Optimizing RDF(S) Queries on Cloud Platforms

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

ABSTRACT

Scalable processing of Semantic Web queries has become a critical need given the rapid upward trend in availability of Semantic Web data. The MapReduce paradigm is emerging as a platform of choice for large scale data processing and analytics due to its ease of use, cost effectiveness, and potential for unlimited scaling. Processing queries on Semantic Web triple models is a challenge on the mainstream MapReduce platform called Apache Hadoop, and its extensions such as Pig and Hive. This is because such queries require numerous joins which leads to lengthy and expensive MapReduce workflows. Further, in this paradigm, cloud resources are acquired on demand and the traditional join optimization machinery such as statistics and indexes are often absent or not easily supported.

In this demonstration, we will present RAPID+, an extended Apache Pig system that uses an algebraic approach for optimizing queries on RDF data models including queries involving inferencing. The basic idea is that by using logical and physical operators that are more natural to MapReduce processing, we can reinterpret such queries in a way that leads to more concise execution workflows and small intermediate data footprints that minimize disk I/Os and network transfer overhead. RAPID+ evaluates queries using the *Nested TripleGroup Data Model and Algebra* (NTGA). The demo will show a comparative evaluation of NTGA query plans vs. relational algebra-like query plans used by Apache Pig and Hive.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query Processing

Keywords

RDF(S); SPARQL; MapReduce; Hadoop

1. INTRODUCTION AND MOTIVATION

In recent years, large scale data processing platforms based on the MapReduce [1] framework have been widely adopted for scalable processing of (semi) structured and unstructured data on the Web. For example, large production MapReduce clusters with as many as 10K nodes using hundreds PB of storage are being employed in Web companies such as Yahoo! and Facebook. They are used to process the data crawled from the Web[1] and the data produced from 500 million users[2], respectively. Publicly available Semantic Web data is showing similar growth rates with other data in the web. For example, the data represented using the foundational data model for the Semantic Web called RDF[3] is now over several billions of triples[4] (the basic unit of RDF) on the Web. Thus, it is worthwhile exploring how MapReduce-based systems can be leveraged for scalable processing.

In MapReduce, data processing tasks (or queries) are encoded in terms of a sequence of *Map* and *Reduce* function pairs (or MR workflows). In the extended MapReduce systems such as Hive[5] and Pig[6], a high-level data primitive (e.g., relational join operation) is mapped into a single MapReduce cycle. Each MapReduce job involves high disk and network I/O costs for sorting and transferring intermediate data between Map and Reduce functions. Therefore, for the queries requiring multiple MR jobs, processing costs compound across MR workflows. RDF is a fine-grained model representing relationships as binary relations, and querying against RDF usually requires multiple join operations to reassemble related data. Therefore, processing RDF queries on MapReduce often leads to lengthy MR workflows. Such flows pose limitations on how operations can be grouped into fewer MapReduce cycles i.e. shorter workflows, and the case of inference-based queries such as those that use RDF(S)[7] entailment is even more challenging since rewriting such queries for execution usually yields even more complex queries, which forms the multiple unions of conjunctive queries. Another important issue is that MapReduce-based systems generally lack the traditional join optimization machinery like statistics and indexes. Further, it is not clear how to support such techniques without significant overhead given the large pre-processing times which may lead to financial overhead in rented cluster scenarios like using Amazon cloud services. Also, distributed Hadoop-based indexing techniques like HBase[8] do not currently support joins efficiently.

In this demonstration, we will present an extended Apache Pig system called RAPID+[2], that uses an alternative algebraic framework for optimizing queries on RDF data models including inference-based queries to best suit the nuances of MapReduce processing. The basic idea is that by using logical and physical operators that are more natural to MapReduce processing, we can reinterpret such queries in a way that leads to more concise execution workflows and small intermediate data footprints which minimizes overall costs

[1] Yahoo Developer Network: http://developer.yahoo.com/blogs/hadoop

WWW 2013 Companion, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2038-2/13/05.

[2] Facebook Engineering Note: http://www.facebook.com/Engineering/notes
[3] Resource Description Framework: http://www.w3.org/TR/rdf-concepts
[4] Statistics on the Linked Data: http://stats.lod2.eu
[5] Apache Hive: http://hive.apache.org
[6] Apache Pig: http://pig.apache.org
[7] RDF Semantics: http://www.w3.org/TR/rdf-mt
[8] Apache Hbase: http://hadoop.apache.org/hbase

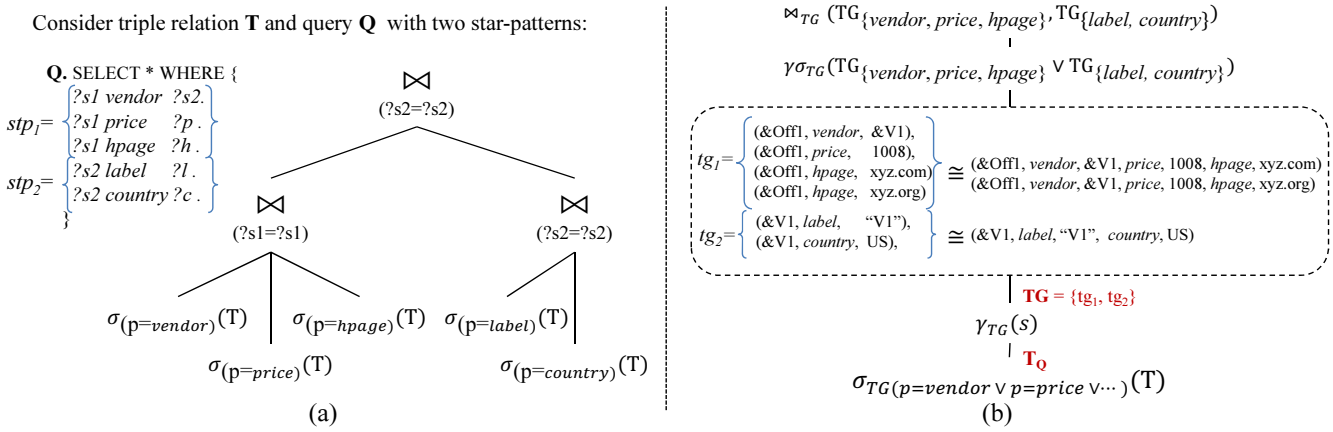

Figure 1: Processing of a graph pattern query Q with two star-patterns stp_1 and stp_2 over a triple relation T: a) The execution plan based on the relational-style approach and b) The NTGA-based plan

of processing. The RAPID+ data model and algebra is called Nested TripleGroup Data Model and Algebra (NTGA)[4], and is fully integrated into Apache Pig. Also, the demo will show comparisons of NTGA operators and query plans and the relational-style algebra query plans used by systems such as Apache Pig and Hive, along with a comparative performance of the two types of plans for graph pattern queries and simple ontological queries.

2. PROCESSING RDF GRAPH PATTERN QUERIES ON MAPREDUCE

RDF and SPARQL. An RDF database is a collection of triples (*Subject*, *Property*, *Object*) where Property is a named binary relation between resources (identified using URIs) or between resources and literal values. For example, in Fig.1(b), we have some simplified triples from the BSBM[9] benchmark dataset. The triple (&V1, hpage, xyz.com) asserts that the vendor &V1 (URI omitted for brevity) has the homepage xyz.com. A collection of triples may be queried using a "graph pattern", the fundamental querying construct of SPARQL[10], which is essentially a collection of triples in which at least one of the Subject, Property or Object is a variable denoted by a leading '?'.

Processing Queries with the Relational-style Approach. Fig.1(a) shows a graph pattern query **Q**, which retrieves the details of the product offer and their vendors. The first triple pattern will match triples whose property is *vendor* which is essentially a select operation on the property type. Triple patterns with common variables imply an implicit join condition. For example, the triple patterns sharing a common subject variables (e.g., *?s1* in Fig.1(a)) are processed with the multiple joins on the subject fields. Consequently, processing graph pattern queries is accomplished using a sequence of relational join operations. In MapReduce, all triple patterns that join on the same variable can be joined in the same MapReduce cycle. Therefore, a query is executed in a workflow whose length is equal to number of join variables. Fig.1(a) shows the corresponding execution plan processing two star patterns whose subject variables are *?s1, ?s2*, which connected with a common variable, *?s2*. Processing each star pattern requires two separate joins, resulting in two MR cycles, and connecting those two stars also requires additional join, producing the third MR cycle. Overall, three MR jobs are required.

[9] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark

[10] SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query

Processing Queries with the NTGA-based Approach. The nested triplegroup model represents RDF data as 'groups of related triples' or *triplegroups*. 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. Note that given an input data file, such a model can be created by a 'grouping' operation on the subject column of a triple relation. This is achieved by the NTGA operator called `TG_GroupBy`(γ_{TG}), which produces groups of triples in T_Q based on their subject; for example, $tg_1 \in TG_{\{vendor,price,hpage\}}$ in Fig.1(b) is a *Subject triplegroup* with triples sharing the common subject $\&Off1$. Such a grouping operation is restricted to only the property types involved in a query by prior filter/select operation. When loading triples in T, such filter operations are made by the NTGA operator `TG_LoadFilter`(σ_{TG}) based on properties in stp_1 and stp_2. Now, each resulting triplegroup can be viewed as a potential match for ***any*** of the star subquery patterns in a query. For example, for the query in Fig.1(b) involving 2 star patterns denoted as stp_1 = *vendor*, *price*, *hpage* and stp_2 = *label*, *county*, we see that the triplegroup tg_1 and tg_2 "matches" stp_1 and stp_2 respectively, which means that they contain the constituent triples for a match of those star patterns. Given the set of the triplegroups that result from the grouping phase, the next step is to ensure that each triplegroup meets all structural constraints of the subpattern that it is a potential match for i.e. checking that it is indeed a match in the operator `TG_GroupFilter`($\gamma\sigma_{TG}$). It enforces the structural constraints in a star subpattern e.g. tg_1 in Fig.1(b) is a valid match for the set of properties $\{vendor, price, hpage\}$ but violates the constraint $\{vendor, validFrom\}$, which leads to be pruned out. Finally, the valid matches are "joined" as specified in the graph pattern to create complete subgraph structures that match the graph pattern query using the `TG_Join`($\bowtie_{TG}$) operator. Since it is straightforward to "flatten" a triplegroup to yield an equivalent n-tuple, we say that triplegroups are *content-equivalent* to n-tuples (a concept made precise in [4]). Fig.1(b) shows the same result represented as an n-tuple.

From the query evaluation perspective, the advantage of NTGA-based approach is that we are able to compute results for ***all*** star subquery patterns in a single MapReduce cycle since a grouping operation is implemented as a single MapReduce cycle. This is much less expensive than the 1 cycle per star join approach that

would be needed for the relational algebra style plans in Pig and Hive. This has significant advantages in RDF query processing particularly since multiple star subquery patterns are quite common. Generally, TripleGroup-based pattern matching for a query with n star sub patterns are transformed into the MapReduce workflow with n MR jobs, e.g., 2 MR jobs for the query in Fig.1(a). Note that the same query requires $(2n-1)$ MR jobs when using the relational-style approach. Another salient point about the data model is that it allows the concise representation of intermediate results when processing queries involving properties that are multi-valued e.g a vendor may have several offers, etc. In the results formed by relational algebra expressions, intermediate results have some redundancy which adds avoidable I/O and network transfer costs to overall cost of processing. For example, the triplegroup stp_1 represents the object values on the *hpage* of *&Off1* as {xyz.com, xyz.org} while n-tuple repeats the predicate homepage, e.g., (&Off1, hpage, xyz.com, &Off1, hpage, xyz.org).

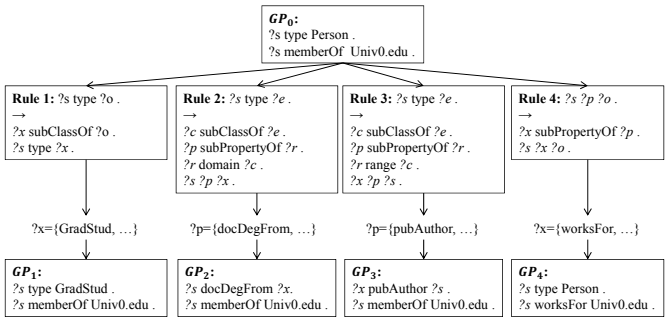

Figure 2: Query expansion using the re-writing rules for LUBM Q5.

2.1 Answering RDF(S) Ontological Queries on MapReduce

Given an RDF model and an ontology specification, it is possible that there are more triples entailed than those explicitly represented in an RDF model. In order to get complete answers to queries, we may first compute the closure of the model (all entailed triples) using reasoning algorithms. However, such techniques significantly blow up the size of the model and can be computationally expensive. Further, when updates are applied to the model, the reasoning process must be repeated to compute newly entailed triples. A different approach is motivated by the fact that given a query on a model, only a portion of an entailment is required to answer the query and this can be done at query time. Further, it can be done using so called "perfect re-writings" based on schema triples that rewrite such queries to unions of conjunctive queries (UCQs), which can be executed by relational processing systems. So an important aspect of query evaluation in this situation is how to evaluate such often large unions of conjunctive queries or graph patterns with UNIONs. In relational MapReduce-based frameworks like Pig and Hive, each branch of a UNION query is executed using a separate MapReduce job. For example, Fig.2 shows the generation of a set of schema-related subqueries that retrieve the inferred terms that are related to query $Q5$ in LUBM benchmark[11]. Leveraging the rules in [5], we generate subqueries GP_1, GP_2, and GP_3 from GP_0 that retrieve all inferred subclass, domain, and range relationships of *Person* for a triple pattern with *type* predicate such as *(?s, type, Person)*. For a triple pattern with any other predicate such as *(?s memberOf Univ0)*, we generate subqueries to retrieve all the subproperty relationships of *memberOf*, e.g., GP_4 from GP_0. Those subqueries are grouped with three UNION operators, e.g., $GP_1 \cup GP_2 \cup GP_3 \cup GP_4$, and can be executed using four individual MapReduce jobs.

As an alternative, more efficient approach can be used, which rewrites this query using the OPTIONAL (OPT) clauses. This idea is adapted from the proposal in [3] or multi-query optimization of SPARQL queries. To apply this here, we can use common subexpressions as the root of the query and then have OPT clauses represent the different alternative patterns for completing the query pattern. For example, the subquery GP_1, GP_2, and GP_3 share the common triple pattern *(?s memberOf Univ0.edu)*. We can merge these subqueries into the one using three OPT clauses, e.g., SELECT * WHERE { (?s memberOf Univ0.edu) OPT {(?s type GradStud)} OPT {(?s docDegFrom ?x)} OPT {(?x pubAuthor ?s)}}. We can process this merged query with three left outer joins (LOJ) such as $((GP_1 \bowtie GP_2) \bowtie GP_3)$. This approach requires additional filters and projections in the end to produce the equivalent result with the previous approach.

With NTGA, we can interpret those subqueries in a more efficient way. Essentially, we will consider the patterns in the different branches of the UNION operator as the parameter or constraints of $\gamma\sigma_{TG}$, which enforces star-join structural constraints against triplegroups as earlier described. For example, we can extract the set of predicates from each sub query (GP_1, GP_2, GP_3, GP_4) and build the parameter of $\gamma\sigma_{TG}$ as $(\{type, memberOf\} \vee \{pubAuthor, memberOf\} \vee \{docDegFrom, memberOf\}) \vee \{type, worksFor\}$.

One remaining issue is that the subquery GP_3 contains the two star patterns (i.e., *?s* and *?x*) , which require additional join operations to connect them after grouping operation. To avoid the use of the additional MR job for this join, NTGA *flips* the triples matching to *(?x pubAuthor ?s)* as *(?s pubAuthor ?x)*, which result in reversing the position of the subject and object values. Before producing the final answer, we flip back the corresponding triples for the correctness. From these techniques, NTGA maintains the number of required MR cycles as one regardless of the number of subqueries.

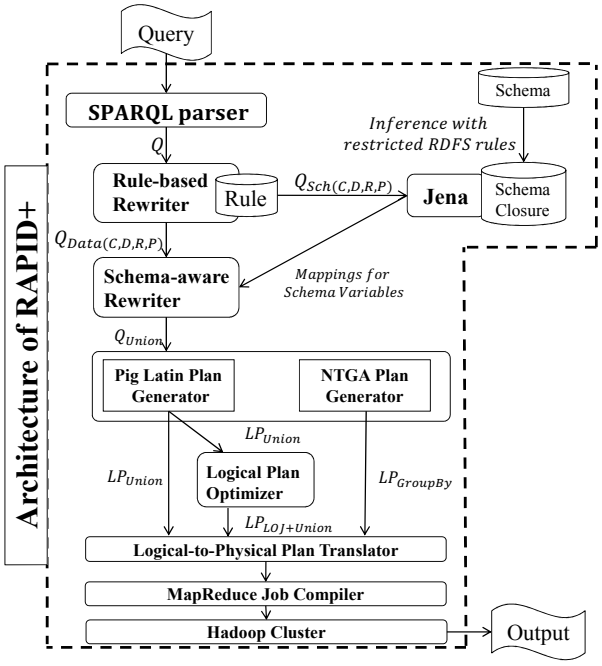

Figure 3: The architecture and Dataflow of RAPID+.

[11] LUBM Benchmark: http://swat.cse.lehigh.edu/projects/lubm

2.2 Architecture of RAPID+

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[12] or using NTGA-related high level commands that have been added to the Pig Latin interface. Fig.3 shows the general flow of processing the query. Once SPARQL parser parses the query Q, our *Rule-based Re-writer* expands it using the rules shown in Fig.2. The schema triple patterns in Q or $Q_{Sch(C,D,R,P)}$ are processed by Jena first, generating the variable mappings for the schema variables. With these mappings, *Schema-aware Rewriter* re-writes the remaining part of the query Q or $Q_{Data(C,D,R,P)}$, e.g., GP_1, GP_2, GP_3, and GP_4 in Fig.2, generating the union of sub queries, Q_{Union}. This is fed into the *Plan Generator* translating the query expression tree as the logical plan represented with the operators in Pig. *Pig Latin Plan Generator* builds the plan with the Pig's union operators or LP_{Union}, and this is further optimized by the *Logical Plan Optimizer*, converting it as the one with union of left outer joins $LP_{LOJ+Union}$ for the approach using optional patterns. Similarly, *NTGA Plan Generator* produces the plan consisting of NTGA operators, $LP_{GroupBy}$. All the logical plans are fed into the *Logical-to-Physical Plan Translator*, and compiled by the *Job Compiler*, producing a workflow of MR jobs in the end. Finally, they are executed in the Hadoop cluster and produced the expected output.

2.3 Preliminary Evaluations

Fig. 4(a) shows the preliminary performance evaluation of Union, LOJ-Union, and NTGA approaches for the 23GB dataset of LUBM dataset on a 5-node cluster in VCL[13]. NTGA shows a performance gain of 90% over Union approach for LUBM `Q3`, `Q4`, and `Q5`. In general, the length of the MR workflow using the Union approach depends on the size of rewritings, and performs the worst among the three approaches. Fig. 4(b) shows a scalability study with increasing size of the dataset (23GB and 45GB) on the 10-node cluster. With the increasing data size, NTGA scales better with an increasing performance gain of 36% (23GB) to 55%(45GB) over LOJ-Union for `Q4`, and 52%(23GB) to 65%(45GB) for `Q5` respectively.

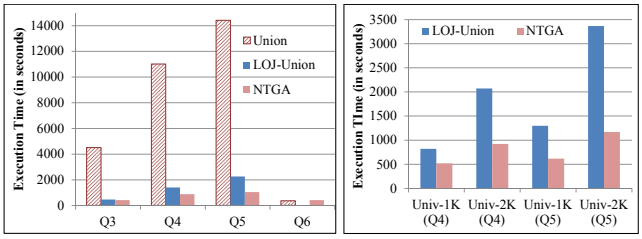

Figure 4: (a) Performance evaluation of the queries (5-node) (b) Scalability study with increasing size of RDF graphs (10-node).

3. DEMONSTRATION PROPOSAL

The goal of the demonstration is to allow users to explore query execution plans for processing graph pattern and ontological queries on MapReduce platforms. Users will be able to analyze the relational-style algebra plans used in Pig and Hive vs. the NTGA execution plans. Fig. 5 shows the screenshot presenting the logical plan based

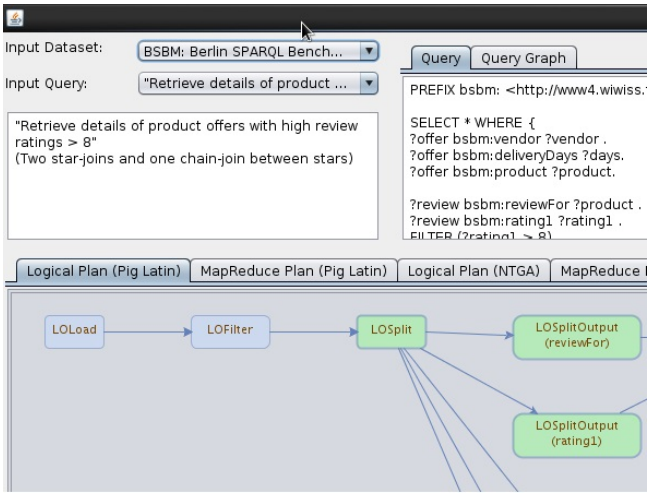

Figure 5: The screenshot of the demonstration showing the logical plan of the example query.

on the relational-style algebra for the selected query. Example metrics (e.g., the number of MapReduce cycles, the size of materialized intermediate results in bytes) will be provided for analysis to assess the impact of the two kinds of algebras and their operators on MapReduce processing. The demonstration will use a remote 5-node cluster hosted on VCL. In case of limited connectivity in the demo room, we will alternatively use a local Hadoop cluster.

3.1 Data Set and Queries

One synthetic benchmark and one real-world data set will be used for demonstration purposes. The LUBM benchmark dataset contains information about Universities, while Unitprot[14] is a real-world biological dataset containing protein sequence and function information. In this demonstration, we will use (i) graph pattern matching queries in SPARQL with varying sub structures (varying number of joins, varying number of star subpatterns, and varying selectivity), and (ii) ontological queries with varying size of inference-based rewritings.

4. ACKNOWLEDGMENT

The work presented in this paper is partially funded by NSF grants IIS-0915865 and IIS-1218277.

5. REFERENCES

[1] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI* (2004), pp. 10–10.

[2] KIM, H., RAVINDRA, P., AND ANYANWU, K. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. *Proc. VLDB 4*, 12 (2011).

[3] LE, W., KEMENTSIETSIDIS, A., DUAN, S., AND LI, F. Scalable Multi-query Optimization for SPARQL. In *ICDE* (2012), pp. 666–677.

[4] RAVINDRA, P., KIM, H., AND ANYANWU, K. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *Proc. ESWC*, vol. 6644. 2011, pp. 46–61.

[5] STUCKENSCHMIDT, H., AND BROEKSTRA, J. Time-Space Trade-offs in Scaling up RDF Schema Reasoning. In *Proc. WISE* (2005), pp. 172–181.

[12] Apache Jena: http://jena.apache.org

[13] Virtual Computing Lab in NCSU: http://vcl.ncsu.edu

[14] UniProt (Universal Protein Resource): http://www.uniprot.org