From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra

HyeongSik Kim, Padmashree Ravindra, Kemafor Anyanwu
Department of Computer Science, North Carolina State University, Raleigh, NC
{hkim22, pravind2, kogan}@ncsu.edu

ABSTRACT

MapReduce-based data processing platforms offer a promising approach for cost-effective and Web-scale processing of Semantic Web data. However, one major challenge is that this computational paradigm leads to high I/O and communication costs when processing tasks with several join operations typical in SPARQL queries. The goal of this demonstration is to show how a system RAPID+, an extension of Apache Pig, enables more efficient SPARQL query processing on MapReduce using an alternative query algebra called the Nested TripleGroup Algebra (NTGA). The demonstration will offer opportunities for users to explore NTGA-Hadoop query plans for different SPARQL query structures as well as explore relationships between query plans based on relational algebra operators and those using NTGA operators.

1. INTRODUCTION

The amount of Semantic Web data represented using the Resource Description Framework (RDF) is increasing rapidly. For example, the statistics page of Freebase datasets shows that about 337 million triples has been added to its repository in the past 4 years. Also, the statistics for the Linking Open Data community shows an increase of a few billion RDF triples just in the past couple of years. Consequently, an issue of increasing importance is how to enable Web scalable data processing techniques for Semantic Web data.

Parallel data processing techniques based on MapReduce[3] have recently been explored for graph pattern matching[7] and indexing[5] of large-size RDF triple datasets. One challenge that arises when processing SPARQL graph pattern queries using the MapReduce computational paradigm is the large amount of disk I/Os and communication generated. This is due to the fact that the fine-grained modeling of data in RDF requires multiple join operations to assemble related data even for relatively simple tasks. Multiple join operations lead to multiple MapReduce(MR) cycles resulting in a huge amount of intermediate data being materialized and exchanged between the multiple Map and Reduce phases.

Our work [9][10] addresses this problem by proposing an intermediate algebra called the *Nested Triple Group Algebra* (NTGA). NTGA re-interprets subquery structures in a SPARQL query graph pattern in a way that leads to fewer MapReduce cycles during query processing. NTGA also enables the use of a data representation format that allows a more compressed representation of intermediate results which further helps to mitigate I/O costs. This approach has been implemented as an extension to Apache Pig[8] and shows up to 60% performance improvement over processing SPARQL queries using the traditional Pig approach.

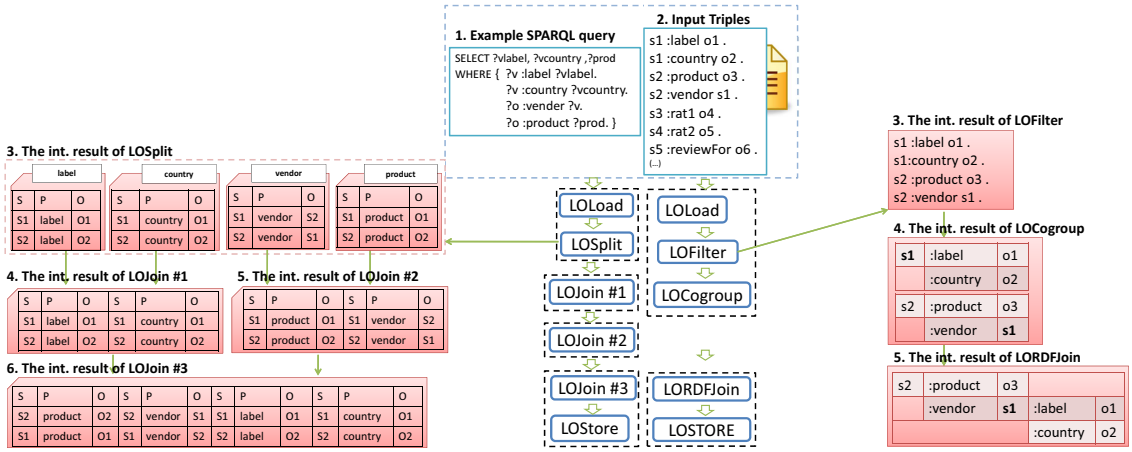
The goal of this proposal is to demonstrate how efficiency in SPARQL query processing is achieved using the NTGA algebra when compared with relational algebra-like operators in systems such as Pig. The remainder of this demonstration proposal is organized as follows: Section 2 provides some relevant background. Section 3 gives an overview of the NTGA data model and operators along with the query optimization strategies presented in [9][10]. It also overviews the architecture of the extended Pig system that we have implemented called RAPID+. Section 4 describes the demonstration scenario along with the sample datasets and queries.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

2. BACKGROUND

An RDF database is a collection of *triples* (*Subject*, *Predicate*, *Object*) where *predicates* are named binary relations between resources or between resources and literal values. For example, the triple {`Vendor1 foaf:homepage <http://www.v1.com>`} asserts that a resource `Vendor1`'s homepage is `http://www.v1.com/`. Its data model can also be viewed as a labeled graph in which resources and literals are nodes labeled with URIs and values respectively, and edges are labeled with *predicate* names. The fundamental querying construct of SPARQL, the standard query language for RDF, is a *graph pattern*, which is essentially a collection of *triple patterns*. Each *triple pattern* is a triple in which *at least one* of the subject, predicate or object is a variable (denoted by a leading ?). The result of a graph pattern query is a list of all variable substitutions that cause a query pattern to match a subgraph in the database. For example, the following triple pattern will match the country where `Vendor1` is located: {`Vendor1 bsbm:country ?country .`}. Generally, the evaluation of a graph pattern is done using a series of join operations to connect triples into subgraph structures that match the query graph pattern.

In MapReduce paradigm, a task is represented in terms of two functions: *Map* and *Reduce*. Using the open source implementation of MapReduce called Hadoop, a join operation in this paradigm can be interpreted in the following manner: Map and Reduce are executed by a set of nodes designated as *"Mappers"* and *"Reducers"* respectively. In the Map phase of the join operation, the tuples are annotated based on the value of the join key. The output of the Map phase is written onto the local disk of the Mappers. Once the Map phase is completed, the Reducers connect to pre-assigned Mappers,



(a) With Pig Latin Operators. **(b)** With NTGA Operators.

Figure 1: Plan and Data Flow.

retrieve their designated intermediate results, and "reduce" them based on the Reduce function. For the join operation, the Reduce phase is responsible for packaging tuples that have matching join values and the resulting joined tuples are written out to the HDFS. If a naive approach is employed in scenarios with multiple join operations, each join is processed in a separate MapReduce cycle. In other words, in each MapReduce cycle, the Mappers materialize their results on disk for consumption by the Reducers in that cycle while the Reducers materialize their results on disk for use by the Map phase of the next MapReduce cycle. Since several join operations are typical in SPARQL graph pattern matching, multiple intermediate results are generated requiring several materialization steps and communication messages. Consequently, proposing techniques to reduce the number of MapReduce cycles is crucial for optimization of query processing performance.

One observation that has been exploited for optimization of SPARQL query processing is that graph patterns often consist of multiple star structures as sub patterns. Pig offers some optimizations that can exploit star sub query patterns - several join operations on the same join key (i.e. star patterns) can be processed in the same MapReduce cycle. This optimization will allow us to reduce the number of MapReduce cycles needed for processing a query to single MapReduce cycle per star subpattern plus cycles for the join operations connecting the star substructures. The result is less I/O costs compared to the "MapReduce cycle per join" approach. The upper section of Figure 1 shows an example SPARQL query based on the Berlin SPARQL benchmark dataset[1] describing *Vendors* and their product *Offers*. It asks to "retrieve the name of *Vendors*, the country where the vendors are located, and their product *Offers*". Processing this query using Pig Latin's query algebra results in the query plan shown in Figure 1a. The logical plan can be described as follows:

1. Load the input dataset using the `LOLoad` operator.
2. Create vertical partitioned relations using the `LOSplit` operator.
3. Join partitioned relations based on join conditions. (In SPARQL, join conditions are implied by repeated occurrence of variables in different triple patterns e.g. *?v* and *?o* in the example SPARQL query of Figure 1). For each star join i.e. join of multiple relations on the same variable, the join will be computed in a single MR cycle such as `LOJoin` #1 and #2 in Figure 1a.4 and Figure 1a.5. Subsequently, the join process is repeated for each star join such as `LOJoin` #3 in Figure 1a.6.
4. Store the final result on disk using the `LOStore` operator.

Though the star-join/MapReduce cycle approach reduces the required number of MapReduce cycles, processing queries with multiple star patterns may still be expensive. [9] also discusses some additional limitations with naive processing of semistructured data such as RDF data using the Pig Latin algebra. An alternative algebra called the Nested TripleGroup Algebra NTGA and data representation format was proposed in [10] to address these limitations and allow for processing RDF data in a more natural way - in terms of "groups of triples" or *TripleGroups* rather than a set of n-tuples like in relational algebra.

Essentially, the functionality of NTGA operators have been refactored differently to allow processing of graph pattern queries using fewer MapReduce cycles. Given our example query, the NTGA based query plan in Figure 1b uses 2 vs. 4 MR cycles in the previous approach. Our empirical evaluation of TripleGroup-based processing showed up to 60% of performance improvement over the approach using the traditional algebra for certain types of graph pattern matching queries.

3. SYSTEM MODEL & ARCHITECTURE

3.1 Nested TripleGroup Data Model and Algebra

A foundational concept in the NTGA data model is that of a *TripleGroup*.

Definition. A *TripleGroup* tg is a relation of triples t_1,t_2,...t_k, whose schema is defined as (S, P, O). Further, any two triples t_i, $t_j \in tg$ have overlapping components i.e. t_i [col_i] = t_j[col_j] where col_i, col_j refer to *subject* or *object* component.

When all triples agree on their *subject* (*object*) values, we call them $Subject$ ($Object$) *TripleGroups* respectively, and they correspond to a star sub graph rooted at the $Subject$ ($Object$) node. To understand the intuition that motivates our TripleGroup based processing approach, observe that performing a `GROUP BY` operation on the subject (S) column of a triple relation (S, P, O) results in a set of groups of triples that have the same value for the subject field i.e. Subject TripleGroups. Most importantly, this operation computes ALL possible star substructures in a single operation executed in a single MapReduce cycle. This implies that the results for ALL star join subquery patterns in a query will be computed in a single

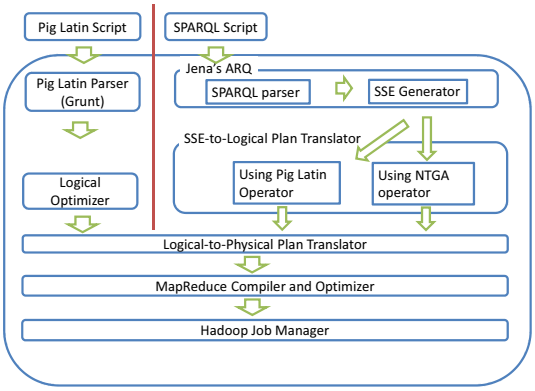


Figure 2: Overall Execution Flow and Architecture.

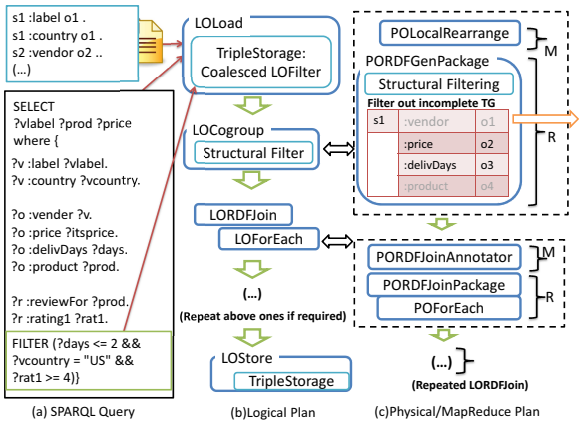


Figure 3: Mappings between Query, Logical/Physical Plan.

MapReduce cycle rather than one MapReduce cycle/star join. This provides the opportunity for optimizing graph pattern query processing. Having computed the results of multiple star-structured subquery patterns, the next issue is manipulating these groups of triples to generate the final query result. This leads to the following operators that comprise the NTGA.

- **`TG_GroupBy`**: this operator groups a triple relation T on either the subject or object columns i.e.,

$$TG\_GroupBy(T, [T.subject \mid T.object])$$

returns a set of TripleGroups TG that agree on their subject/object values i.e. Subject/Object TripleGroups. Figure 1b.4 shows the result of this operator on the example input relation with the subject column specified as grouping column. We assign types to TripleGroups based on the predicates they contain. e.g. $TG_{\{:label,:country\}}$ and $TG_{\{:vendor,:product\}}$ in Figure 1b.4. The subset of TripleGroups that have the same type contains information equivalent to a relation in the relational model.

- **`TG_GroupFilter`**: this operator filters a set of TripleGroups based on structural constraints given in a query. Since the TripleGroups are created by a mere grouping operation, it is not guaranteed that all TripleGroups meet the constraints given in query e.g. may not contain all predicates specified by the graph pattern. The following expression

$$TG\_GroupFilter(TG, QueryPredicateList1, QueryPredicateList2, ..., QueryPredicateListk)$$

returns the set of TripleGroups that each contains all predicates in *ONE* of the Query Predicate Lists. i.e. are structurally complete with respect to *some* query subpattern; e.g. $TG\_GroupFilter(TG, (\{:label, :country\}, \{:product, :vendor\}))$.

- **`TG_Join`**: in many cases, a query will be comprised of multiple star-join subpatterns that are linked together to form a larger graph pattern. The `TG_GroupBy` and `TG_GroupFilter` operators compute answers to all star subpatterns which may need to be "joined" to create the final result. The `TG_Join` operator is defined on a set of TripleGroups and takes in as parameters, the labels of the two types of TripleGroups to be joined and a join condition e.g.

$$TG\_Join(TG_{type1}, TG_{type2}, join\ condition)$$

Figure 1b.5 shows the result of the join expression, $TG\_Join(\{:label, :country\}, \{:vendor, :product\}, TG_{\{:vendor,:product\}}.object = TG_{\{:label,:country\}}.subject)$. The result of a object-subject join operation like our example is a set of Nested TripleGroups which nests the TripleGroups from the right operand in *one* triple from each of the TripleGroups of left operand.

3.2 Architecture and Data Flow in RAPID+

3.2.1 SPARQL-To-Logical Plan Layer

RAPID+ is an extended Apache Pig system that integrates the NTGA data model and algebra. Its extensions include support for expression of graph pattern matching queries either via an integrated SPARQL query interface using Jena's ARQ[6] or using NTGA-related high level commands that have been added to the Pig Latin interface. At the logical and physical layers, NTGA operators are implemented as either extensions of the Pig Latin logical and physical operators or new operators. It also introduces appropriate extensions to query plan generation components to enable generation of logical and physical plan based on NTGA operators. Figure 2 shows the process which is elaborated in the next section.

3.2.2 Data Flow and Logical-to-Physical Plan with NTGA Operators

Assume that we want to process the SPARQL query in Figure 3, which is a slightly more complex version of the previous example query. It seeks to *the list of vendors, their products, and price from the vendors selling products in US within a delivery time of two days and review rating is at least 4.* This query's graph pattern consists of three star-join structures combining descriptions of resources of type *Vendor*, *Offer*, and *Review* denoted by the variables *?v*, *?o* and *?r* respectively, and two chain-join patterns combining these star patterns. This query is expressed using either interface and finally results in an NTGA based logical plan which consists of the following operators:

- **`TripleStorage`**: this loader is a specialized one for RDF data, parameterized with value-based filter conditions. Figure 3.b shows the logical plan which contains `LOLoad` operator loading triples and applying the value-based filter condition such as $?days \leq 2$. This offers some cost savings by avoiding future processing and materialization steps for irrelevant triples to a given query.
- **`LOCogroup`**: the next logical operator is an extended version of Pig Latin's `LOCogroup` that combines the grouping operation specified by `TG_GroupBY` with the `TG_GroupFilter` structure-based filtering operation. The traditional execution plan would execute the grouping step in the reduce phase of a MapReduce cycle, then the groupfiltering phase in a Map cycle of the subsequent MapReduce cycle, thus requiring at least 2 MapReduce cycles. Our extended `LOCogroup` merges both operations into

a single MapReduce cycle. This is achieved at the physical plan level by the introduction of a new Reduce-phase "packaging" operator called `PORDFGenPackage`. This results in reducing the amount of materialized intermediate data and and savings in I/O and communication costs. For example, Figure 3.c shows a TripleGroup containing only the predicates :price and :delivDays being filtered out by the `PORDFGenPackage` operator because it is missing 2 specified predicates, :vendor and :product.

- **`LORDFJoin`**: to support the `TG_Join` between TripleGroups, a new logical operator called `LORDFJoin` is added in RAPID+. This logical operator takes as input a single TripleGroup relation containing Nested TripleGroup and produces a set of Nested TripleGroups. As an example, the plan shown in Figure 3 uses two `LORDFJoin` operators to join the Nested TripleGroups whose *subjects* are *?o* and *?r* and the ones whose *subjects* are *?v* and *?o*.

In general, the TripleGroup based pattern matching for a query with n star sub patterns compiles into a MapReduce flow with n MR cycles (1 cycle to compute all star TripleGroup subquery results and n -1 cycles to join the n star join subqueries), which is half the number of MR cycles (n for each star join subquery + n -1 joins to link the star join subquery results) required to process the same query using the relational style operators in Pig.

4. DEMONSTRATION SCENARIO

The goal of the demonstration is to allow users to explore TripleGroup based processing of SPARQL queries on MapReduce platforms using our Nested TripleGroup Algebra. Users can also compare the performance of query plans obtained by using TripleGroup operators against the relational-style processing in existing systems like Apache Pig. Metrics for I/Os and communication cost such as the number of MapReduce cycles, the size of intermediate results, and the amount of bytes read and written in HDFS among MapReduce phases, etc, for the two types of query plans will be shown to users to help them understand the impact of the NTGA operators query processing performance. The demonstration will use a remote 5-node cluster hosted on NCSU's VCL[2]. In case of limited connectivity in the demo room, we will alternatively use a local virtual machine cluster. This section provides details about the queries and data set that will be used in our demonstration, followed by a brief description of the demonstration scenarios.

4.1 Data Set and Queries

The demonstration will use one synthetic and one real-world data set. The BSBM is a synthetic benchmark dataset generator used for evaluation of SPARQL query processing systems. Its dataset contains information about vendors, the products that they offer, and the reviews of these products. YAGO[4] is a real-world knowledge base that is derived from Wikipedia and contains various entities like persons, organizations etc. In this demonstration, we will use pattern matching SPARQL queries with varying sub structures. The queries will include *Graph Patterns* with two or more star sub structures and *Filtered Graph Patterns* which are graph patterns that also include filter constructs.

4.2 Description

In our demonstration, users will be given a list of sample queries to choose from and based on their selection, an NTGA based query plan will be generated. Users may then be allowed to make modifications to NTGA query plans and see its impact on the cost of query processing. For a given NTGA query plan, a user can request to replace an NTGA operator with an "equivalent" relational query

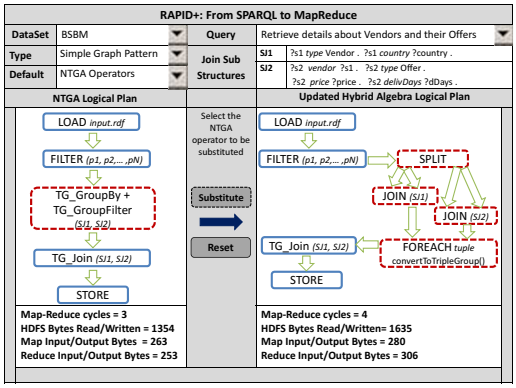


Figure 4: Comparative Exploration of Relational and TripleGroup Algebra Query Plans.

operator or set of operators. The hybrid query plan will be generated with operators from the two algebras and the differences in costs will be shown. Figure 4 shows an example demonstration scenario where the user decides to compute the star patterns using the relational style `JOIN` operator instead of the NTGA `TG_GroupBy` + `TG_GroupFilter` operators. The right hand side of Figure 4 shows the updated hybrid logical query plan after the substitution. This task will help users gain better insights into the costs of each operator viz a viz relational algebra operators as well as the relationship between the two query algebras.

5. ACKNOWLEDGMENTS

This work was partially funded by NSF grant IIS-0915865.

6. REFERENCES

[1] Berlin SPARQL benchmark. http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec.

[2] Virtual Computing Laboratory (VCL). http://vcl.ncsu.edu.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, pages 107–113, 2004.

[4] J. Hoffart, F. Suchanek, K. Berberich, E. Kelham, G. de Melo, G. Weikum, F. Suchanek, G. Kasneci, M. Ramanath, and A. Pease. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Commun. ACM*, 52(4):56–64, 2009.

[5] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. CLOUD, pages 1–10, 2010.

[6] B. McBride. Jena: a Semantic Web Toolkit. *Internet Computing, IEEE*, 6(6):55–59, 2002.

[7] A. Newman, Y.-F. Li, and J. Hunter. Scalable Semantics - The Silver Lining of Cloud Computing. eScience, pages 111–118, 2008.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. SIGMOD, pages 1099–1110, 2008.

[9] P. Ravindra, V. V. Deshpande, and K. Anyanwu. Towards Scalable RDF Graph Analytics on MapReduce. MDAC, pages 5:1–5:6, 2010.

[10] P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *The Semanic Web: Research and Applications*, LNCS, pages 46–61. 2011.