* (BFS) on an input graph which can be considered as a tree rooted at a given node, *N*. At the initialization step, the operation creates a temporary named graph, G_T, which maintains intermediate state and inserts *N* as the starting node. At the computation step, the operation retrieves the neighbors of each node in G_T and stores them into G_T. This process is repeated until the number of nodes in G_T converges. At the finalization step, the intermediate state is discarded by dropping G_T. At computation step, a naive SPARQL query may iterate over every node in G_T and try to insert into G_T nodes that have been already inserted at each iteration. This approach possibly yields unnecessary computations causing performance degradation. In order to avoid such unnecessary computation, we leverage tightly binding triple patterns and filter operations SPARQL provides. First, nodes inserted to G_T at iteration *I* are labeled with $I + 1$ so that only those nodes can be iterated at the next iteration. Second, we minimize intermediate triples passed to INSERT command by using a NOT EXIST filter. The algorithm for our NE operation is shown in Table 1.

Step	Algorithm
Initialization	CREATE GRAPH G_T; INSERT { GRAPH G_T { N < label > 1 }};

Computation	*while*(***convergence_check()*** == false) {
	INSERT { GRAPH G_T { ?neighbor <label> I_{next} }
	WHERE {
	SELECT ?neighbor
	WHERE {
	{ GRAPH G_T { ?node <label> $I_{current}$ }}
	{
	{ ?node ?edge ?neighbor }
	UNION
	{ ?neighbor ?edge ?node }
	}
	FILTER NOT EXIST { GRAPH G_T { ?neighbor <label> ?any }}
	}
	};
	}
	convergence_check() :
	SELECT (COUNT(*) AS ? count)
	WHERE { GRAPH G_T
	{ ?s ?p ?o }
	};
	if (?count == *previous_count*)
	return true ;
	else return false ;

Finalization	DROP GRAPH G_T;

Table 1. Node Eccentricity Algorithm

4.3 Triangle Counting (TC)

Implementing TC operation in SPARQL has a potential problem in that it can produce large intermediate data [7]. Moreover, since we focus on undirected graphs and represent each undirected edge with a single triple, it is unavoidable to union two patterns < *?s ?p ?o* > and < *?o ?p ?s* > for recovering the "undirected" property of edges, producing more intermediate data. In order to reduce such intermediate data, we impose an order on the subject and object of triples so that only triples whose subjects are lexicographically less than the objects are counted. As a result, only one triangle pattern can be retrieved among isomorphic patterns. We accomplish the lexicographic ordering by using two filter conditions: FILTER(STR(?x) < STR(?y)) and FILTER(STR(?y) < STR(?z)). Those two filter conditions sufficiently impose the necessary ordering because STR(?x) < STR(?z) is automatically met by the transitive property of inequality [7]. The resulting initial algorithm for TC operation is shown in Table 2. This algorithm is not iterative and just a single run of the SPARQL query can count the entire triangles in a given dataset. However, this query still produces unnecessary intermediate data during query processing. When we carefully consider the triple patterns in that query, it is observed that each line of UNION performs semantically same computation in conjunction with two filters. The query plan generated from the query is shown in Figure 3(a). The query plan has a SEQUENCE that consists of three children. If STR(?x) < STR(?z) that is implicitly met is added to the third child, the three children conducts the exactly same work wasting computation and memory resources. Moreover, due to the iterator-based query processing approach of Jena, the SEQUENCE operator iterates the entire triples several times for each triple returned from one child operation tree. This can cause a lot of computation. Hence, we optimize the initial algorithm so that such duplicate computation can be avoided. The optimized algorithm is shown at the right-side of Table 2. At initialization step, the optimized TC operation creates a temporary named graph, G_T, and retrieves triples that meet the ordering restriction only once. The intermediate result from the initializing query is stored in G_T. At computation step, the operation finds and counts triangles from the

reduced intermediate triples in G_T. Finally, it removes the intermediate state by removing G_T. The query plans for the optimized algorithms are shown in Figure 3(b).

Step	(A)	(B)

Initialization		CREATE GRAPH G_T;
		INSERT { GRAPH G_T { ?s ?p ?o }}
		WHERE {
		SELECT ?s ?p ?o
		WHERE {
		{ ?s ?p ?o }
		UNION
		{ ?o ?p ?s }
		FILTER(STR(?s) < STR(?o))
		}
		};

Computation	SELECT COUNT (DISTINCT *)
	WHERE {
	{ ?x ?p ?y } UNION { ?y ?p ?x }
	{ ?y ?p ?z } UNION { ?z ?p ?y }
	{ ?z ?p ?x } UNION { ?x ?p ?z }
	FILTER(STR(?x) < STR(?y))
	FILTER(STR(?y) < STR(?z))
	};
		SELECT COUNT (DISTINCT *)
		WHERE {
		{ GRAPH G_T
		{
		?x ?p ?y .
		?y ?p ?z .
		?x ?p ?z .
		}
		}
		};

Finalization		DROP GRAPH G_T;

Table 2. Triangle Counting: (A) Initial Algorithm (B) Optimized Algorithm

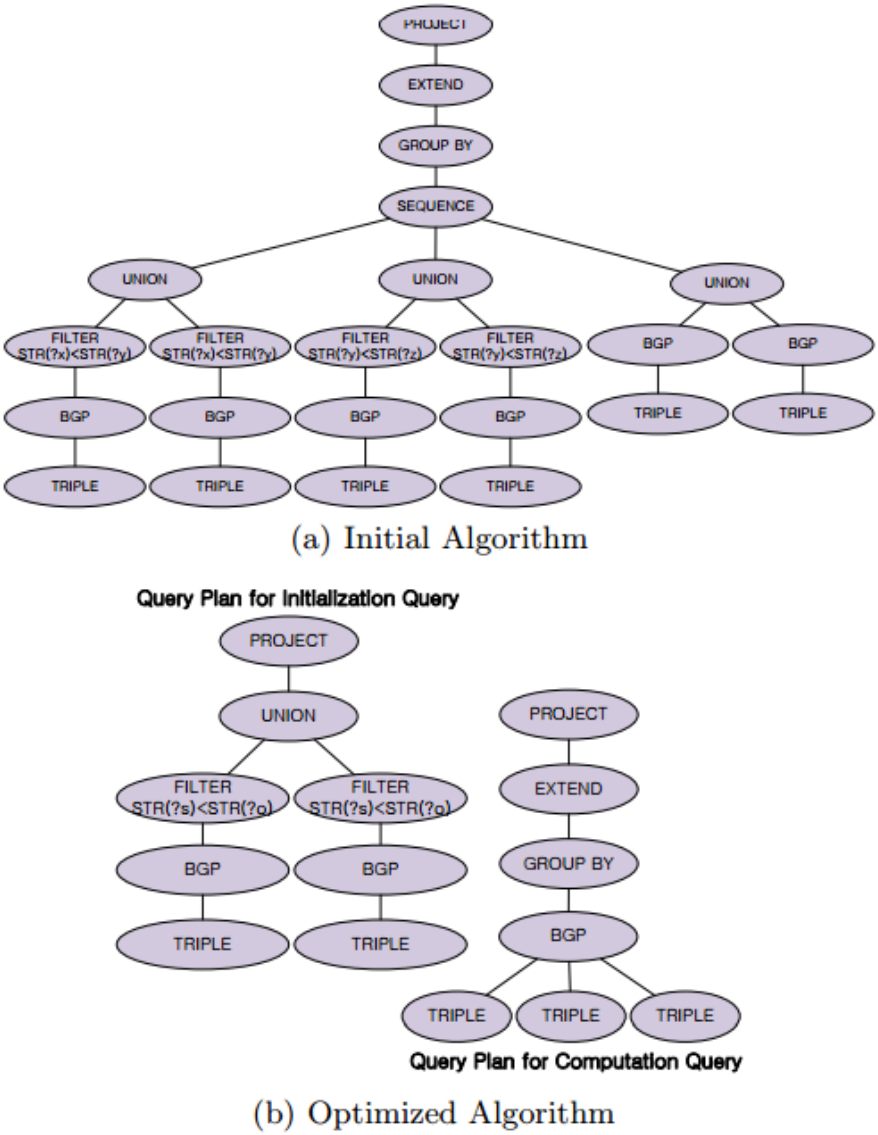


Figure 3. Query Plans for Triangle Counting Algorithms

4.4 Connected Components (CC)

The algorithm for our CC operation resembles *min-label propagation* [13]: at initial time, each node n sets $min\_label(n) = n$ and propagates its $min\_label(n)$ to the neighbors. When n receives min_label* from its neighbors, it updates its $min\_label(n) = min(min\_label*)$ and propagates the new $min\_label(n)$. This process is repeated until all nodes agree on termination. In our SPARQL-based approach, *min_labels* for all N nodes are initialized in a temporary named graph G_T as triples of $< n_i <label> n_i >$ $(1 \le i \le N)$. At computation step, an UPDATE command (DELETE followed by INSERT) retrieves the egocentric network $G_{ego}(n_i)$ centered on n_i where nodes in $G_{ego}(n_i)$ represent min_label*. Then, it calls an aggregation function MIN(min_label*) on $G_{ego}(n_i)$ to calculate a new $min\_label(n_i)'$ and updates the corresponding labeling triple $< n_i <label> min\_label(n_i) >$ in G_T as $< n_i <label> min\_label(n_i)' >$. At the same time, the UPDATE command inserts into G_T a triple of $< n_i <count> 1 >$ for n_i whose $min\_label(n_i)$ is updated. The UPDATE command is repeatedly processed until there is no $< n_i <count> 1 >$ in G_T. After the

computation step completes, G_T is dropped for discarding intermediate state. This initial CC operation is shown in Table 3(A). In order to minimize intermediate data, it exploits filter operation in two ways. The inner filter FILTER(STR(?minimum) > STR(?label)) prevents any label $l \in min\_label^*$ in $G_{ego}(n_i)$ and $l > min\_label(n_i)$ from being passed to the GROUP BY operation. The outer filter FILTER(?original != ?update) avoids operations updating $min\_label(n_i)$ to $min\_label(n_i)'$ if $min\_label(n_i) = min\_label(n_i)'$. Having analyzed the initial CC operation, we observed a notable problem on the underlying query processor in the current release of Jena, which can make a huge impact on the overall query processing performance. The observation is that when a binary JOIN operation has one input from a nested SELECT query with a GROUP BY or ORDER BY operation while the other input comes from a graph, the order of the two input sources represented in a SPARQL query decides the selection of the actual operation for the join. If the nested SELECT block appears before the GRAPH block, then the compiler replaces the JOIN operation with a SEQUENCE operation. Since the SEQUENCE operation does not generate a stage but passes processed record to its upper operation, it is more efficient than a JOIN operation. Figure 4(a) shows the query plan generated from the UPDATE SPARQL query for the initial CC operation. The query plan contains two staging operations (JOIN and GROUP BY) which are expensive as discussed in Section 3.2. In order to accomplish such platform-aware optimization, we switch the nested SELECT and the GRAPH blocks of the query. The query plan of the optimized SPARQL query is shown in Figure 4(b). Compared to the initial query plan, the optimized plan contains just one staging operation (GROUP BY).

Step	(A)	(B)

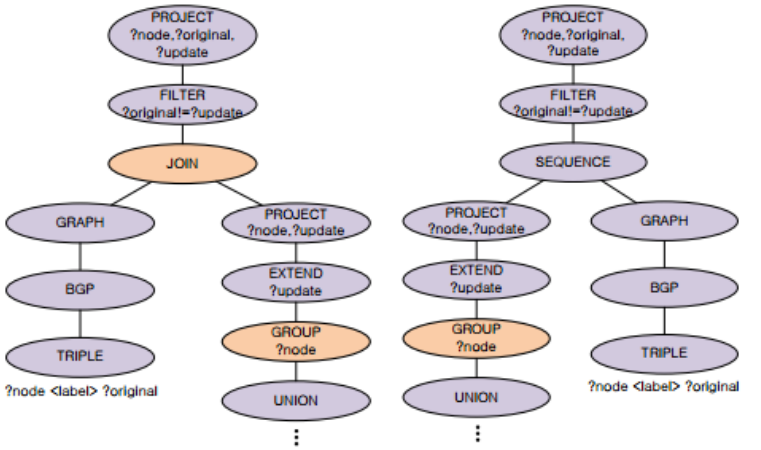
Initialization	CREATE GRAPH G_T;
	INSERT { GRAPH G_T { ?node <label> ?node }}
	WHERE {
	{ ?node ?edge ?o }
	UNION
	{ ?s ?edge ?node }
	};
		Same

Computation	*while*(***convergence_check()*** == false) {
	DELETE { GRAPH G_T { ?s <count> ?o }}
	WHERE {
	{ GRAPH G_T { ?s <count> ?o }}
	};
	DELETE { GRAPH G_T { ?node <label> ?original }
	}
	INSERT { GRAPH G_T
	{ ?node <label> ?update ; <count> 1 }}
	WHERE {
	{ GRAPH G_T .....[**PatternBlockA**]
	{ ?node <label> ?original }
	}
	{ .....[**PatternBlockB**]
	SELECT ?node (MIN(?label) AS ?update)
	WHERE {
	{
	{
	{ GRAPH G_T { ?node <label> ? minimum }
	}
	{ ?node ?edge ?neighbor }
	UNION
	{ ?neighbor ?edge ?node }
	{ GRAPH G_T
	{ ?neighbor <label> ?label }
	}
	FILTER (STR(?minimum) > STR(?label))
	}
	UNION
		PatternBlockA and **PatternBlockB** are reversed.

	{ GRAPH G_T
	{ ?node <label> ?label }
	}
	}
	GROUP BY ?node
	}
	FILTER (?original != ?update)
	};
	}
	
	convergence_check() :
	SELECT (COUNT(*) AS ?changed)
	WHERE {
	{ GRAPH G_T { ?node <count> ?count }}
	};
	if(?changed == 0) return true;
	else return false ;	

Finalization	DROP GRAPH G_T;	Same

Table 3. Connected Components: (A) Initial Algorithm (B) Optimized Algorithm



(a) Initial CC Query Plan (b) Optimized CC Query Plan

Figure 4. Comparison of the Query Plans for CC Operations

5. PERFORMANCE EVALUATION

In this section, we provide information about experiment setup and present and discuss the evaluation results. We evaluated the execution time of our implementation so as to discover the efficiency of the optimizations we performed.

5.1 Experiment Setup

As datasets, we used *Stanford Network Analysis Platform* (*SNAP*) dataset collection [18]. The SNAP dataset collection is provided for general purpose network analysis and graph mining and has more than

50 large graph datasets. We chose seven datasets each of which has different numbers of nodes and edges. Each dataset is stored in an individual file where each line has two node labels that are linked with a directed edge. We converted those graph data into *N-Triples* [29]. Table 4 shows the summary of datasets. For node eccentricity (NE) operation, we used the input nodes shown on the fifth column. We evaluated our operations on a standalone workstation which has an i5-4252U 1.3GHz quad-core processor and 16 GB 1600 MHz DDR3 RAM. We installed two popular open-source triplestores, Apache Jena Fuseki 1.1.1 [21] and RDF4J Sesame 2.8.1 [24] and configured the maximum heap space for JVM to 80% of the total memory. Both triplestores performed in-memory data processing. As a large-scale high performance computing environment, we used uRiKA [26] which is based on Cray XMT supercomputer with 2 TB shared-memory and 8192 hardware threads. We only took processing time into account and did not count data loading time since our query optimization only affects processing time.

Name	Nodes	Edges	Triangles	Input Node to NE
ca-AstroPh(AS)	18,772	198,110	1,351,441	node:2444
com-DBLP(DB)	317,080	1,049,866	2,224,385	node:0
soc-Epinion1(EP)	75,879	508,837	1,624,481	node:0
com-Youtube(YO)	1,134,890	2,987,624	3,056,386	node:1
web-Google(GO)	875,713	5,105,039	13,391,903	node:0
soc-LiveJournal(LI)	4,847,571	68,993,773	285,730,264	node:0
com-Orkut(OR)	3,072,441	117,185,083	627,584,181	node:1

Table 4. Dataset Summary

5.2 Experiment Results

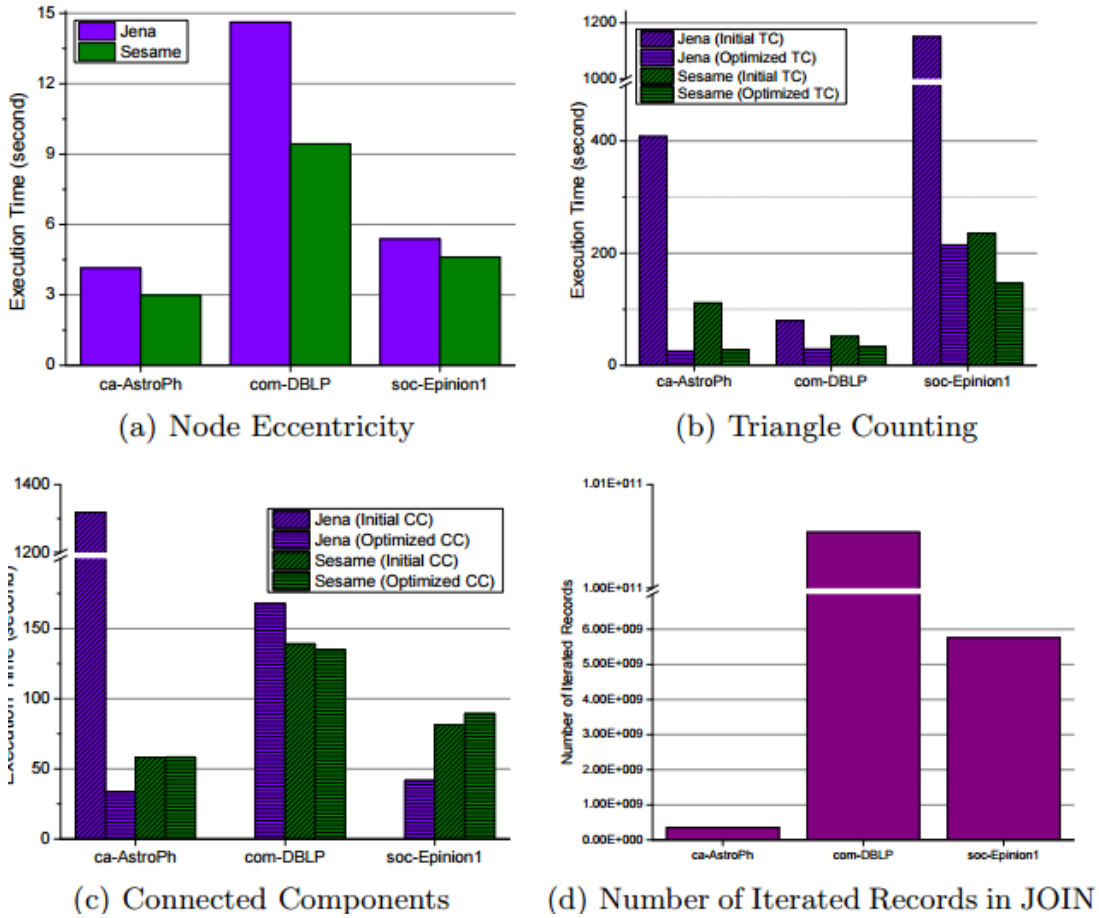


Figure 5. Comparisons of the Operations on Different Triplestores

Stand-alone systems: first, we evaluated our three graph analysis operations on the workstation and measured execution times. Due to the limited processing resources, we used three smallest datasets in this setup: *ca-AstroPh*, *com-DBLP*, and *soc-Epinion1*. Figure 5(a) to Figure 5(c) show the execution times of our graph analysis operations. The NE operation was processed fastest taking less than 15 seconds on both triplestores. While the NE operation is iterative, tightly bounding filter conditions could keep the computation cost of the operation low. The optimization we performed on the initial TC operation could effectively improve performance: 64 - 93% on Jena and 36 - 74% on Sesame. Jena showed 1.5 - 4.8 times worse performance than Sesame over the three datasets when they processed the initial TC operation. However, the optimized TC operation reduced such gap on performance and even showed better performance on Jena with *ca-AstroPh* and *com-DBLP*. For the initial CC operation, Jena did not complete with *com-DBLP* and *soc-Epinion1* within an hour and we stopped its execution while Sesame processed the operation in less than 150 seconds. The optimization on CC, however, could reduce the execution times a lot even resulting in 40% and 53% faster execution times than Sesame on *ca-AstroPh* and *soc-Epinion1*, respectively. We explain our query analysis results and discuss with the optimizations and their impacts on performance on Jena.

Dataset	Operation	Initialization	Computation	Group By	Total
ca-AstroPh	Initial TC	0	52,859,332	10,811,528	63,670,860
	Optimized TC	792,320	4,499,218	1,351,441	6,642,979
com-DBLP	Initial TC	0	25,245,229	2,224,385	27,469,614
	Optimized TC	2,099,732	7,773,591	2,224,385	12,097,708
soc-Epinion1	Initial TC	0	113,557,432	4,326,715	117,884,147
	Optimized TC	1,017,674	23,667,393	1,624,481	26,309,548

Table 5. Numbers of Iterated Records in TC Operations

Comparison of the initial and optimized TC operations: we counted the number of iterated records for each TC operation on Jena. Table 5 shows the iteration numbers. As we discussed in the previous section, the initial TC operation yielded a large iterator subtree rooted at a SEQUENCE (Figure 3(a)). This large iterator tree caused that triples were iterated many times increasing computational complexity. For the three datasets, the initial TC operation iterated around 2 - 10 times more input triples than the optimized operation. Moreover, such large iterator tree produced 2.5 and 8 times more duplicate intermediate data on *soc-Epinion1* and *ca-AstroPh* datasets, respectively. On the other hand, the optimized TC operation avoided duplicate computation and produced a query plan which did not contain a large SEQUENCE subtree as shown in Figure 3(b). The query plan scanned the entire graph only two times to generate two directed edges from each directed edge. The initialization query in the optimized TC also could reduce duplicate intermediate data in advance by enforcing an order on nodes constituting an edge. This could minimize the cost of the expensive GROUP BY operation.

Comparison of the initial and optimized CC operations: as shown in Table 3, the optimized CC operation is exactly same as the initial version except the order of two pattern blocks. On Jena, the resulting query plan of the optimized operation, however, replaces an expensive JOIN operation with a SEQUENCE operation, reducing the number of stages at query execution. Because the CC operations are iterative so that the same SPARQL query is executed multiple times, the accumulated benefit from those multiple executions is tremendous. The initial CC operation performs JOIN in order to combine a previous minimum label and an update minimum label for each node and results in V^2 record iterations. Figure 5(d) shows the total number of records the JOIN iterated. Even if *ca-AstroPh* contains the smallest numbers of nodes and edges, the initial CC operation took about 20 minutes. On *com-DBLP* and *soc-Epinion1* which have more nodes and edges, it did not complete in a few hours. From this analysis, our experience is that users cannot expect that the current Jena query engine always generates optimal query plans and they are

responsible for writing SPARQL queries which generate best query plans. We could not observe such problems on Sesame.

Evaluation of our graph analysis operations on uRiKA: as shown in Figure 6(a), the NE operation could be processed within 100 seconds even with the large graph datasets such as *soc-LiveJournal* and *com-Orkut* which have several millions of nodes and tens of millions of edges. The graph in Figure 6(b) reveals that the TC operation showed drastic increase on the execution times with larger graph datasets. As explained in Section 4.3, this operation can produce a lot of intermediate data during its execution. The query plan of the initial TC has a GROUP BY which is a staging operation as shown in Figure 3(a). Hence, an amount of intermediate data from the SEQUENCE subtree can degrade the performance, even making query executions failed. We observed that the initial TC operation could not run on *soc-LiveJournal* and *com-Orkut* due to such large intermediate data. The CC operation completed its execution in about 10 minutes for all datasets as shown in Figure 6(c). However, the optimization we performed on the CC operation did not show much improvement on its performance on uRiKA.

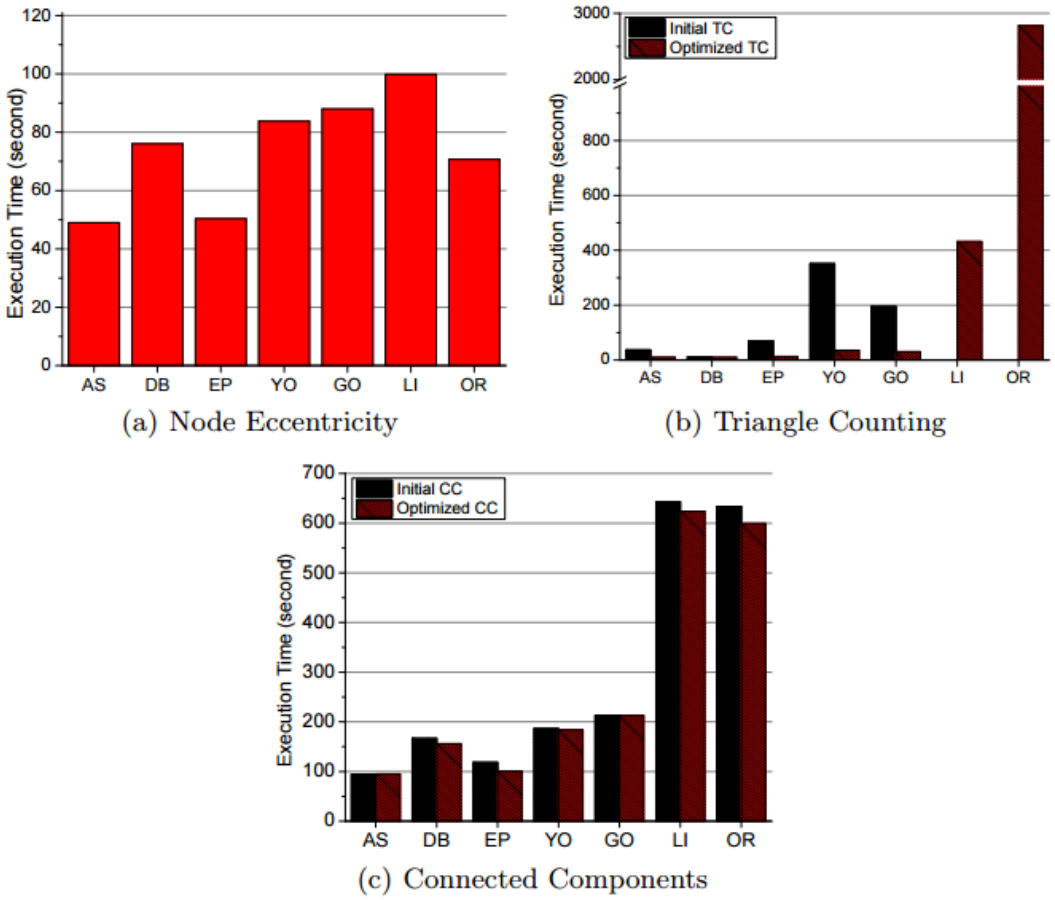


Figure 6. Comparisons of the Operations on uRiKA

6. CONCLUSION

In this report, we extensively studied query plans for most commonly used graph operations on Jena, Sesame, and uRiKA systems and proposed query optimization techniques. Our optimization process

achieved more than 35% performance improvement for triangle counting operation on those triplestores and at least 38 times better performance for connected components operation on Jena. We believe that our optimization work can be also leveraged for building a more extended set of graph analysis operations that can be performed by similar SPARQL queries. We observed that current SPARQL query processing engines do not produce optimal query plans in many cases, and it can cause significant performance degradation. That is, in order to enable efficient graph analysis operations, users are responsible for manually analyzing and optimizing their operations, which can be a burdensome and complicated work for moderate-skilled users. Hence, as a future work, we plan to provide a framework that generates optimized queries in a systematic way for triplestores.

The optimized code has been made available at https://github.com/ssrangan/gm-sparql.

7. ACKNOWLEDGEMENTS

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The research was sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy.

REFERENCES

[1] Jimenez, E., Goodman, E. L.: Triangle Finding: How Graph Theory can Help the Semantic Web. In: Joint Workshop on Scalable and High-Performance Semantic Web Systems, pp. 45–58 (2012)
[2] Hage, P., Harary, F.: Eccentricity and Centrality in Networks. J. Social Networks. 17(1), 57–63 (1995)
[3] Barab´asi, A. L., Albert, R.: Emergence of Scaling in Random Networks. J. Science. 286(5439), 509–512 (1999)
[4] Bloem, R., Gabow, H. N., Somenzi, F.: An Algorithm for Strongly Connected Component Analysis in n log n Symbolic Steps. In: 3rd International Conference on Formal Methods in Computer-Aided Design FMCAD 2000. LNCS, vol. 1954, pp. 56–73. Springer, Austin (2000)
[5] Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University (1999)
[6] Takes, F. W, Kosters, W. A.: Computing the Eccentricity Distribution of Large Graphs. J. Algorithms, pp. 100–118 (2013)
[7] Gubichev, A., Then, M.: Graph Pattern Matching - Do We Have to Reinvent the Wheel? In: 2nd Internation Workshop on Graph Data Management Experiences and Systems, pp. 1–7 (2014)
[8] Najork, M., Fetterly, D., Halverson, A., Kenthapadi, K., Gollapudi, S.: Of Hammers and Nails: An Empirical Comparison of Three Paradigms for Processing Large Graphs. In: 5th International Conference on Web Search and Data Mining WSDM 2012, pp. 103–112 (2012)
[9] Techentin, R. W., Gilbert, B. K. Lugowski, A., Deweese, K., Gilbert, J. R., Dull, E., Hinchey, M., Reinhardt, S. P.: Implementing Iterative Algorithms with SPARQL. In: Proceedings of the Workshops of the EDBT/ICDT 2014 Join Conference, pp. 216–223 (2014)
[10] Morsey, M., Lehmann, J., Auer, S., Ngomo, A. N.: Usage-Centric Benchmarking of RDF Triple Stores. In: AAAI 2012, pp. 2134–2140 (2012)
[11] Hartig, O., Bizer, C., Freytag, J. C.: Executing SPARQL Queries over the Web of Linked Data. In: 8th International Semantic Web Conference ISWC 2009, pp. 293–309 (2009)
[12] Almeida, P. S., Baquero, C., Cunha, A.: Fast Distributed Computation of Distances in Network. In: CDC 2012, pp. 5215–5220 (2012)
[13] Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., Bu, Y.: Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. In: PVLDB 2014. 7(14), 1821–1832 (2014)
[14] Kang, U., Tsourakakis, C. E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System. In: ICDM 2009, pp. 229–238 (2009)
[15] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004, pp. 137–150 (2004) 16 Optimizing Graph Analysis Operations on Linked Data
[16] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework. In: OSDI 2014, pp. 599–613 (2014)
[17] Angles, R., Gutierrez, C.: Querying RDF Data from a Graph Database Perspective. In: ESWC 2005, pp. 346–360 (2005)
[18] Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection, http://snap.stanford.edu/data (2014)
[19] Lim, S., Lee, S., Sukumar, S. R., Ganesh, G., Brown, R. C.: Graph Processing Platforms at Scale. In: Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software. (2015)
[20] Lee, S., Sukumar, S. R., Lim, S.: Graph Mining Meets the Semantic Web. In: ICDE Workshop on Data Engineering meets the Semantic Web. (2015)
[21] Apache Jena, https://jena.apache.org/
[22] Resource Description Framework (RDF), http://www.w3.org/RDF/
[23] SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/
[24] OpenRDF Sesame, http://rdf4j.org/
[25] Virtuoso, http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/

[26] Cray uRiKA-GD, http://www.cray.com/products/analytics/urika-gd
[27] NetworkX, https://networkx.github.io/
[28] Neo4j, http://neo4j.com/
[29] RDF N-Triples, http://www.w3.org/TR/n-triples