An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce

Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu

Department of Computer Science, North Carolina State University, Raleigh, NC
{pravind2,hkim22,kogan}@ncsu.edu

Abstract. Existing MapReduce systems support relational style join operators which translate multi-join query plans into several Map-Reduce cycles. This leads to high I/O and communication costs due to the multiple data transfer steps between *map* and *reduce* phases. SPARQL graph pattern matching is dominated by join operations, and is unlikely to be efficiently processed using existing techniques. This cost is prohibitive for RDF graph pattern matching queries which typically involve several join operations. In this paper, we propose an approach for optimizing graph pattern matching by reinterpreting certain join tree structures as grouping operations. This enables a greater degree of parallelism in join processing resulting in more "bushy" like query execution plans with fewer Map-Reduce cycles. This approach requires that the intermediate results are managed as sets of groups of triples or *TripleGroups*. We therefore propose a data model and algebra - *Nested TripleGroup Algebra* for capturing and manipulating TripleGroups. The relationship with the traditional relational style algebra used in Apache Pig is discussed. A comparative performance evaluation of the traditional Pig approach and RAPID+ (Pig extended with NTGA) for graph pattern matching queries on the BSBM benchmark dataset is presented. Results show up to 60% performance improvement of our approach over traditional Pig for some tasks.

Keywords: MapReduce, RDF graph pattern matching, optimization techniques.

1 Introduction

With the recent surge in the amount of RDF data, there is an increasing need for scalable and cost-effective techniques to exploit this data in decision-making tasks. MapReduce-based processing platforms are becoming the *de facto* standard for large scale analytical tasks. MapReduce-based systems have been explored for scalable graph pattern matching [1][2], reasoning [3], and indexing [4] of RDF graphs. In the MapReduce [5] programming model, users encode their tasks as *map* and *reduce* functions, which are executed in parallel on the Mappers and Reducers respectively. This two-phase computational model is associated with an inherent communication and I/O overhead due to the data transfer between the Mappers and the Reducers. Hadoop[1] based systems like Pig [6] and Hive [7] provide high-level query languages that improve usability and support automatic data flow optimization similar to database systems. However,

[1] http://hadoop.apache.org/core/

G. Antoniou et al. (Eds.): ESWC 2011, Part II, LNCS 6644, pp. 46–61, 2011.

most of these systems are targeted at structured relational data processing workloads that require relatively few numbers of join operations as stated in [6]. On the contrary, processing RDF query patterns typically require several join operations due to the fine-grained nature of RDF data model. Currently, Hadoop supports only *partition parallelism* in which a single operator executes on different partitions of data across the nodes. As a result, the existing Hadoop-based systems with the relational style join operators translate multi-join query plans into a linear execution plan with a sequence of multiple Map-Reduce (MR) cycles. This significantly increases the overall communication and I/O overhead involved in RDF graph processing on MapReduce platforms. Existing work [8][9] directed at uniprocessor architectures exploit the fact that joins presented in RDF graph pattern queries are often organized into star patterns. In this context, they prefer bushy query execution plans over linear ones for query processing. However, supporting bushy query execution plans in Hadoop based systems would require significant modification to the task scheduling infrastructure.

In this paper, we propose an approach for increasing the degree of parallelism by enabling some form of *inter-operator parallelism*. This allows us to "sneak in" bushy like query execution plans into Hadoop by interpreting star-joins as groups of triples or *TripleGroups*. We provide the foundations for supporting TripleGroups as first class citizens. We introduce an intermediate algebra called the *Nested TripleGroup Algebra* (NTGA) that consists of TripleGroup operators as alternatives to relational style operators. We also present a data representation format called the *RDFMap* that allows for a more easy-to-use and concise representation of intermediate query results than the existing format targeted at relational tuples. RDFMap aids in efficient management of schema-data associations, which is important while querying *schema-last* data models like RDF. Specifically, we propose the following:

- A TripleGroup data model and an intermediate algebra called *Nested TripleGroup Algebra* (NTGA), that leads to efficient representation and manipulation of RDF graphs.
- A compact data representation format (*RDFMap*) that supports efficient TripleGroup-based processing.
- An extension to Pig's computational infrastructure to support NTGA operators, and compilation of NTGA logical plans to MapReduce execution plans. Operator implementation strategies are integrated into Pig to minimize costs involved in RDF graph processing.
- A comparative performance evaluation of Pig and RAPID+ (Pig extended with NTGA operators) for graph pattern queries on a benchmark dataset is presented.

This paper is organized as follows: In section 2, we review the basics of RDF graph pattern matching, and the issues involved in processing such pattern queries in systems like Pig. We also summarize the optimization strategies presented in our previous work, which form a base for the algebra proposed in this paper. In section 3.1, we present the TripleGroup data model and the supported operations. In 3.2, we discuss the integration of NTGA operators into Pig. In section 4, we present the evaluation results comparing the performance of RAPID+ with the existing Pig implementation.

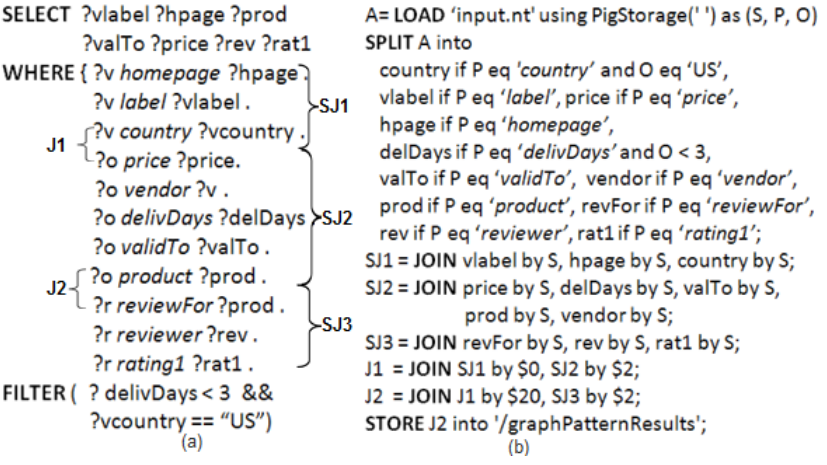


Fig. 1. Example pattern matching query in (a) SPARQL (b) Pig Latin (VP approach)

2 Background and Motivation

2.1 RDF Graph Pattern Matching

The standard query construct for RDF data models is a graph pattern which is equivalent to the select-project-join (SPJ) construct in SQL. A graph pattern is a set of triple patterns which are RDF triples with variables in either of the s, p, or o positions. Consider an example query on the BSBM[2] data graph to retrieve "*the details of US-based vendors who deliver products within three days, along with the review details for these products*". Fig. 1 (a) shows the corresponding SPARQL query, whose graph pattern can be factorized into two main components (i) three star-join structures ($SJ1$, $SJ2$, $SJ3$) describing resources of type Vendor, Offer, and Review respectively, two chain-join patterns ($J1$, $J2$) combining these star patterns and, (ii) the filter processing.

There are two main ways of processing RDF graph patterns depending on the storage model used: (i) triple model, or (ii) vertically partitioned (VP) storage model in which the triple relation is partitioned based on properties. In the former approach, RDF pattern matching queries can be processed as series of relational style self-joins on a large triple relation. Some systems use multi-indexing schemes [8] to counter this bottleneck. The VP approach results in a series of join operations but on smaller property-based relations. Another observation [8] is that graph pattern matching queries on RDF data often consist of multiple star-structured graph sub patterns. For example, 50% of the benchmark queries in BSBM have at least two or more star patterns. Existing work [8][9] optimize pattern matching by exploiting these star-structures to generate bushy query plans.

[2] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/`

2.2 Graph Pattern Matching in Apache Pig

Graph Pattern Matching in Pig. Map Reduce data processing platforms like Pig focus on ad hoc data processing in the cloud environment where the existence of pre-processed and suitably organized data cannot be presumed. Therefore, in the context of RDF graph pattern processing which is done directly from input documents, the VP approach with smaller relations is more suitable. To capture the VP storage model in Pig, an input triple relation needs to be "split" into property-based partitions using Pig Latin's `SPLIT` command. Then, the star-structured joins are achieved using an m-way `JOIN` operator, and chain joins are executed using th traditional binary `JOIN` operator. Fig. 1(b) shows how the graph pattern query in Fig. 1(a) can be expressed and processed in Pig Latin. Fig. 2 (a) shows the corresponding query plan for the VP approach. We refer to this sort of query plan as Pig's approach in the rest of the paper. Alternative plans may change the order of star-joins based on cost-based optimizations. However, that issue does not affect our discussion because the approaches compared in this paper all benefit similarly from such optimizations. Pig Latin queries are compiled into a sequence of Map-Reduce (MR) jobs that run over Hadoop. The Hadoop scheduling supports *partition parallelism* such that in every stage, one operator is running on different partitions of data at different nodes. This leads to a linear style physical execution plan. The above logical query plan will be compiled into a linear execution plan with a sequence of five MR cycles as shown in Fig. 2 (b). Each join step is executed as a separate MR job. However, Pig optimizes the multi-way join on the same column, and compiles it into a single MR cycle.

Issues. (i) Each MR cycle involves communication and I/O costs due to the data transfer between the Mappers and Reducers. Intermediate results are written to disk by Mappers after the *map* phase, which are read by Reducers and processed in the *reduce* phase after which the results are written to HDFS (Hadoop Distributed File System). These costs are summarized in Fig. 2 (b). Using this style of execution where join operations are executed in different MR cycles, join-intensive tasks like graph pattern matching will

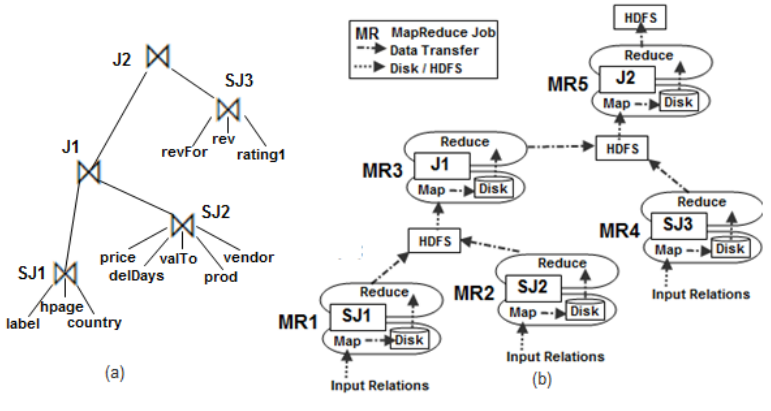


Fig. 2. Pattern Matching using VP approach (a) Query plan (b) Map-Reduce execution flow

result in significant I/O and communication overhead. There are other issues that contribute I/O costs e.g. the SPLIT operator for creating VP relations generates concurrent sub flows which compete for memory resources and is prone to disk spills. (ii) In imperative languages like Pig Latin, users need to explicitly manipulate the intermediate results. In *schema-last* data models like RDF, there is an increased burden due to the fact that users have to keep track of which columns of data are associated with which schema items (*properties*) as well as their corresponding values. For example, for the computation of join $J2$, the user needs to specify the join between intermediate relation $J1$ on the value of property type "product", and relation $SJ3$ on the value of property type "reviewFor". It is not straightforward for the user to determine that the value corresponding to property "product" is in column 20 of relation $J1$. In *schema-first* data models, users simply reference desired columns by attribute names.

TripleGroup-based Pattern Matching. In our previous work [10], we proposed an approach to exploit star sub patterns by re-interpreting star-joins using a *grouping-based join algorithm*. It can be observed that performing a group by Subject yields groups of tuples or *TripleGroups* that represent all the star sub graphs in the database. We can obtain all these star sub graphs using the relational style GROUP BY which executes in a *single* MR cycle, thus minimizing the overall I/O and communication overhead in RDF graph processing. Additionally, repeated data processing costs can be improved by *coalescing* operators in a manner analogous to "pushing select into cartesian product" in relational algebra to produce a more efficient operator. The empirical study in our previous work showed significant savings using this TripleGroup computational style, suggesting that it was worth further consideration.

In this paper, we present a generalization of this strategy by proposing an intermediate algebra based on the notion of TripleGroups. This provides a formal foundation to develop first-class operators with more precise semantics, to enable tighter integration into existing systems to support automatic optimization opportunities. Additionally, we propose a more suitable data representation format that aids in efficient and user-friendly management of intermediate results of operators in this algebra. We also show how this representation scheme can be used to implement our proposed operators.

3 Foundations

3.1 Data Model and Algebra

Nested TripleGroup Algebra (NTGA) is based on the notion of the *TripleGroup* data model which is formalized as follows:

Definition 1. (*TripleGroup*) 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. Fig. 3 (a) is an example of a *subject TripleGroup* which corresponds to a star sub graph. Our data model allows TripleGroups to be nested at the object component.

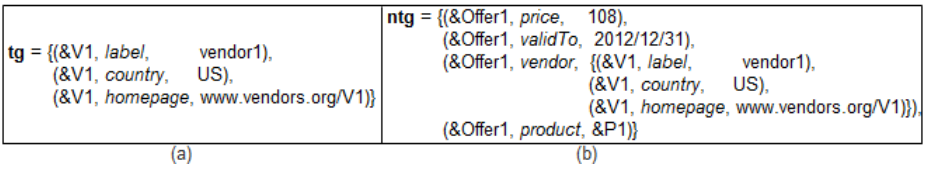


Fig. 3. (a) Subject TripleGroup tg (b) Nested TripleGroup ntg

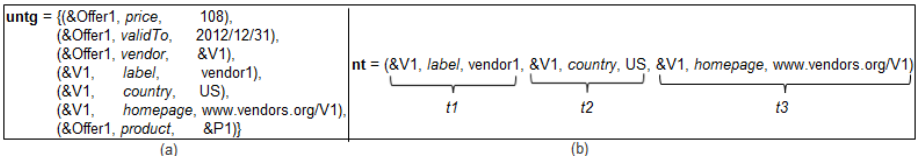


Fig. 4. (a) ntg.unnest() (b) n-tuple after tg.flatten()

Definition 2. (*Nested TripleGroup*) A nested TripleGroup ntg consists of a root TripleGroup ntg.root and one or more child TripleGroups returned by the function ntg.child() such that:
For each child TripleGroup $ctg \in ntg$.child(),

– $\exists\, t_1 \in ntg$.root, $t_2 \in ctg$ such that t_1.Object = t_2.

Nested TripleGroups capture the graph structure in an RDF model in a more natural manner. An example of a nested TripleGroup is shown in Fig. 3 (b). A nested TripleGroup can be "unnested" into a flat TripleGroup using the unnest operator. The definition is shown in Fig. 5. Fig. 4 (a) shows the TripleGroup resulting from the unnest operation on the nested TripleGroup in Fig. 3 (b). In addition, we define the flatten operation to generate an "equivalent" n-tuple for a given TripleGroup. For example, if $tg = t_1$, t_2,..., then the n-tuple tu has triple $t_1 = (s1, p1, o1)$ stored in the first three columns of tu, triple $t_2 = (s2, p2, o2)$ is stored in the fourth through sixth column, and so on. For convenience, we define the function triples() to extract the triples in a TripleGroup. For the TripleGroup in Fig. 3 (a), the flatten is computed as tg.triples($label$) ⋈ tg.triples($country$) ⋈ tg.triples($homepage$), resulting in an n-tuple as shown in Fig. 4 (b). It is easy to observe that the information content in both formats is equivalent. We refer to this kind of equivalence as *content equivalence* which we will denote as ≅. Consequently, computing query results in terms of TripleGroups is lossless in terms of information. This is specifically important in scenarios where TripleGroup-based processing is more efficient.

We define other TripleGroup functions as shown in Fig.5 (b). The *structure-labeling function* λ assigns each TripleGroup tg, with a label that is constructed as some function of tg.props(). Further, for two TripleGroups tg_1, tg_2 such that tg_1.props() ⊆ tg_2.props(), λ assigns labels such that $tg_1.\lambda() \subseteq tg_2.\lambda()$. The labeling function λ induces a partition on a set of TripleGroups based on the structure represented by the property types present in that TripleGroup. Each equivalence class in the partition consists of TripleGroups that have the exact same set of property types.

Next, we discuss some of the TripleGroup operators proj, filter, groupfilter, and join, which are formally defined in Fig.5 (c).

Symbol	Description
tg	TripleGroup
TG	Set of TripleGroups
tp	Triple pattern
ntg.root	Root of the nested TripleGroup
ntg.child()	Children of the nested TripleGroup
$?v_{tp}$	A variable in the triple pattern tp

(a)

Function	Returns
tg.props()	Union of all property types in tg
tg.triples()	Union of all triples in tg
tg.triples(p_i)	Triples in tg with property type p_i
$tg.\lambda()$	Structure label for tg based on tg.props()
$\delta(tp)$	A triple matching the triple pattern tp
$\delta(?v_{tp})$	A variable substituion in the triple matching tp

(b)

Operator	Definition
load($\{t_i\}$)	$\{ tg_i \mid tg_i = t_i$, and t_i is an input triple$\}$
proj$_{?v_{tp}}(TG)$	$\{\delta_i(?v_{tp}) \mid \delta_i(tp) \in tg_i, tg_i \in TG$ and $tp.\lambda() \subseteq tg_i.\lambda()\}$
filter$_{\Theta(?v_{tp})}(TG)$	$\{ tg_i \mid tg_i \in TG$ and $\exists\ \delta_i(tp) \in tg_i$ such that $\delta_i(?v_{tp})$ satisfies the filter condition $\Theta(?v_{tp})\}$
groupfilter(TG, P)	$\{ tg_i \mid tg_i \in TG$ and tg_i.props() = P $\}$
join($?v_{tp_x}$:TG_x, $?v_{tp_y}$:TG_y)	Assume $tg_x \in TG_x, tg_y \in TG_y, \exists\ \delta_1(tp_x) \in tg_x, \delta_2(tp_y) \in tg_y$, and $\delta_1(?v_{tp_x}) = \delta_2(?v_{tp_y})$ if O-S join, then $\{ntg_i \mid ntg_i$.root $= tg_x, \delta_1(tp_x)$.Object = $tg_y\}$ else $\{ tg_x \cup tg_y\}$
tg.flatten()	$\{tg$.triples(p_1) $\bowtie tg$.triples(p_2)...$\bowtie tg$.triples(p_n) where $p_i \in tg$.props()$\}$
ntg.unnest()	$\{ t_i \mid t_i$ is a non-nested triple in tg.root $\}$ $\cup \{ (s, p, s') \mid t' = (s, p, (s', p', o'))$ is a nested triple in tg.root$\}$ $\cup \{ ctg_i$.unnest() $\mid ctg_i \in tg$.child() $\}$

(c)

Fig. 5. NTGA Quick Reference (a) Symbols (b) Functions (c) Operators

(**proj**) The proj operator extracts from each TripleGroup, the required triple component from the triple matching the triple pattern. From our example data, proj$_{?hpage}(TG)$ =$\{www.vendors.org/V1\}$.

(**filter**) The filter operator is used for *value-based filtering* i.e. to check if the TripleGroups satisfy the given filter condition. For our example data, filter$_{price>500}$ (TG) would eliminate the TripleGroup ntg in Fig. 3 (b) since the triple ($\&Offer1$, $price$, 108) does not satisfy the filter condition.

(**groupfilter**) The groupfilter operation is used for *structure-based filtering* i.e. to retain only those TripleGroups that satisfy the required query sub structure. For example, the groupfilter operator can be used to eliminate TripleGroups like tg in Fig. 3 (a), that are structurally incomplete with respect to the equivalence class $TG_{\{label,country,homepage,mbox\}}$.

(**join**) The join expression join($?v_{tp_x}$:TG_x,$?v_{tp_y}$:TG_y) computes the join between a TripleGroup tg_x in equivalence class TG_x with a TripleGroup tg_y in equivalence class TG_y based on the given triple patterns. The triple patterns tp_x and tp_y share a common variable $?v$ at O or S component. The result of an object-subject (O-S) join is a nested TripleGroup in which tg_y is nested at the O component of the join triple in tg_x. For example, Fig. 6 shows the nested TripleGroup resulting from the join operation between equivalence classes $TG_{\{price,validTo,vendor,product\}}$ and $TG_{\{label,country,homepage\}}$ that join based on triple patterns $\{?o\ vendor\ ?v\}$ and $\{?v\ country\ ?vcountry\}$ respectively. For object-object (O-O) joins, the join operator computes a TripleGroup by union of triples in the individual TripleGroups.

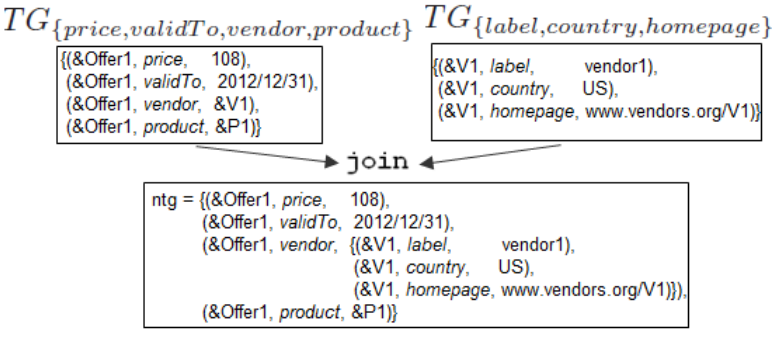


Fig. 6. Example join operation in NTGA

Execution plan using NTGA and its mapping to Relational Algebra. TripleGroup-based pattern matching for a query with n star sub patterns, compiles into a MapReduce flow with n MR cycles as shown in Fig. 7. The same query executes in double the number of MR cycles $(2n - 1)$ using Pig approach. Fig. 7 shows the equivalence between NTGA and relational algebra operators based on our notion of *content equivalence*. This mapping suggests rules for lossless transformation between queries written in relational algebra and NTGA. First, the input triples are loaded and the triples that are not part of the query pattern are filtered out. Pig `load` and `filter` operators are *coalesced* into a `loadFilter` operator to minimize costs of repeated data handling. The graph patterns are then evaluated using the NTGA operators, (i) star-joins using Pig's `GROUP BY` operator, which is *coalesced* with the NTGA `groupFilter` operator to enable *structure-based filtering* (represented as `StarGroupFilter`) and, (ii) chain joins on TripleGroups using the NTGA `join` operator (represented as `RDFJoin`). The final result can be converted back to n-tuples using the NTGA `flatten` operator. In general, TripleGroups resulting from any of the NTGA operations can be mapped to Pig's tupled results using the `flatten` operator. For example, the `StarGroupFilter` operation results in a set of TripleGroups. Each TripleGroup can be transformed to an equivalent n-tuple resulting from relational star-joins $SJ1$, $SJ2$, or $SJ3$.

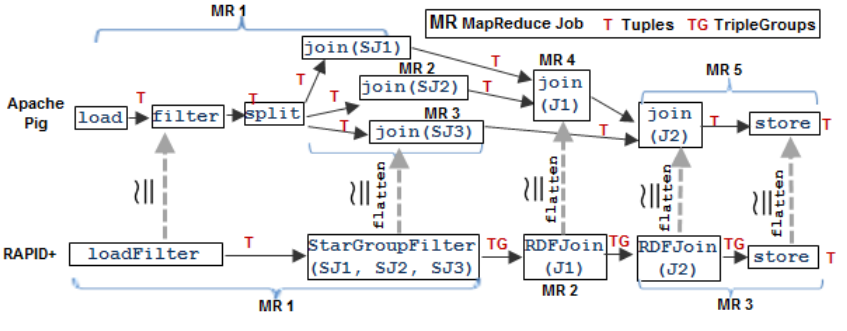


Fig. 7. NTGA execution plan and mapping to Relation Algebra

3.2 RAPID+: Integrating NTGA Operators into Pig

Data Structure for TripleGroups - RDFMap. Pig Latin data model supports a bag data structure that can be used to capture a TripleGroup. The Pig data bag is implemented as an array list of tuples and provides an iterator to process them. Consequently, implementing NTGA operators such as `filter`, `groupfilter`, `join` etc. using this data structure requires an iteration through the data bag which is expensive. For example, given a graph pattern with a set of triple patterns TP and a data graph represented as a set of TripleGroups TG, the `groupfilter` operator requires matching each triple pattern in TP with each tuple t in each TripleGroup $tg \in TG$. This results in the cost of the `groupfilter` operation being $O(|TP|*|tg|*|TG|)$. In addition, representing triples as 3-tuple (s, p, o) results in redundant $s(o)$ components for subject (object) TripleGroups. We propose a specialized data structure called *RDFMap* targeted at efficient implementation of NTGA operators. Specifically it enables, (i) efficient look-up of triples matching a given triple pattern, (ii) compact representation of intermediate results, and (iii) ability to represent structure-label information for TripleGroups. RDFMap is an extended HashMap that stores a mapping from property to object values. Since subject of triples in a TripleGroup are often repeated, RDFMap avoids this redundancy by using a single field Sub to represent the subject component. The field EC captures the structure-label (equivalence class mapped to numbers). Fig. 8. shows the RDFMap corresponding to the Subject TripleGroup in Fig. 3 (a). Using this representation model, a nested TripleGroup can be supported using a nested $propMap$ which contains another RDFMap as a value. The $propMap$ provides a property-based indexed structure that eliminates the need to iterate through the tuples in each bag. Since $propMap$ is hashed on the P component of the triples, matching a triple pattern inside a TripleGroup can now be computed in time $O(1)$. Hence, the cost of the `groupfilter` operation is reduced to $O(|P|*|TG|)$.

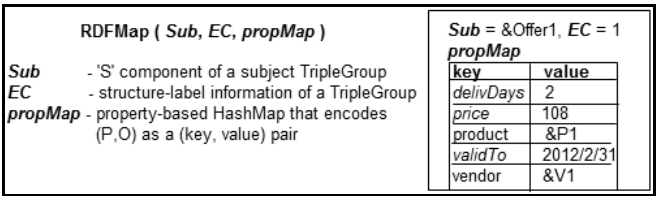


Fig. 8. RDFMap representing a subject TripleGroup

Implementing NTGA operators using RDFMap. In this section, we show how the property-based indexing scheme of an RDFMap can be exploited for efficient implementation of the NTGA operations. We then discuss the integration of NTGA operators into Pig.

`StarGroupFilter`. A common theme in our implementation is to *coalesce* operators where possible in order to minimize the costs of parameter passing, and context switching between methods. The `starGroupFilter` is one such operator, which coalesces

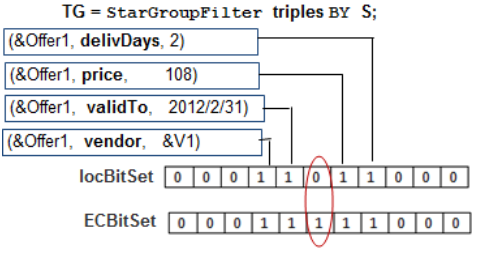
the NTGA `groupfilter` operator into Pig's relational `GROUP BY` operator. Creating subject TripleGroups using this operator can be expressed as:

$$TG = \texttt{StarGroupFilter}\ \text{triples by}\ S$$

The corresponding *map* and *reduce* functions for the `StarGroupFilter` operator are shown in Algorithm 1. In the *map* phase, the tuples are annotated based on the S component analogous to the *map* of a `GROUP BY` operator. In the *reduce* function, the different tuples sharing the same S component are packaged into an *RDFMap* that corresponds to a subject TripleGroup. The `groupfilter` operator is integrated for *structure-based filtering* based on the query sub structures (equivalence classes). This is achieved using global bit patterns (stored as BitSet) that concisely represent the property types in each equivalence class. As the tuples are processed in the *reduce* function, the local BitSet keeps track of the property types processed (line 6). After processing all tuples in a group, if the local BitSet (`locBitSet`) does not match the global BitSet (`ECBitSet`), the structure is incomplete and the group of tuples is eliminated (lines 10-11). Fig. 9 shows the mismatch between the `locBitSet` and `ECBitSet` in the sixth position that represents the missing property "product" belonging to the equivalence class $TG_{\{price,validTo,delivDays,vendor,product\}}$. If the bit patterns match, a labeled *RDFMap* is generated (line 13) whose $propMap$ contains all the (p,o) pairs representing the edges of the star sub graph. The output of `StarGroupFilter` is a single relation containing a list of RDFMaps corresponding to the different star sub graphs in the input data.

Algorithm 1. StarGroupFilter

```
1  Map (Tuple tup(s,p,o))
2  return (s,tup)
3  Reduce (Key, List of tuples T)
4  foreach tup(s,p,o) in T do
5  |   Initialize: Sub ← s; EC ← findEC(p)
6  |   locBitset.set(p)
7  |   propMap.put(p,o)
8  end foreach
9  ECBitSet ← global BitSet for EC
10 if locBitset != ECBitSet then
11 |   return null
12 else
13 |   return new RDFMap(Sub, EC, propMap)
14 end if
```



Algorithm 2. RDFJoin

```
1  Map (RDFMap rMap)
2  if joinKey == * then
3  |   return (rMap.Sub, rMap)
4  else
5  |   key ← rMap.propMap.get(joinKey)
6  |   return (key, rMap)
7  end if
8  Reduce (Key, List of RDFMaps R)
9  foreach rMap in R do
10 |   if rMap.EC == EC1 then
11 |   |   list1.add(rMap)
12 |   else if rMap.EC == EC2 then
13 |   |   list2.add(rMap)
14 |   end if
15 end foreach
16 foreach outer in list1 do
17 |   foreach inner in list2 do
18 |   |   propMapNew ←
   |   |   joinProp(outer.propMap,
   |   |   inner.propMap)
19 |   |   ECNew ← joinSub(outer.EC,
   |   |   inner.EC)
20 |   |   SubNew ← joinSub(outer.Sub,
   |   |   inner.Sub)
21 |   |   rMapNew ← new
   |   |   RDFMap(SubNew, ECNew,
   |   |   propMapNew)
22 |   |   resultList.add(rMapNew)
23 |   end foreach
24 end foreach
25 return resultList
```

Fig. 9. Structure-based filtering of TripleGroups

RDFJoin: The RDFJoin operator takes as input a single relation containing RDFMaps, and computes the NTGA join between star patterns as described by Algorithm 2. The O-S join $J1$ between the star patterns in Fig. 1 can be expressed as follows:

$$J1 = \texttt{RDFJoin}\ TG \text{ on } (1{:}'vendor', 0{:}*);$$

where joins on O are specified using the property types, and joins on S are specified as a '*'. In the *map* phase, the RDFMaps are annotated based on the join key corresponding to their equivalence class (lines 2-8). In the *reduce* phase, the RDFMaps that join are packaged into a new RDFMap which corresponds to a nested TripleGroup. The EC of the new joined RDFMap is a function of the EC of the individual RDFMaps. For example, the RDFJoin between TripleGroups shown in Fig. 6, results in an RDFMap whose $propMap$ contains the union of triples from the individual TripleGroups as shown in Fig. 10. In our implementation, the Sub field is a concatenation of the Sub fields of the individual TripleGroups e.g. $\&Offer1.\&V1$. The join result is optimized by eliminating the (p, o) pair corresponding to the join triple if it is no longer required. This reduces the size of the intermediate RDFMaps after each MR cycle. Our join result corresponds to an unnested joined TripleGroup, as shown in Fig. 4 (a).

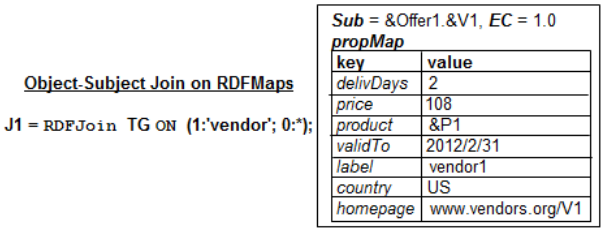


Fig. 10. Example RDFMap after RDFJoin operation

4 Evaluation

Our goal was to empirically evaluate the performance of NTGA operators with respect to pattern matching queries involving combinations of star and chain joins. We compared the performance of RAPID+ with two implementations of Pig, (i) the naive Pig with the VP storage model, and (ii) an optimized implementation of Pig (Pig_{opt}), in which we introduced additional project operations to eliminate the redundant join columns. Our evaluation tasks included, (i) **Task1 -** Scalability of TripleGroup-based approach with size of RDF graphs, (ii) **Task2 -** Scalability of TripleGroup-based pattern matching with denser star patterns, and (iii) **Task3 -** Scalability of NTGA operators with increasing cluster sizes.

Table 1. Testbed queries and performance gain of RAPID+ over Pig (10-node cluster / 32GB)

Query	#Triple Pattern	#Edges in Stars	%gain	Query	#Triple Pattern	#Edges in Stars	%gain
Q1	3	1:2	56.8	Q6	8	4:4	58.4
Q2	4	2:2	46.7	Q7	9	5:4	58.6
Q3	5	2:3	47.8	Q8	10	6:4	57.3
Q4	6	3:3	51.6	2S1C	6	2:4	65.4
Q5	7	3:4	57.4	3S2C	10	2:4:4	61.5

4.1 Setup

Environment: The experiments were conducted on VCL[3], an on-demand computing and service-oriented technology that provides remote access to virtualized resources. Nodes in the clusters had minimum specifications of single or duo core Intel X86 machines with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. The experiments were conducted on 5-node clusters with block size set to 256MB. Scalability testing was done on clusters with 10, 15, 20, and 25 nodes. Pig release 0.7.0 and Hadoop 0.20 were used. All results recorded were averaged over three trials.

Testbed - Dataset and Queries: Synthetic datasets (n-triple format) generated using the BSBM tool were used. A comparative evaluation was carried out based on size of data ranging from 8.6GB (approx. 35 million triples) at the lower end, to a data size of 40GB (approx. 175 million triples). 10 queries (shown in Table 1) adapted from the BSBM benchmark (Explore use case) with at least a star and chain join were used. The evaluation tested the effect of query structure on performance with, (i) $Q1$ to $Q8$ consisting of two star patterns with varying cardinality, (ii) $2S1C$ consisting of two star patterns, a chain join, and a filter component (6 triple patterns), and (ii) $3S2C$ consisting of three star patterns, two chain joins, and a filter component (10 triple patterns). Query details and additional experiment results are available on the project website[4].

4.2 Experiment Results

Task1: Fig. 11 (a) shows the execution times of the three approaches on a 5-node cluster for $2S1C$. For all the four data sizes, we see a good percentage improvement in the execution times for RAPID+. The two star patterns in $2S1C$ are computed in two separate MR cycles in both the Pig approaches, resulting in the query compiling into a total of three MR cycles. However, RAPID+ benefits by the grouping-based join algorithm (`StarGroupFilter` operator) that computes the star patterns in a single MR cycle, thus reducing one MR cycle in total. We also observe cost savings due to the integration of `loadFilter` operator in RAPID+ that coalesces the `LOAD` and `FILTER` phases. As expected, the Pig_{opt} performs better than the naive Pig approach due to the decrease in the size of the intermediate results.

[3] `https://vcl.ncsu.edu/`

[4] `http://research.csc.ncsu.edu/coul/RAPID/ESWC_exp.htm`

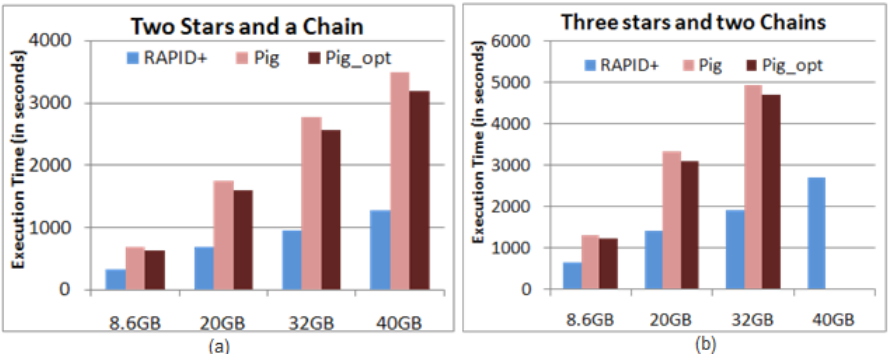


Fig. 11. Cost analysis on 5-node cluster for (a) 2SIC (b) 3S2C

Fig. 11 (b) shows the performance comparison of the three approaches on a 5-node cluster for $3S2C$. This query compiles into three MR cycles in RAPID+ and five MR cycles in Pig / Pig_{opt}. We see similar results with RAPID+ outperformed the Pig based approaches, achieving up to 60% performance gain with the 32GB dataset. The Pig based approaches did not complete execution for the input data size of 40GB. We suspect that this was due to the large sizes of intermediate results. In this situation, the compact representation format offered by the RDFMap proved advantageous to the RAPID+ approach. In the current implementation, RAPID+ has the overhead that the computation of the star patterns results in a single relation containing TripleGroups belonging to different equivalence classes. In our future work, we will investigate techniques for delineating different types of intermediate results.

Task2: Table 1 summarizes the performance of RAPID+ and Pig for star-join queries with varying edges in each star sub graph. NTGA operators achieve a performance gain of 47% with $Q2$ (2:2 cardinality) which increases with denser star patterns, reaching 59% with $Q8$ (6:4 cardinality). In addition to the savings in MR cycle in RAPID+, this demonstrates the cost savings due to smaller intermediate relations achieved by eliminating redundant subject values and join triples that are no longer required. Fig. 12 (b) shows a comparison on a 5-node cluster (20GB data size) with Pig_{opt} which eliminates join column redundancy in Pig, similar to RDFMap's concise representation of subjects within a TripleGroup. RAPID+ maintains a consistent performance gain of 50% across the varying density of the two star patterns.

Task3: Fig. 12(a) shows the scalability study of $3S2C$ on different sized clusters, for 32GB data. RAPID+ starts with a performance gain of about 56% with the 10-node cluster, but its advantage over Pig and Pig_{opt} reduces with increasing number of nodes. The increase in the number of nodes, decreases the size of data processed by each node, therefore reducing the probability of disk spills with the `SPLIT` operator in the Pig based approaches. However, RAPID+ still consistently outperforms the Pig based approaches with at least 45% performance gain in all experiments.

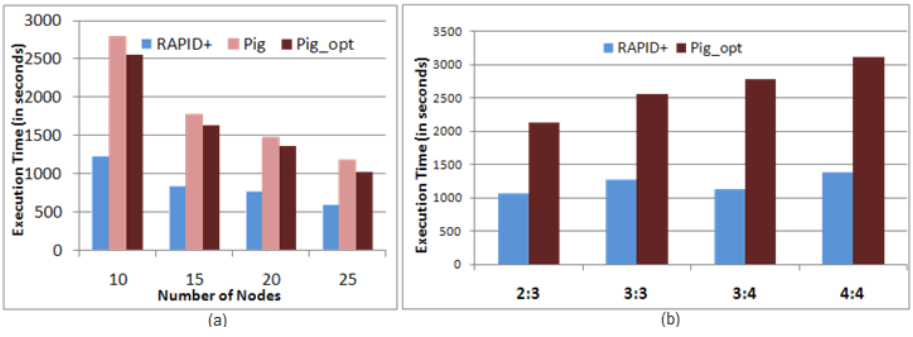


Fig. 12. Scalability study for (a) 3S2C varying cluster sizes (b) two stars with varying cardinality

5 Related Work

Data Models and High-Level Languages for cluster-based environment. There has been a recent proliferation of data flow language such as Sawzall [11], DryadLINQ [12], HiveQL [7], and Pig Latin [6] for processing structured data on parallel data processing systems such as Hadoop. Another such query language, JAQL[5] is designed for semi-structure data analytics, and uses the (key, value) JSON model. However, this model splits RDF sub graphs into different bags, and may not be efficient to execute bushy plans. Our previous work, RAPID [13] focused on optimizing analytical processing of RDF data on Pig. RAPID+ [10] extended Pig with UDFs to enable TripleGroup-based processing. In this work, we provide formal semantics to integrate TripleGroups as first-class citizens, and present operators for graph pattern matching.

RDF Data processing on MapReduce Platforms. MapReduce framework has been explored for scalable processing of Semantic Web data. For reasoning tasks, specialized map and reduce functions have been defined based on RDFS rules [3] and the OWL Horst rules [14], for materializing the closure of RDF graphs. Yet another work [15] extends Pig by integrating schema-aware RDF data loader and embedding reasoning support into the existing framework. For scalable pattern matching queries, there have been MapReduce-based storage and query systems [2],[1] that process RDFMolecules. Also, [16] uses HadoopDB [17] with a column-oriented database to support a scalable Semantic Web application. This framework enables parallel computation of star-joins if the data is partitioned based on the Subject component. However, graph pattern queries with multiple star patterns and chain join may not benefit much. Another recent framework [4] pre-processes RDF triples to enable efficient querying of billions of triples over HDFS. We focus on ad hoc processing of RDF graphs that cannot presume pre-processed or indexed data.

Optimizing Multi-way Joins. RDF graph pattern matching typically involves several join operations. There have been optimization techniques [9] to re-write SPARQL queries into small-sized star-shaped groups and generate bushy plans using two physical join operators called njoin and gjoin. It is similar in spirit to the work presented

[5] `http://code.google.com/p/jaql`

here since both exploit star-shaped sub patterns. However, our work focuses on parallel platforms and uses a grouping-based algorithm to evaluate star-joins. There has been work on optimizing m-way joins on structured relations like slice join [18]. However, we focus on joins involving RDF triples for semi-structured data. Another approach [19] efficiently partitions and replicates of tuples on reducer processes in a way that minimizes the communication cost. This is complementary to our approach and the partitioning schemes could further improve the performance of join operations. [20], investigates several join algorithms which leverage pre-processing techniques on Hadoop, but mainly focus on log processing. RDFBroker [21] is a RDF store that is based on the concept of a *signature* (set of properties of a resource), similar to NTGA's *structure-labeling* function λ. However, the focus of [21] is to provide a natural way to map RDF data to database tables, without presuming schema knowledge. Pregel [22] and Signal/Collect [23] provide graph-oriented primitives as opposed to relational algebra type operators, and also target parallel platforms. The latter is still in a preliminary stage and has not completely demonstrated its advantages across parallel platforms.

6 Conclusion

In this paper, we presented an intermediate algebra (NTGA) that enables more natural and efficient processing for graph pattern queries on RDF data. We proposed a new data representation format (RDFMap) that supports NTGA operations in a more efficient manner. We integrated these NTGA operators into Pig, and presented a comparative performance evaluation with the existing Pig implementation. For certain classes of queries, we saw a performance gain of up to 60%. However, there might be certain scenarios in which it may be preferable not to compute all star patterns. In such cases, we need a hybrid approach that utilizes cost-based optimization techniques to determine when the NTGA approach is the best. We will also investigate a more efficient method for dealing with heterogeneous TripleGroups resulting from join operations.

References

1. Newman, A., Li, Y.F., Hunter, J.: Scalable Semantics: The Silver Lining of Cloud Computing. In: IEEE International Conference on eScience (2008)
2. Newman, A., Hunter, J., Li, Y., Bouton, C., Davis, M.: A Scale-Out Rdf Molecule Store for Distributed Processing of Biomedical Data. In: Semantic Web for Health Care and Life Sciences Workshop (2008)
3. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning Using MapReduce. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 634–649. Springer, Heidelberg (2009)
4. Husain, M., Khan, L., Kantarcioglu, M., Thuraisingham, B.: Data Intensive Query Processing for Large Rdf Graphs Using Cloud Computing Tools. In: IEEE International Conference on Cloud Computing, CLOUD (2010)
5. Dean, J., Ghemawat, S.: Simplified Data Processing on Large Clusters. ACM Commun. 51, 107–113 (2008)
6. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: Proc. International Conference on Management of data (2008)

7. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: A Warehousing Solution over a Map-Reduce Framework. Proc. VLDB Endow. 2, 1626–1629 (2009)
8. Neumann, T., Weikum, G.: The Rdf-3X Engine for Scalable Management of Rdf Data. The VLDB Journal 19, 91–113 (2010)
9. Vidal, M.-E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently Joining Group Patterns in SPARQL Queries. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 228–242. Springer, Heidelberg (2010)
10. Ravindra, P., Deshpande, V.V., Anyanwu, K.: Towards Scalable Rdf Graph Analytics on Mapreduce. In: Proc. Workshop on Massive Data Analytics on the Cloud (2010)
11. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall. Sci. Program. 13, 277–298 (2005)
12. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P.K., Currey, J.: Dryadlinq: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In: Proc. USENIX Conference on Operating Systems Design and Implementation (2008)
13. Sridhar, R., Ravindra, P., Anyanwu, K.: RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 715–730. Springer, Heidelberg (2009)
14. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL Reasoning with Webpie: Calculating the Closure of 100 Billion Triples. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 213–227. Springer, Heidelberg (2010)
15. Tanimura, Y., Matono, A., Lynden, S., Kojima, I.: Extensions to the Pig Data Processing Platform for Scalable Rdf Data Processing using Hadoop. In: IEEE International Conference on Data Engineering Workshops (2010)
16. Abouzied, A., Bajda-Pawlikowski, K., Huang, J., Abadi, D.J., Silberschatz, A.: Hadoopdb in Action: Building Real World Applications. In: Proc. International Conference on Management of data (2010)
17. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: Hadoopdb: an Architectural Hybrid of Mapreduce and Dbms Technologies for Analytical Workloads. Proc. VLDB Endow. 2, 922–933 (2009)
18. Lawrence, R.: Using Slice Join for Efficient Evaluation of Multi-Way Joins. Data Knowl. Eng. 67, 118–139 (2008)
19. Afrati, F.N., Ullman, J.D.: Optimizing Joins in a Map-Reduce Environment. In: Proc. International Conference on Extending Database Technology (2010)
20. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in Mapreduce. In: Proc. International Conference on Management of data (2010)
21. Sintek, M., Kiesel, M.: RDFBroker: A Signature-Based High-Performance RDF Store. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 363–377. Springer, Heidelberg (2006)
22. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-Scale Graph Processing. In: Proc. International Conference on Management of data (2010)
23. Stutz, P., Bernstein, A., Cohen, W.: Signal/Collect: Graph Algorithms for the (Semantic) Web. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 764–780. Springer, Heidelberg (2010)