Efficient Processing of RDF Graph Pattern Matching on MapReduce Platforms

Padmashree Ravindra Seokyong Hong HyeongSik Kim Kemafor Anyanwu
Department of Computer Science, North Carolina State University
{pravind2, shong3, hkim22, kogan}@ncsu.edu

ABSTRACT
Broadened adoption of the Linking Open Data tenets has led to a significant surge in the amount of Semantic Web data, particularly RDF data. This has positioned the issue of scalable data processing techniques for RDF as a central issue in the Semantic Web research community. The RDF data model is a fine-grained model representing relationships as binary relations. Thus, answering queries (typically *graph pattern matching* queries) over RDF data requires several join operations to reassemble related data. While MapReduce based processing is emerging as the *de facto* paradigm for processing large scale data, it is known to be inefficient for join-intensive workloads. In addition, most of the existing techniques for optimizing RDF data processing do not transfer well to the MapReduce model and often require significant lead time for pre-processing. Such a requirement may not be desirable for *on-demand* cloud database scenarios where the goal is to reduce the *Time-To-Result* (TTR). In this position paper, we argue that some of these challenges can be overcome by rethinking the operators for graph pattern processing, as well as adopting dynamic optimization techniques that exploit information from the previous execution steps to eliminate intermediate results that are irrelevant in the context of future execution steps. We present some preliminary evaluation results of the proposed techniques.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems - *query processing*

General Terms
Algorithms, Languages, Performance

Keywords
RDF graph pattern matching, SPARQL, MapReduce, Hadoop

1. INTRODUCTION
There has been a rapid increase in the amount of available Semantic Web data in the past few years as Semantic Web tenets gain broadened adoption. This includes data from domains ranging from scientific (e.g. DrugBank, Linked Clinical Trials), business (e.g. BBC, New York Times), government (e.g. data.gov, data.gov.uk) and general purpose communities (Wikipedia, Linked Open Data). To give a sense of the kind of growth, we note that the Linked Open Data cloud grew from around 26 billion RDF[1] triples in September 2010 to over 31 billion triples by the following year. [19] Therefore, one of the important issues in Semantic Web community is the development of scalable and efficient techniques for processing large amounts of Semantic Web data. The most common representation model for Semantic Web is called the Resource Description Framework (RDF). RDF database is a collection of triples (*Subject*, *Predicate*, *Object*) where *Predicates* are named binary relations between *Subject* and *Object* that represent either resources or literal values in the Web. The RDF data model can also be viewed as a directed and labeled graph with nodes representing *Subject* / *Object* that are connected by labeled edges representing *Predicates*.

The key construct for processing RDF data is *graph pattern matching*. Here, users describe the required structure of patterns as queries and systems return as answers all occurrences of the pattern found in the data. The standard query language for expressing graph pattern matching queries on RDF data is called SPARQL[2]. Each triple pattern in a given SPARQL query is a triple in which at least one of the *Subject*, *Predicate* or *Object* is a variable, denoted by a leading question mark. Therefore, the query attempts to match those patterns to sub graphs in the database and the result of a graph pattern query is a list of all variables substituted from those graphs. For example, the triple {Vendor1, *homepage*, www.vendors.org/v1} in Figure 1 (a), represents that the homepage of a resource Vendor1 is www.vendors.org/v1. This triple matches the triple pattern {Vendor1 *homepage* ?homepage}, where the variable ?homepage has a valid binding to the object value www.vendors.org/v1 of the example triple.

Traditionally, RDF data is stored as a ternary relation and graph pattern matching queries are processed as a series of relational join operations. Due to the fine-grained nature of the RDF data model, it is common to require several join operations to answer a slightly complex query. Another commonly performed task on Semantic Web data is *reasoning*, that applies the standardized semantic inference rules to compute all inferable facts from the set of explicitly represented facts. Depending on the approach for performing reasoning, this task also relies heavily on joins. Thus, the ability to process join-intensive workloads is crucial for processing Semantic Web data.

When considering large scale processing and analytics in data-intensive applications using cloud environment, the recent *de facto* standard is the MapReduce [3] programming model made popular by Google and its open source implementations such as Hadoop [18]. Such platforms have been explored for graph pattern matching [5][6][13][15]. However, existing approaches naively translate join-intensive workloads such as graph pattern matching into a long chain of MapReduce jobs, leading to significant I/O, communication, and sorting overhead as discussed in the next section. Another typical problem of processing join operations on MapReduce is that the entire relations to be joined must be loaded,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DataCloud-SC'11, November 14, 2011, Seattle, Washington, USA.
Copyright 2011 ACM 978-1-4503-1144-1/11/11...$10.00.

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

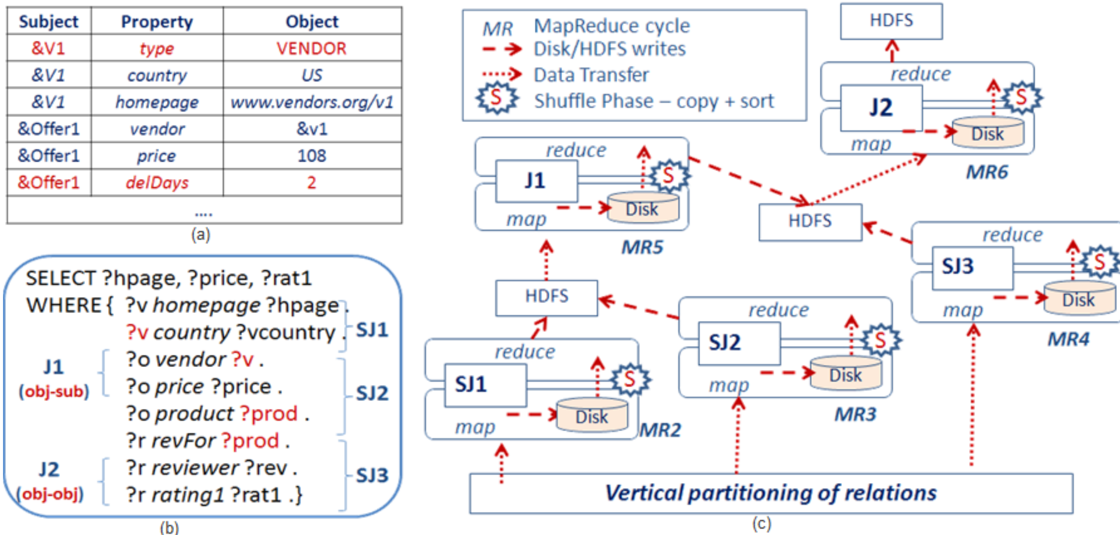

Figure 1 (a) Example RDF triples representing Vendors and their Product Offers (b) SPARQL query to retrieve details about Vendors, their Offers and the Product Reviews (c) Query processing in Pig

sorted, and transferred between map and reduce phases even if they do not eventually join. Such irrelevant record processing can degrade the performance while processing such join-intensive workloads. Further, we also show how adopting existing optimization techniques used for single-node environments to MapReduce framework is not straightforward. Therefore, it is our position that the development of new strategies is crucial. We overview two complimentary strategies that can be used to optimize graph pattern matching queries without the need for time consuming preprocessing, (i) *algebraic optimization based on a new algebra called the Nested TripleGroup Algebra* – supports a grouping based approach for star-join computation, that reduces the #MR cycles required to process graph pattern matching queries, and (ii) *inter-job information passing* – generates and sends summary information on intermediate data of previous jobs to other relevant jobs in the query plan, that can be used to prevent irrelevant data from being loaded, sorted, and transferred between MapReduce jobs in a query plan. The remainder of this paper is organized as follows: Section 2 provides some relevant background. Section 3 gives an overview of the challenges that we encountered. Section 4 describes the approach we are proposing to overcome the challenge, followed by a case study in Section 5.

2. BACKGROUND

2.1 Join Processing in MapReduce

In the MapReduce programming model, users encode their tasks in terms of two functions: the *map* whose general signature is map (*Key1*, *Value1*) → list (*Key2*, *Value2*), processes input key-value pairs and produces intermediate data as a list of key-value pairs; the *reduce* with signature reduce (*Key2*, list (*Value2*)) → (*Key3*,*Value3*) merges values into a group according to the intermediate key, and generates a final key-value pair for each group. Hadoop is an open source implementation of the MapReduce model and provides fault-tolerance and automatic parallelization of the map and reduce functions. Its architecture includes a master process (*JobTracker*) which schedules *m* instances of the map function on nodes (*mappers*) and *r* instances of the reduce function on nodes (*reducers*) over the cluster. The JobTracker splits the input data into "chunks" which are assigned to mappers. Once the mappers produce and write their output i.e. collection of key value pairs to disk, each key is mapped to a unique reduce task. This mapping forms the basis of data distribution among the reducers and thus they may get unequal amounts of data to process. The reduce phase consists of 3 sub-phases: *copy* - the map output is copied from the disk at the mapper nodes to reducer nodes, *sort* - the collected map output is sorted based on key values and *reduce* - reduce function is applied to the data. After the reduce function is completed, their outputs are saved to the *Hadoop Distributed File System - HDFS*.

To interpret the relational join operation in the MapReduce paradigm, each tuple from the participating relations is tagged based on the join columns. This will enable all the records with the same join key to be assigned and transferred to the same reducer. In reduce-phase, each of *r* reducers accumulates the records into separate in-memory buffers according to their tags and performs the join. This is called the *Standard Repartitioning Join* [2]. One thing to note is that all intermediate data generated in map-phase has to be sorted and transferred into the reduce-phase. These data sorting and transferring steps are crucial cost factors that may affect the data processing time on MapReduce [2][4]. Moreover, in a case that the join selectivity is high, a large number of records are involved in such data sorting and transfer steps even if those records are not eventually joined.

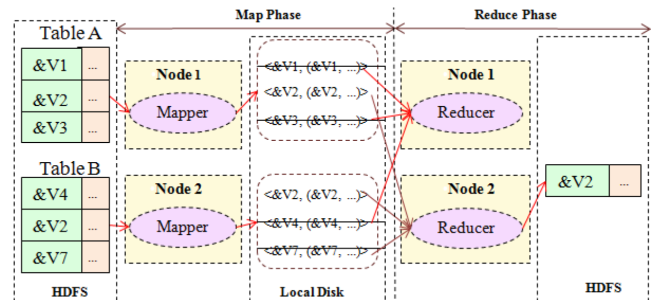

Figure 2: An example Hadoop job plan for processing joins

Figure 2 shows an example Hadoop plan for processing an equi-join of two relations, *TableA* and *TableB*. It can be observed that only one record, *&V2*, is part of the final join result. However, other irrelevant records are loaded and sorted in the map-phase, and transferred between the map and reduce phases, even though

they are eventually filtered out by the equi-join operation. In the traditional DBMSs, several techniques such as indexing can be exploited to remove such irrelevant records. However, the Hadoop framework and its extensions such as Hive and Pig do not provide such valuable indexing mechanism.

Fragment-Replication Join and *Map-Merge Join* are alternative join approaches that can remove the data sorting and shuffling overhead by processing the join in the map-phase [2]. However, the fragment-replication join is applicable only when one of the input tables is small enough to fit in memory. In the case of the map-merge join, an additional pre-processing phase is required to partition input data according to its join key. More importantly, this approach cannot effectively deal with scenarios where one of the inputs to the join is an intermediate result generated from earlier query operators. Another recently proposed technique is the *Semi-Join* [2], but it also suffers from similar limitations.

2.2 Generating MapReduce Execution Workflows from Declarative Queries

Typical data processing tasks may involve several operations or multiple instances of the same operation. A task may include one or more filtering conditions, join, grouping, and aggregation operations. Executing such a task on MapReduce requires users to determine the appropriate map and reduce functions and to implement them in a way that achieves the most efficient workflow, which is not always trivial. In order to eliminate this burden on the users, Hadoop extensions such as Apache *Hive* [16] and *Pig* [11] offer high-level declarative query and dataflow languages, *HiveQL* and *Pig Latin* respectively, that are compiled to generate efficient MapReduce workflows. This is similar in spirit to database systems where users write high-level declarative queries and the system is responsible for generating an optimized execution plan. The high-level query expressed by the user is compiled into a series of MapReduce (MR) cycles where the output generated at the end of one cycle is saved onto the HDFS and fed as input into the next MR cycle forming a MapReduce workflow. Naïve compilers in some existing systems follow rule-based translation of each relational operator into one or more MR cycles. Such naïve translations in the case of join-intensive workloads such as RDF graph pattern matching result in lengthy MapReduce workflows with a separate MR cycle for each join operation. The example graph pattern query in Figure 1(b) would be compiled into 7 self-join operations on a triple data model, leading to 7 MR cycles. Lengthy execution workflows lead to performance inefficiency since each MR cycle compounds the overall I/O, sorting, and communication cost, in addition to the materialization cost between two contiguous cycles. This motivates the need for optimization techniques that minimize the length of MapReduce workflows and also reduce the intermediate data. A possible optimization to reduce the required #MR cycles is to group related operations into the same MR cycle. For example, n-way join operations can be used to replace joins on the same column. Using this optimization, the example graph pattern in Figure 1 (b) can be executed as 3 n-way join operations (SJ1, SJ2, and SJ3 respectively), which reduces the overall #MR cycles to 5 (3 for the n-way joins, and 2 to further join them) as shown in Figure 1 (c). Further, we can avoid the expensive self-join operations on large ternary relations and reduce the initial load costs by adopting the *vertical partitioning* (VP) [1] storage model that partitions the input into property-based relations. The VP approach can be implemented in Pig Latin by using the `SPLIT` operator. Figure 3 (a) (top) shows Pig's logical plan corresponding to our example query in Figure 1 (b) using the VP approach. In the next section, we discuss other optimization techniques and review the challenges of optimizing graph pattern matching queries on MapReduce-based platforms.

2.3 CHALLENGES

The key challenges to overcome in order to achieve efficient processing of RDF data processing are (1) minimizing the length of MapReduce execution workflows – reduces the overall costs of processing by reducing the number of iterations for disk I/Os, communication and sorting steps; (2) minimizing the size of intermediate data - reduces the amount of data written and read from disk i.e. disk I/O, the amount of data communicated between nodes and, the size of data sorted during the sorting steps. A natural inclination is to explore how existing optimization strategies for graph pattern matching designed for single-node environments can be adopted into the MapReduce model to achieve these goals. As the following discussion will show, there are several challenges that limit the possibility of doing so, suggesting the need for other kinds of strategies.

The state-of-the-art techniques for graph pattern matching [10] [17] rely heavily on comprehensively indexing RDF data. The data is indexed in all possible ways that allows every execution step to be supported by some index. In particular, the goal is always to enable the use of the merge-join algorithm for as many join operations as possible because it is the most efficient join algorithm for sorted data. Consequently, most indexing schemes are ordered index structures such as B+Trees that maintain data in a desired order. These comprehensive indexing strategies enable "index-only" plans that do not require any other files for query processing except the index files. Given this scenario, the major challenge in the context of MapReduce is how to use these indexes during the map and reduce phases of each join. These data structures will need to be "chunked" and partitioned across nodes similar to the way data is done. Such a partitioning strategy is not straightforward because such tree structures will need to be chunked in meaningful ways that maintain root-to-leaf path relationships. Further, this approach would require performing a single-join-operation-per-MR-cycle. Note that the cost based optimization used in RDF-3X [10] just changes the order of joins but does not change the number of join steps.

While the previously discussed techniques index data at the level of binary relations, other approaches try to index data at a coarser level of granularity. This is motivated by the fact that the task of reassembling related data in RDF is very common in graph pattern matching queries and hence a good fraction of the joins in queries focus on this reassembly task. The reassembly process results in star-shaped join structures that can be exploited for optimization (refer to SJ1, SJ2, and SJ3 in Figure 1(b)). In order to efficiently match the star-structured graph patterns to star subgraphs in data, we can cluster related data prior to query processing. By doing so, it might appear that we could eliminate the need for several join operations potentially reducing the number of MR cycles. In fact, it may be possible to implement such join operations as a special kind of complex filter operation in this storage model. Unfortunately, realizing this is not as straightforward as it might first seem. Since RDF is semi-structured and represented using binary relations, clustering related data results in variable length tuples containing data and metadata. As an example, in the strategy employed in SHARD [13], the example triple data in Figure 1 (a) will appear as the following after clustering:

(&V1, *type,* VENDOR, *country,* US, *homepage,* www.vendors.org/v1)

(&Offer1, *vendor*, &V1, *price,* 108, *delDays,* 2)

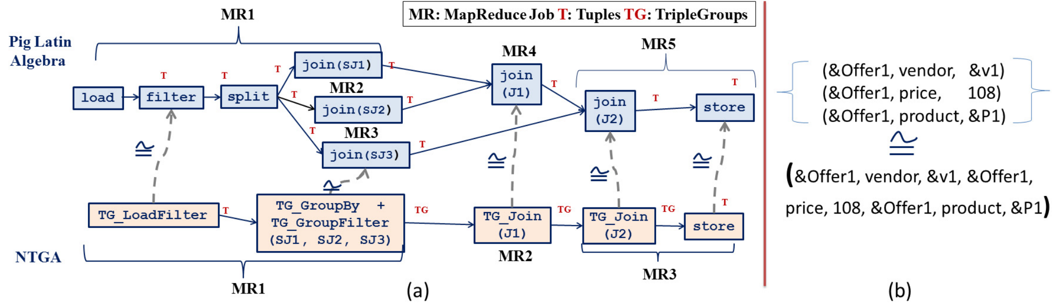

Figure 3: (a) Comparison and mapping between relational and NTGA for the example query plan (b) An example TripleGroup and its content equivalent n-tuple

To match the star-join pattern with the *country* and *homepage* predicates, the algorithm will first need to identify the relevant columns in each tuple, since the columns in this clustered model are not identified in a meaningful way like in the relational model.

Here attribute names as well as their values are part of the data and the number of occurrences of a particular attribute may differ from row to row. Therefore, extracting the relevant fields does not translate to the relational filter operation, which motivates the need for a specialized operator when using this storage model. In SHARD, the matching process is done using an iterative join strategy where during each iteration, the columns related to one particular predicate are identified using a special function and the values in those columns are joined to the current intermediate result. This approach results in as many MapReduce cycles as the number of join operations. Another limitation common to all these discussed approaches is that they all require preprocessing either for computing indexes or clustering data and so on. Often these preprocessing steps are in the order of hours which can be limiting in the case of on-demand data processing tasks where the main goal is to reduce the required Time-To-Result (time from availability of input to generation of output).

In the sequel, we will present some techniques that address the two challenges earlier mentioned. To reduce the length of the workflow, we propose to *reinterpret the set of star joins in a query as a* `GROUP-BY` *operation* [1] . The results are equivalent to the results using join operations with respect to content but differ in their structure and representation. However, this allows all star-joins to be computed concurrently in a single MR cycle. To deal with the problem of reducing intermediate data, we propose an *inter-job information passing* approach that summarizes the intermediate content of previous jobs and passes the summary information to later jobs to prune in advance irrelevant records which are involved in expensive data I/O and transmission steps. In traditional DBMSs, this approach is named as *Sideways Information Passing*, a set of techniques that generate summary information about intermediate records processed in a part of a query plan to another relevant part to filter out irrelevant records. However, it is not straightforward to adapt such techniques to the MapReduce platforms. For example, RDF-3X assumes pipelined parallelism and shared-memory environment which allows summary information to be generated and exchanged between operators in an efficient manner. However, the MapReduce computing model assumes partitioned parallelism and there is no communication method to efficiently send information between operators. Therefore, it is necessary to develop an information passing framework that is easily and effectively adaptable to MapReduce environment.

3. PROPOSED APPROACH

3.1 Algebraic Optimization - TripleGroup based Processing

We propose to re-interpret the star-join operations in a graph pattern as a `GROUP BY` operation. The rationale for the grouping approach is that typical graph pattern matching queries involve multiple star-join structures (50% of BSBM benchmark queries have 2 or more star-patterns), which would have otherwise led to a separate MR cycle to computer each star-join. However, by using a single `GROUP BY` operation, all the required star-joins can be computed concurrently in a single MR cycle. For example, the 3 star-joins (SJ1, SJ2, and SJ3) in Figure 1 (b) can be computed using a single `GROUP BY` on *Subject* in a single MR cycle, thus reducing the required #MR cycles. However, this grouping strategy produces "groups of triples" or *TripleGroups* instead of n-tuples, and requires specialized operators for efficient processing of TripleGroups. Figure 3 (b) (top) shows an example TripleGroup corresponding to the group of triples with *Subject* &Offer1, and belongs to the equivalence class $\text{TG}_{\{vendor,\, price,\, product\}}$.

We propose an intermediate *Nested Triplegroup algebra* (NTGA) that supports special TripleGroup based operators for efficient processing of RDF graph patterns. The triple relation is loaded using NTGA's `TG_LoadFilter` which *coalesces* two operations: `TG_Load` - loads RDF statements into TripleGroups and `TG_Filter` - eliminates irrelevant TripleGroups that fail to satisfy the SPARQL `FILTER` construct i.e. *value-based filtering*. The value-based filtering is processed during the load phase since Hadoop does not currently support efficient indexing scheme to selectively retrieve records from HDFS. The `TG_GroupBy` operator is equivalent to the grouping operator in relational algebra, and generates TripleGroups based on the common *Subject*. The TripleGroups thus generated are filtered using the `TG_Groupfilter` operator to eliminate TripleGroups with missing edges that violate the structural constraints specified in the given query. For example, the following TripleGroup *tg1*:

tg1 = { (&V2, *type*, VENDOR),
(&V2, *country*, US),
(&V2, *delDays*, 6) }

does not satisfy the structure-based filtering operation:

`TG_GroupBy` ($\mathbf{TG}_{\{type,\, country,\, homepage\}}$)

due to the missing edge corresponding to the *Predicate* homepage. After the completion of structure-based filtering, we process the joins between the star patterns using the `TG_Join` operator. The `TG_Join` operator is semantically equivalent to the relational join operator except that it is defined on TripleGroups. The object-subject join operation (`J1` in Figure 1 (b)) between the TripleGroup classes $\mathbf{TG}_{\{vendor,\, price,\, product\}}$ (corresponding to star-

join SJ2) and $\mathbf{TG}_{\{type,\ country,\ homepage\}}$ (corresponding to star-join SJ1) respectively can be expressed as follows:

TG_Join (?o *vendor* ?v: $\mathbf{TG}_{\{vendor,\ price,\ product\}}$
?v *country* ?vcountry: $\mathbf{TG}_{\{type,\ country,\ homepage\}}$,)

and results in a nested TripleGroup *ntg* whose root is the TripleGroup sharing subject *&Offer1* and the child is the TripleGroup sharing the subject *&V1* (marked in bold):

ntg = { (&Offer1, *vendor*, { **(&V1, *type*, VENDOR),**
(&V1, *country*, US),
(&V1, *homepage*, www.vendors.org/v1),
}),
(&Offer1, *price*, 108),
(&Offer1, *product*, &P1) }

The support for nested TripleGroups allows for more natural representation of RDF graphs using NTGA. NTGA also supports flattening operators to create an n-tuple equivalent of a TripleGroup (TG_Flatten), and to flatten a nested TripleGroup (TG_Unnest). Figure 3 (b) (bottom) shows the equivalent n-tuple resulting from the TG_Flatten operator on the example TripleGroup (top) sharing the common *Subject* &Offer1. Figure 3 (a) shows a comparison of the MR execution workflow to process the example query in Figure 1 (b) using the Pig Latin and the NTGA operators. It also demonstrates that the example query can be executed in just 3 MR cycles using NTGA as opposed to 5 MR cycles using the Pig approach. Figure 3 (a) also represents the mapping between the two plans through the concept of *content equivalence* (denoted by ≅) that enables lossless transformation between queries written in relational algebra and NTGA.

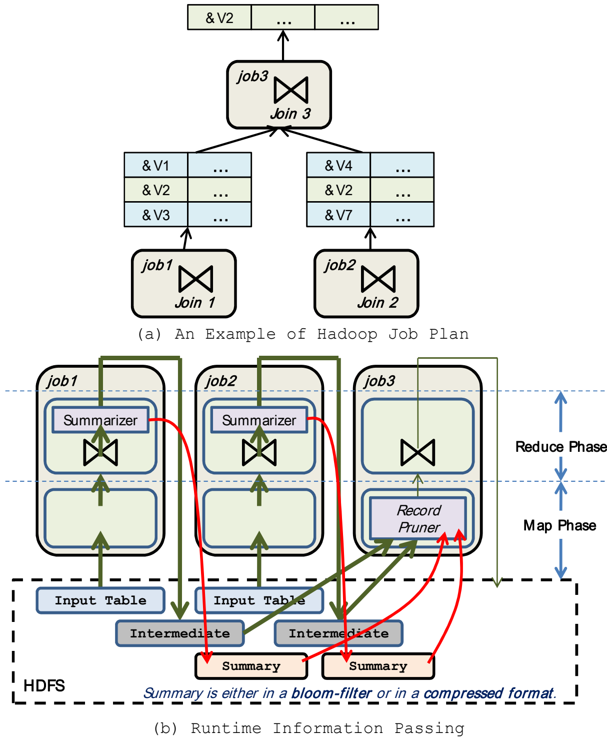

Figure 4: Information Passing in Hadoop Query Plan

3.2 Dynamic Optimization – Information Passing

As discussed in the previous section, traditional sideways information passing cannot be deployed in the Hadoop environment which assumes partitioned parallelism and lacks of efficient communication media. Hence, we design and develop a new adaptive information passing technique called *Inter-job Information Passing* that runs At compile-time, the query optimizer collects all the possible dataflow between MapReduce jobs from the physical query plan, calculates all the equivalent classes, creates an information passing plan, and finally embeds the plan into the Hadoop job plan.

Figure 4 (a) is an example of a Hadoop job plan where the Hadoop job ***job3*** receives its inputs from the previous jobs, ***job1*** and ***job2*** respectively. Figure 4 (b) shows a brief representation of the run-time information passing operations. When ***job1*** and ***job2*** are executed, the summarizer receives the streams of outputs, and creates the summary information on the columns that will be joined in ***job3***. Then, the summary information is represented in a more compact format to reduce the data transfer costs, and is stored in the shared information store that resides on HDFS. When ***job3*** runs, the previous summary information is loaded into *PSO* (*Parameterized Filter Operator*) at Map-phase and used to filter out the input data streams. For example, in Figure 4 (b), ***job1*** generates a filter than contains a list of distinct join key values - &V1, &V2 and &V3 and ***job2*** generates a filter containing a list of &V2, &V4 and &V7 respectively. When ***job3*** runs, it loads the two lists into the in-memory buffer and filters out all records from each input stream that do not match the values in the corresponding list. Therefore, the generated summary information can eliminate the irrelevant records, &V1, &V3, &V4, and &V7 from participating in the expensive sorting and data transfer steps within the job cycle.

Another important issue is that the overhead resulting from the creation and the transmission of summary information may exceed the performance gain achieved by the information passing technique in some cases. Therefore, we provide a *benefit estimation model* and integrate it with the query optimizer to dynamically enable or disable the feature of the technique. Our benefit estimation model considers the possible benefits in sorting, transferring, and merging steps which can be earned from data reduction caused by a given size of summary information. Also, the model considers the overhead caused by generating and transferring summary information via Hadoop's DistributedCache.

3.3 General Query Plan strategy

We address the two challenges in MapReduce-based graph pattern matching (described in section 0) by applying the two proposed techniques. First, we minimize the #MR cycles required for the star-join computation phase by using the TripleGroup-based processing using the NTGA operators. In addition, we can collect the summary information during NTGA's star-join computation phase and integrate the information-passing technique to reduce the intermediate data in the subsequent MR cycles. Thus, in the map phase of the TG_Join operator, we can eliminate irrelevant TripleGroups that do not join in the subsequent phases. Both these techniques help in minimizing the overall I/O, communication, sorting and materialization costs, leading to efficient processing of RDF data.

4. CASE STUDY

In this section, we present an empirical study of the performance of the two optimization techniques discussed in section 3.1 and 3.2 respectively to enable efficient RDF graph pattern matching on MapReduce-based platforms.

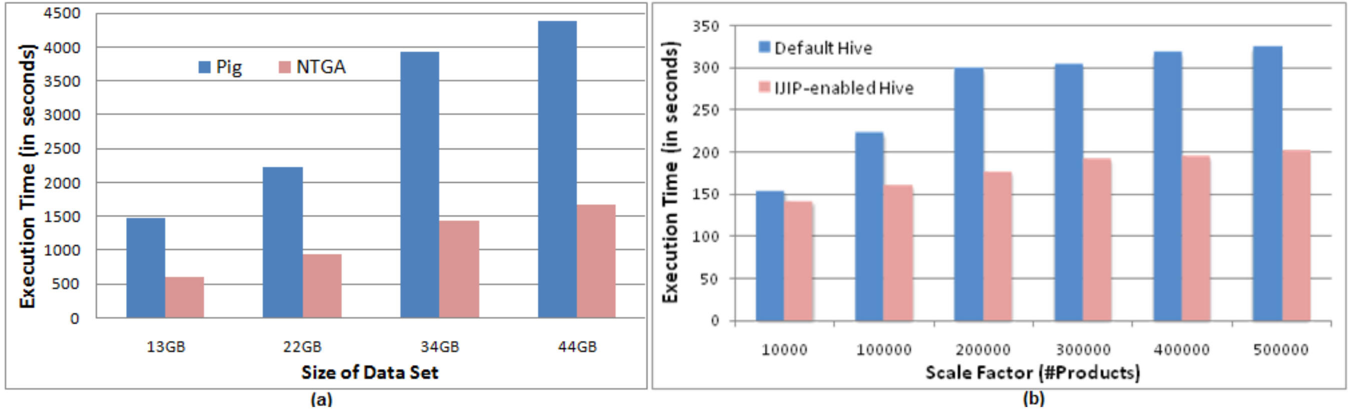

Figure 5: Scalability study with increasing size of RDF graphs for (a) TripleGroup-based processing of RDF graph patterns and (b) Inter-job Information Passing technique for join-intensive workloads (5-node cluster)

4.1 Task Description

Task A - Scalability of TripleGroup-based processing: We evaluated the performance of the NTGA approach with increasing size of RDF graphs. We extended Pig to integrate the NTGA operators [7] and compared its performance with that of RDF graph processing using relational operators in naïve Pig.

Task B - Performance Evaluation of Hybrid Plans: We also performed a comparative evaluation of a hybrid approach containing a mix of Pig Latin and NTGA operators. The notion of *content equivalence* allows for lossless translations between the algebra as shown in Figure 3 (a). For example, each TripleGroup in the result of the star-join computation phase using NTGA's `TG_GroupBy` (MR1 in NTGA plan in Figure 3 (a)) is content equivalent to some n-tuple in the result of the corresponding phase using Pig's `JOIN` operator (MR1, MR2, and MR3 in Pig Latin plan). For the hybrid approach, we considered two execution plans, (i) *NTGA-StarJoin* – using NTGA's `TG_GroupBy` operator for star-join computations, `TG_Flatten` to flatten the TripleGroups into n-tuples, and Pig's `JOIN` operator to compute the join between the stars; (ii) *Pig-StarJoin* – Pig's n-way `JOIN` operator for star-join computations, NTGA's `TG_Unflatten` to convert n-tuples to TripleGroups, and NTGA's `TG_Join` to compute the join between the stars.

Task C – Scalability of Information-Passing technique: We evaluated the benefit of *information passing*, with increasing size of RDF graphs. We extended Hive to enable inter-job information passing (IJIP-enabled Hive) and compared its performance to the default Hive implementation. However, the proposed information passing technique uses a general and abstract query processing model and, therefore, is applicable to other MapReduce-based data processing systems such as Pig.

4.2 Setup and Testbed

Environment: The experiments were conducted on a 5-node cluster on NC State's VCL [14], an on-demand virtual computing environment. Each node in the cluster was a dual core Intel x86 machine with 2.33 GHz processor speed, 3GB RAM and running Red Hat Linux. Experiments used Pig release 0.8.0 and Hive 0.5.0 running on Hadoop 20.0 with block size 256MB.

Testbed – Data Set and Queries: Synthetic BSBM[3] dataset generator was used to generate n-triple files for Task A, with data size ranging from 13GB to 44GB (approx. 170 million triples). The query used for Task A consisted of 6 triple patterns requiring 5 join operations, which comprised of two star-join structures. Two test queries with varying star cardinality were used for Task B (i) *q-small* – three star sub-patterns with 1, 3 and 1 triple pattern in each star respectively, (ii) *q-dense* – three star sub-patterns with 3 triple patterns in each star. Task B was evaluated on input data containing 50,000 products, which corresponds to approx. 17 million triples (approx. 4.3 GB data size). For Task C, three separate Hive tables were generated from BSBM in the format of SQL-dump with the #products ranging from 1000 to 500,000 (approx. 50GB data size). Task C evaluated a query requiring 3 join operations on these three tables.

4.3 Evaluation Results

Task A: Figure 5 (a) shows the execution times for the NTGA and naïve Pig implementations. For all the four data sizes, we see up to 60% improvement in the execution times with the NTGA approach. In the Pig approach, the most efficient execution plan is to compute the two star-patterns using n-way joins, which can be achieved in 3 MR cycles. NTGA's grouping approach computes the 2 star-joins in a single MR cycle, reducing the required #MR cycles to just 2. The advantage is more obvious in complex graph pattern queries with several star-joins.

Task B: Figure 6 shows the comparative evaluation of the two hybrid plans along with the NTGA and Pig execution plans for the two query patterns q-small and q-dense respectively.

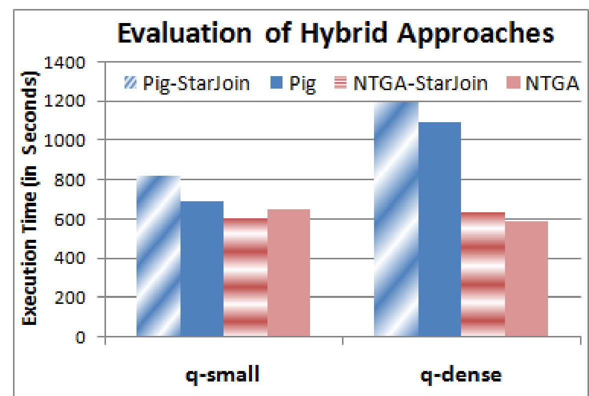

Figure 6: Performance evaluation of hybrid plans with varying density of star sub-patterns (4.3GB, 5-node cluster)

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

Pig-StarJoin and Pig show low performance for both queries since the star-join computation using Pig's `JOIN` operator leads to lengthy execution workflows. Pig-StarJoin does worse than Pig due to the overhead of converting n-tuples to TripleGroups before the join between the stars. On the other hand, NTGA-StarJoin and NTGA show a performance gain of 42% and 46% respectively over Pig approaches for the denser query pattern. This demonstrates the advantage of shorter execution plans using the NTGA approach to compute star-joins.

Task C: Figure 5 (b) shows that the IP-enabled implementation outperforms the default Hive implementation as the size of the input grows. This provides a promising approach to reduce the size of intermediate data, specifically for processing join-intensive workloads on MapReduce based platforms.

5. CONCLUSION

In this position paper, we propose two optimization techniques that enable efficient processing of join-intensive workloads such as RDF graph pattern matching. These two techniques help to shorten the MR execution workflows, and reduce the intermediate tuples, resulting in less I/O, sorting, and communication costs involved in traditional MapReduce based RDF graph processing.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management using Vertical Partitioning. In Proc. International Conference on Very Large Data Bases, 2007.

[2] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In Proc. International Conference on Management of Data, 2010

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proc. Conference on Symposium on Operating Systems Design & Implementation. Vol. 6. 10-10. 2004

[4] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (without It Even Noticing), In Proc. VLDB Endow. 3, volume 3, 515–529, 2010

[5] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. In Proc. VLDB Endow. 4(11), 2011.

[6] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In Proc. International Conference on Cloud Computing, 2010.

[7] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. In Proc. VLDB Endow. 4(12), 2011.

[8] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of Magic-sets in a Relational Database System. In Proc. International Conference on Management of Data, pages 103–114, 1994.

[9] Thomas Neumann and Gerhard Weikum. Scalable Join Processing on Very Large RDF Graphs. In Proc. International Conference on Management of Data, 2009.

[10] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. In Proc. VLDB Endowment, 19:91–113, 2010.

[11] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In Proc. International Conference on Management of Data, 2008.

[12] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In Proc. Extended Semantic Web Conference, 2011.

[13] Kurt Rohloff and Richard E. Schantz. High-performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: the SHARD Triple-store. In Programming Support Innovations for Emerging Distributed Applications, PSI EtA '10, pages 4:1–4:5, 2010.

[14] H.E. Schaffer, S.F. Averitt, M.I. Hoit, A. Peeler, E.D. Sills, and M.A. Vouk. NCSU's Virtual Computing Lab: A Cloud Computing Solution. In Computer, 42:94–97, 2009.

[15] Yusuke Tanimura, Akiyoshi Matono, Steven Lynden, and Isao Kojima. Extensions to the Pig data processing platform for scalable RDF data processing using Hadoop. In Proc. International Conference on Data Engineering Workshops, 2010.

[16] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a Map-Reduce framework. In Proc. VLDB Endow. 2:1626–1629, 2009.

[17] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. In Proc. VLDB Endow. 1:1008–1019, August 2008.

[18] A. Bialecki, M. Cafarella, D. Cutting, and O. O Malley. Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware. http://hadoop.apache.org/

[19] R.Cyganiak and A. Jentzsch The Linking Open Data Cloud Diagram http://www4.wiwiss.fu-berlin.de/lodcloud/state/