To Nest or Not to Nest, When and How Much: Representing Intermediate Results of Graph Pattern Queries in MapReduce Based Processing

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

ABSTRACT
Many queries on RDF datasets involve triple patterns whose properties are multi-valued. When processing such queries using flat data models and their associated algebras, intermediate results could contain a lot of redundancy. In the context of processing using MapReduce based platforms such as Hadoop, such redundancy could account for a non-trivial proportion of overall disk I/O, sorting and network data transfer costs. Further, when MapReduce workflows consist of multiple cycles as is typical when processing RDF graph pattern queries, these costs could compound over multiple cycles. However, it may be possible to avoid such overhead if nested data models and algebras are used.

In this short paper, we present some on-going research into the use of a nested TripleGroup data model and Algebra (NTGA) for MapReduce based RDF graph processing. The NTGA operators fully subscribe to the NTG data model. This is in contrast to systems such as Pig where the data model supports some nesting but the algebra is primarily tuple based (requiring the flattening of nested objects before other operators can be applied). This full subscription to the nested data model by NTGA also enables support for different unnesting strategies including delayed and partial unnesting. We present a preliminary evaluation of these strategies for efficient management of multi-valued properties while processing graph pattern queries in Apache Pig.

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

General Terms
Algorithms, Performance, Languages

Keywords
Graph pattern queries, MapReduce, Multi-valued properties

1. INTRODUCTION
Recent surge in the amount of RDF [10] data has led to an increased interest in using MapReduce [5] based platforms for processing. In addition to the typical challenges in processing large data, RDF data processing workloads are join-intensive. This is due to the fine-grained nature of data modeled in RDF. RDF models data as triples of the form (Subject, Property, Object) where a Property defines a binary relationship between resources (denoted with a leading ‘&’) or between resources and literals (attributes). For example, the triple (&Prod1, producer, &pc1) states that the resource &Prod1 has a producer &pc1. A collection of RDF triples (an RDF database) can also be modeled as a directed labeled graph with nodes representing Subjects and Objects, and labeled edges denoting Property names. The foundational construct for querying RDF graph data is a triple pattern, which is a triple with a variable (denoted with a leading '?') in any of the Subject, Property or Object positions. For example, the triple pattern (?prodF, pfLabel, ?label) matches all triples with Property pfLabel, whose Subjects and Objects are considered as variable bindings for ?prodF and ?label respectively. Typical queries involve a combination of triple patterns called a graph pattern. The answer to a graph pattern consists of bindings that match all triple patterns. Figure 1 (c) shows an example graph pattern query with 6 triple patterns, to retrieve details about Products and their features.

RDF data is commonly stored as ternary relations and query evaluation is achieved using several relational style joins (shared variables across triple patterns denote equi-joins). Our example query can be expressed as a relational query with 5 self-join operations. Such join-intensive workflows in MapReduce based systems such as Hadoop [24], Apache Pig [19] and Hive [21] often result in lengthy execution workflows with multiple MapReduce cycles. The challenge of such lengthy workflows is the cost associated with each MapReduce cycle - I/O costs (read and writes to the local and distributed file system), sorting (map-to-reducer data transfer) costs. Recent efforts have focused on techniques to shorten MapReduce execution workflows [4][8][20][23].

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SWIM 2012, May 20, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1446-6/20/05...$10.00.

![Figure 1](image-url)
Another important but less studied problem in the context of MapReduce based processing of RDF graph patterns is that of efficient management of multi-valued properties. *Multi-valued properties* (MVP) are a common occurrence in real world datasets including RDF datasets. Common multi-valued relationships such as *citation references* and *Facebook friends* are captured in bibliographic datasets and social networks respectively. Our example graph pattern query in Figure 1 (c) includes an MVP *prodFeature* with multiplicity 19 (each *Product* is associated with 19 *ProductFeatures*). The results of such queries involving MVPs computed using an unnested model and algebra looks as shown in Figure 1 (a). It can be observed that there is redundancy due to the repetition of the values in the non-MVP columns, for each distinct Object of the MVP. Such redundancies increase the overhead in lengthy MR workflows with multiple phases of I/O, sorting and communication costs.

In this paper, we focus on minimizing the overhead due to MVPs while processing graph pattern queries on MapReduce. Our approach is based on using a nested data model to avoid the redundancy in data representation. In contrast to nested data models supported by systems such as Pig, our approach does not require early explicit unnesting prior to joins on nested columns. We build on the advantages of our previously proposed Nested TripleGroup data model and algebra (*NTGA*) [9][20] that groups star-join computations into a single MR cycle, thus reducing the required number of MR cycles for processing graph pattern queries. Specifically, we make the following contributions:

(i) We propose *delayed* and *partial unnesting* strategies to reduce the redundancy factor in intermediate results while processing star subpatterns containing MVPs.

(ii) We present preliminary evaluation of the proposed unnesting strategies for joins on non-MVP and MVP components with varying multiplicity of MVPs for RDF graph pattern queries on a real-world and synthetic benchmark dataset.

The rest of the paper is organized as follows: Section 2 provides a background on MapReduce based RDF graph processing using the relational and NTGA based models. Section 0 describes three possible unnesting strategies for efficient management of MVPs, followed by the case study results in Section 5.

### 2. Preliminaries

Graph pattern queries could either be basic graph patterns (involving only joins) or could be extended with other types of clauses to yield other kinds of graph patterns such as filtered or optional graph patterns. In this section, we focus primarily on basic graph patterns. Assume we have a basic graph pattern query \( GQ = (P_1, P_2, P_3...P_l) \) that defines \( l - 1 \) join operations between the set of properties \( P_1, ... P_l \). A MapReduce execution plan \( MP \) is a sequence of MapReduce cycles \( MR_1, MR_2, ... MR_n \); \( t \leq (l - 1) \) and an assignment \( \mu \) such that \( \mu(P_i, P_j) = k, 1 \leq k \leq t \), indicating that the join of \( P_i \) and \( P_j \) will be executed in the \( k^{th} \) MR cycle. In the following section, we discuss two approaches for evaluating basic graph pattern queries on MapReduce in the context of the Apache Pig system. We assume that the input dataset is not pre-processed i.e. consists of a heterogeneous (different properties) set of triples.

### 2.1 Processing Graph Pattern Queries using Relational Joins in MapReduce

In the MapReduce programming paradigm, tasks are specified as *map()* and *reduce()* functions which are executed in parallel across a cluster of slave nodes (‘m’ mappers and ‘r’ reducers respectively). To process a relational style \( \text{JOIN} \) operation on two relations \( A \) and \( B \) on columns \( a \) and \( b \) respectively, the *map()* function for the \( \text{JOIN} \) operation tags each tuple from \( A \) and \( B \) with the column value for \( a \) and \( b \) respectively. Each map output tuple is mapped to one of the \( r \) partitions, sorted based on its join key, and written into the local disk. The \( r \) partitions or buckets corresponding to the \( r \) reducers constitute the *Reducer Space*. All the records with the same join key are assigned and transferred to the same reducer. In each reducer, the tuples are further grouped based on the join key and the reduce function is invoked for each such group. The \( n \) reduce function groups inside each reducer bucket constitute the *Reducer Function Space*. We refer to the \( i^{th} \) bucket in the *Reducer Space* as \( \text{Reducer}_bkt_i \) and the \( j^{th} \) bucket in its *Reducer Function Space* as \( rf_bkt_{i,j} \). The default partitioning scheme in Hadoop is based on direct hashing of map output keys to the Reducer Space. An intermediate key \( k_l \) is assigned to a *Reducer_bkt* based on the following function:

\[
\text{partition}(k_l) \rightarrow \text{hash}(k_l) \mod r
\]

In effect, an intermediate key-value pair \((k_1, v_1)\) is assigned to \( \text{Reducer}_bkt_{(\text{hash}(k_1) \mod n)} \) \( \neq r \). The mapping to the *Reducer Function Space* is based on the value of the original key \( k_l \). Thus, the tuples in the bucket \( rf_bkt_{i,j} \) correspond to tuples from \( A \) and \( B \) with the same join key, and their join is processed by the same reducer().

The output of the reduce phase is written onto the Hadoop Distributed File System - HDFS, n-way joins between multiple relations that join on equivalent join attributes (star-joins) can be processed similarly in a single MR cycle. In general, the total number of reduce function groups is equal to the number of distinct join keys and the number of tuples in each \( rf_bkt \) depends on the join cardinality.

In order to reduce the overhead of self-joins on large RDF relations, the RDF triples can be vertically partitioned [1] (VP) into smaller relations based on their Property types. Only partitions that are required for the specified join are processed resulting in I/O savings. The VP approach can be adopted in Pig using the SPLIT operator and the \( \text{JOIN} \) operations process the SPLIT output. Our example graph pattern in Figure 1 (c) can be decomposed into two star patterns \( SJ1 \) and \( SJ2 \) respectively, each of which can be expressed as an n-way *JOIN* operation and can be computed in a single MR cycle. Hence the two star patterns \( SJ1 \) and \( SJ2 \) can be computed in 2 MR cycles (MR1 and MR2 respectively), followed by another MR cycle (MR3) to join the star patterns. We refer to this approach as the Pig approach in the rest of the paper.

#### 2.1.1 Redundancy Factor of Multi-valued Properties

Let \( P_i \) be an MVP in \( GQ \) with multiplicity \( M \) and assume that the execution plan for \( GQ \) assigns the join with relation \( P_i \) to MR cycle \( MR_j \) i.e. \( P_i \text{ JOIN } P_j \) \( \text{JOIN} P_{j+1} \text{ JOIN } P_{j+2} \) \( \text{JOIN} \) etc. \( P_{j+2} \). Recall that the result of this operation denoted as \( O \) will be written to HDFS after the reduce phase and will be read in by the map phase of a subsequent MR cycle. Let each tuple in \( O \) have columns \( (A_{11}, A_{12}, A_{13}, A_{22}, A_{23}, ... A_{i1}, A_{i2}, A_{i3}) \) where \( A_{11}, A_{12} \), and \( A_{13} \) represent the Subject, Property and Object components of Property \( P_i \) respectively. The tuples in \( O \) can be partitioned into equivalence classes \( [C_1], [C_2] ... [C_k] \) such that:

- All tuples in \( C_k \) agree on the non-MVP components i.e. \( (A_{11}, A_{12}, A_{13}, ... A_{i1}, A_{i2}, A_{i3}) \) but differ in the column values \( (A_{11}, A_{12}, A_{13}) \) corresponding to MVP \( P_i \), and

- \( F \) is the number of distinct values of \( P_{ij} \), the join column of \( P_i \) used for the current join operation.

In our example data in Figure 1 (a), let us assume that the number of distinct Products is 10 \( (F=10) \). The 4 tuples corresponding to
Subject &Prod1 belong to the same partition Ck where the non-MVP components (Subject, Predicate, Object of all Properties except prodFeature) repeat in each of the 4 tuples. An estimate of the amount of redundant data RD written to disk at the end of the reduce phase for MRk is:

$$RD = \sum_{i=1}^{F} \sum_{f \in Ck} \sum_{x,y \in A_{i}} b(t(A_{xy}))$$

where $b(t(A_{xy}))$ is the size of a column $A_{xy}$. Essentially, this aggregates the size of redundant non-MVP components of all the tuples in $O_i$. We define the redundancy factor $RedF_k$ of the output of cycle MRk as:

$$RedF_k = \frac{RD}{|O_i|}$$

where $|O_i|$ is the sum of the sizes of all columns in all the tuples in $O_i$. RedFk represents the proportion of redundant data written to disk. This is also equivalent to the redundancy factor of the input of any subsequent map phase of a cycle MRk, $k > j$ that reads the output of MRj (represents the proportion of redundant data reads in cycle MRk). The output of MRk has redundancy factor $RedF_k$ as a function of $(RedF_j \ast RF_j)$ where RFj is the reduction factor of cycle MRj which represents the reduction in the size of data due to the selectivity of the operations assigned to cycle MRj. The redundancy factors for cycles that depend on MRk can be defined recursively.

The recursive nature of the redundancy factor shows the ripple effect of redundancy on the costs of reads, writes, sorting, and data transfer in the subsequent MR cycles in the dataflow. This estimation model suggests two things, (i) it is best to avoid large redundancy factors, and (ii) in scenarios where redundancy cannot be avoided, it is best to shorten the length of its ripple effect. Our approach for addressing this problem is twofold: (i) to use a nested data model as our data representation format, and (ii) design different strategies for shortening the ripple effect by using operators that allow delayed and partial unnesting strategies. Our approach builds on our previously proposed nested TripleGroup data model and algebra (NTGA) which we describe next.

### 2.2 Graph Pattern Matching using NTGA on MapReduce

NTGA exploits the fact that star-join structures (pattern with common Subject variable) commonly occur in graph pattern queries. Using the NTGA approach, all star-join subpatterns in a graph pattern query are translated into a single GROUP BY operation on the Subject column. The result of this operation is a set of “groups of triples” or triplegroups that agree on the Subject column. For example, tg1 in Figure 2 is a Subject triplegroup sharing common Subject &Prod1. Each triplegroup is an instance of some star subpattern in the query (tg1 is an instance of SJ1 in our example query). Further, the triplegroups are partitioned into equivalence classes based on their structure e.g. tg1 belongs to the equivalence class TG(pLabel, producer, prodFeature, pProp).

The advantage of the GROUP BY approach is that n star joins in a graph pattern can be computed in a single MR cycle as opposed to n MR cycles using relational joins. For our example query, both the star patterns SJ1 and SJ2 are computed in a single MR cycle (MR1) using NTGA. One important difference is that the result of a relational star-join is a set of n-tuples, each containing the components of the participating relations. The grouping based star-join computation results in a set of groups of triples, each representing a component of the participating relations. The difference in result structure in the grouping approach necessitates the need for additional operations that understand this. The NTGA includes specialized triplegroup based operators. Most relevant operation for our discussion is the “join” operation called the TG_Join which is used to join the star structures together. The TG_Join operator is semantically equivalent to the relational join operator except that it is defined on triplegroups. The object-subject join operation between the triplegroup classes TG(plabel, producer, prodFeature, pProp) (corresponding to star-join SJ1) and TG(pType, pLabel) (corresponding to star-join SJ2) is represented in Figure 2.

$$tg1 = TG(pLabel, producer, prodFeature, pProp) \quad tg2 = TG(pType, pLabel)$$

The TG_Join results in a nested triplegroup ntg whose root is the triplegroup tg1 and the child triplegroup is tg2. The example pattern matching query in Figure 1 (c) can be executed in 2 MR cycles using the NTGA approach as opposed to 3 MR cycles using the relational approach in Pig. In general, for a query with n star sub patterns, the MapReduce workflow generated from an NTGA based plan is n cycles vs. (2n -1) cycles for the relational approach. Previous experimental results showed performance gains of up to 60% for some classes of queries. Additionally, NTGA includes two filtering operators, (i) TG_Filter eliminates irrelevant triplegroups that fail to satisfy the FILTER constructs i.e. value-based filtering, and (ii) TG_GroupFilter eliminates triplegroups that do not satisfy structural constraints as defined by a triplegroup class such as TG(plabel, producer, prodFeature, pProp). Details of the data model and algebra can be found in [20].

### 3. Unnesting Strategies for Efficient Management of Multi-valued Properties

Three possible unnesting strategies can be applied to minimize the redundancy factor in intermediate results while processing graph pattern queries containing MVPs, (i) early complete unnesting in the reduce phase of the star-join cycle, (ii) delayed complete unnesting in the map phase of the join between stars, and (iii) delayed partial unnesting in the map phase of the join between stars. Figure 3 represents an overview of the three strategies and their corresponding query plans for our example query in Figure 1 (c). While the first strategy (Figure 3 (a)) is based on the Pig approach, the other two are NTGA based plans. The figure also denotes the stages at which the intermediate results containing an MVP are nested and unnested (flattened). For the rest of this discussion, we use Star-MVP to denote a star subgraph containing an MVP (result of SJ1 in example query) and MVJoin to denote a join on the object value of an MVP (join between SJ1 and SJ2).
do not require unnesting before operations such as join. This
resultant tuples and hence does not affect
unnest only the non-MVP column which does not incre ase the
columns, where each column is a bag containing corre sponding
tuples from the participating relations as represented in Figure 4.

Full Replication

Object of
processing any subsequent join operation requires u nnesting (or
flattening in Pig Latin parlance) of the join column. In the case of
Pig’s support for a nested data model can be exploi ted to
eliminate data redundancy while representing Star-MVPs. This
can be achieved by processing star-joins as a COGROUP (grouping of
multiple relations) on the vertically partitioned relations based on the Subject column as follows:

COGROUP pLabel by Sub, pProp by Sub… prodFeature by Sub;

A COGROUP on N relations, results in a nested tuple with N
columns, where each column is a bag containing corresponding
tuples from the participating relations as represented in Figure 4.

\[
\{\{&Prod1, pLabel, Product\}\}, \ldots, \{\ldots pProp, \ldots\}\}
\]

Figure 4: Nested tuple resulting from the COGROUP operation on vertically partitioned relations

The COGROUP based approach for star-join computation of Star-MVPs has no redundant non-MVP components and hence
minimizes the redundancy factor for MR_{J1} (RedF_{J1}=0). The reduced RedF_{J1} results in less amount of HDFS writes in MR_{J1}. However, each “column” in the result of a COGROUP is a bag, and Pig’s JOIN operator is not defined on nested columns. Hence, processing any subsequent join operation requires unnesting (or flattening in Pig Latin parlance) of the join column. In the case of a join operation on any of the non-MVP components such as Object of pLabel, we may partially unnest the nested tuple i.e. unnest only the non-MVP column which does not increase the resultant tuples and hence does not affect RedF_{J1}. However, the case of an MVJoin requires complete unnesting of the tuples based on the MVP column, resulting in redundant non-MVP components in the intermediate tuples, similar to the Pig approach (Figure 1 (a)). Both partial and complete unnesting can be achieved using the FLATTEN operator at the end of the reduce phase of the COGROUP as shown in Figure 3 (a). The complete unnesting of the nested tuple results in full replication of the Star-MVP i.e. the replication factor Rep is a function of the multiplicity of the MVP.

3.2 Delayed Complete Unnesting: Map-side Full Replication

NTGA data model and algebra are integrated into our extended Pig system, RAPID+ [9]. NTGA operators are nesting-aware and do not require unnesting before operations such as join. This allows the unnesting of Star-MVPs to be delayed to the map phase of the MVJoin operation as opposed to the reduce phase of a previous cycle. For example, the unnesting of Star-MVP can be delayed till MR_{J1} using NTGA (Figure 3 (b)), as opposed to unnesting in MR_{J2} using the Pig approach (Figure 3 (a)). RAPID+ uses an internal representation scheme, RDFMap (extended multi-map) that concisely represents tripegroups and supports efficient look-up of (Property, Object) pairs. In order to support MVPs, RDFMap is extended to support (Property, List<Objects>) pairs as shown in Figure 5 (top). For the rest of this discussion, we use the notation Pr1_rMap(pf1…pfn) to refer to a tripegroup corresponding to Product &Prod1 and containing an MVP prodFeature with n Object values {pf1…pfn}.

For graph pattern queries involving MVJoin, the unnest of Pr1_rMap is delayed till the map phase of the MVJoin. The map-side-unnest operation is integrated into the map phase of NTGA’s join, that completely unnests a Star-MVP and generates a map output tuple for each flattened copy of the Star-MVP (Rep is a function of the multiplicity of the MVP). Figure 5 (a) represents the 5 map output tuples containing Pr1_rMap, each tagged with one of the 5 object values of the MVP prodFeature. The delayed unnesting approach minimizes the redundancy factor for MR_{J2} (RedF_{J2}=0) resulting in savings in HDFS writes in the star-join phase of Star-MVP as well as reduced amount of HDFS reads in the MVJoin phase.

\[
\{\{&Prod1, pLabel, Product\}\}, \ldots, \{\ldots pProp, \ldots\}\}
\]

Figure 5: A comparison of the delayed map-side unnesting strategies: (a) Complete unnest (with Rep=5) and (b) Partial unnest (with Rep=2)

3.3 Delayed Partial Unnesting: Map-side Partial Replication

We observe that if the multiplicity of an MVP is greater than the number of partitions in the reducer space, it is likely that multiple copies of the Star-MVP are assigned to the same partition. Consider Figure 5 (a) with 2 reducers (r=2), where 3 copies of Pr1_rMap corresponding to the join keys pf1, pf3, pf5 respectively, are mapped to the same Reducer_bkt. The map-side sorting costs, local writes and network communication costs can...
be reduced if the references to \( Pr_l\_rMap \) can be shared across the reduce function space i.e. if \( Rep \) can be reduced.

We propose an alternative key-assignment strategy for MVJoins, such that multiple map keys can be assigned to the same \( rf\_bkt \) in the reduce function space. This partitioning scheme is based on indirect hashing of the map output key to the Reducer space. We define a function \( func^* \) that is used to map intermediate keys to the Reducer space. Given an intermediate key \( k1 \) we define:

\[
func^*(k1) = k1^*
\]

that maps the original key \( k1 \) to a group key \( k1^* \). The default partitioning scheme is then applied based on the group key \( k1^* \):

\[
\text{partition}(k1) = \text{hash}(k1^*) \% r
\]

In effect, an intermediate key-value pair \((k1, v1)\) is assigned to \( Reducer\_bkt(\text{hash}(k1^*) \% r) \) and two intermediate keys \( k1 \) and \( k2 \) are assigned to the same \( rf\_bkt \) if \( k1^* = k2^* \). In our example in Figure 5, \( p1f^0 = p5^0 = p5^1 \) and thus, the keys \{\( p1f, p5^0, p5^1 \)} map to the same bucket \( rf\_bkt_{p1} \), requiring just 1 copy of \( Pr_l\_rMap \) to be transferred to \( Reducer\_bkt \). We integrate the \( func^* \) into the map phase of NTGA’s join operator. Algorithm 1 shows map and reduce functions for the MVJoin operator (OptMVJoin).

Algorithm 1: OptMVJoin

```
OptMap (key: null, val: RDFMap rMap)
1. pList ← map-side-partial-unnest(rMap, func*);
2. foreach partialMap ∈ pList do
3. \( MVList \) ← extract MVP ObjList from partialMap;
4. \( \text{OptReduce}(key: rf\_bkt_{key}, val: List of RDFMaps R) \)
5. leftList ← extract Star-MVPs;
6. rightHash ← extract non-MVP Stars;
7. foreach leftR in leftList do
8. \( MVList \) ← extract MVP ObjList from leftR;
9. foreach joinKey in MVList do
10. \( \text{rightR} \leftarrow \text{rightHash}.get(joinKey); \)
11. Emit <JoinRDFMaps(leftR, rightR)>;
```

The map-side-partial-unnest operation partially unnests the RDFMap based on \( func^* \) (line 1). The two unnested RDFMaps in our example are \( Pr_l\_rMap(p1f, p5^0) \) and \( Pr_l\_rMap(p2, p4) \). A map output tuple is generated for each partially unnested RDFMap (line 2-4). The replication factor \( Rep \) is now a function of \( func^* \). However, since multiple join keys are now mapped to the same reduce function, the reduce function selectively joins the records in \( rf\_bkt_{k1} \), based on the original key \( k1 \). In our prototype implementation, the non-MVP relation (ProductFeature) is hashed based on the join key (line 5). Our algorithm iterates through each RDFMap in the Star-MVP relation (Product), and probes the hashed relation for each Object value (join key) of the MVP (lines 7-10). When a match is found, the two RDFMaps are joined as per the definition of \( TG\_Join \).

**Note on \( func^* \):** Our prototype implementation uses a hash function \( func^* \) parameterized by a partition factor \( N \), where \( N \) depends on several factors such as size of input data, multiplicity of MVP, number of reducer nodes, and average number of tuples that can be processed by a reducer. The impact of some of these factors is discussed in section 5.1.

4. Related Work

The vertical partitioning (VP) [1] storage model has been recommended for efficient storage of RDF data with MVPs to eliminate the redundant storage of the non-MVP components. An alternative storage scheme [11] includes MVPs in clustered n-ary relations based on the multiplicity of the MVP. Both these approaches may lead to redundancy while answering graph pattern queries with multiple star patterns and joins between them. The nested relational model [2] and object-oriented databases support sets, lists, maps and objects to capture multi-valued attributes and complex objects. Such extended models eliminate frequently occurring joins between objects by allowing nesting of related objects. Set-valued joins [6][15] and other set-level comparisons [12] have been studied in the context of main-memory and centralized systems.

Amongst the RDF processing systems on MapReduce, the heuristic cost-based plan generator in [8] greedily groups the non-conflicting joins in a query to minimize the required number of MR cycles. HadoopDB [3] uses a hybrid database-Hadoop architecture to split execution between the database systems and Hadoop. However, the reduction in the number of MR cycles is limited to the portion of the query evaluation that can be pushed into the database, and is currently dependent on a tunable parameter [7] that determines the partitioning scheme. In [23], the optimized grouping of join operations that can be executed in a single MR cycle is computed using an adaptive replicated join scheme that extends the work in [4]. Other techniques have been proposed to minimize the map output as well as the associated shuffle costs. 

**Case Study**

The goal of our evaluation was to compare the performance of the tuple-based and NTGA-based processing of graph pattern queries with MVPs, along with the impact of the proposed unnesting strategies. We chose Apache Pig that supports tuple-based algebra (Pig-Def and Pig-Opt), and extended it to include the NTGA operators (NTGA-Naïve) and the optimized MVP-join algorithm (NTGA-Opt). Pig-Def and Pig-Opt use the VP approach described in section 2. Table 1 summarizes the nesting and unnesting strategies used in the four approaches. Pig-Opt introduced additional project operators to eliminate the redundant join columns. We mainly consider query patterns that involve MVP as (i) non-join MVP – the join variable is not multi-valued; and (ii) MVJoin - object of MVP is a join variable.

**Table 1: Nesting and Unnesting Strategies**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Star-join of Star-MVP (Multiplicity M)</th>
<th>MVJoin phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pig-Def</td>
<td>M tuples (no nesting)</td>
<td>-</td>
</tr>
<tr>
<td>Pig-Opt</td>
<td>MVJoin: M tuples</td>
<td>Otherwise: 1 nested tuple</td>
</tr>
<tr>
<td>NTGA</td>
<td>1 nested RDFMap</td>
<td>Map-side full unnest</td>
</tr>
<tr>
<td>NTGA-Opt</td>
<td>1 nested RDFMap</td>
<td>Map-side partial unnest</td>
</tr>
</tbody>
</table>

**Setup:** The experiments were conducted on VCL. Error! Reference source not found., an on-demand virtual computing environment provided by NCSU. Each node in the cluster was a dual core Intel X86 machine with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. The experiments were...
Conducted on 10-node Hadoop clusters with block size set to 256MB and heap-size for child jvms set to 1024MB. Pig release 0.8.0 and Hadoop 0.20.1 were used. Our system [9] uses Jena’s ARQ as the SPARQL parser. All results recorded were averaged over two or more trials.

Testbed - Dataset and Queries: Synthetic datasets (n-triple format) generated using the BSBM\(^1\) generator tool were used. Three data sizes were generated using number of Products as the scalability factor – BSBM-250k, BSBM-500k, BSBM-1000k, with the size of data ranging from 22GB (BSBM-250k with 250,000 Products and approx. 87M triples in total) to a data size of 87GB (BSBM-1000k with 1000,000 Products and approx. 350M triples in total). BSBM dataset includes two MVPs – productFeature and type defined for a class of Products, with approx. multiplicity of 19 and 6 respectively. The evaluation tested varied graph pattern structures with multiple star patterns (S1, S2, S3) as represented in Table 2. Additional experiments were conducted on a real-world dataset – the infobox dataset (DBInfobox) from DBPedia [26] with 26.75M triples (17.5M properties and 9.25M types) of size 3.5GB. More than 45% of properties in this dataset are multi-valued, with both low and high multiplicity. Further, the multiplicity of some MVPs varied highly across instances. For example, the multiplicity of MVP influenced varied from 1 to as high as 50 for some instances. Three queries with varying number and multiplicity of MVPs were chosen – Dbp1, Dbp2 and Dbp3 contain graph patterns with 4, 5, and 6 MVPs respectively. Additional details about the evaluated queries are available on the project website\(^2\).

<table>
<thead>
<tr>
<th>Query</th>
<th>Joins</th>
<th>#Edges in Star-patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-1Star</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>High-1Star</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>Low-2Star</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>High-2Star</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>MV-2p</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>MV-3p</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>MV-4p</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>MV-5p</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>MVJoin-First</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>MVJoin-Last</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>Dbp1</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>Dbp2</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
<tr>
<td>Dbp3</td>
<td>S1(Mvp):3</td>
<td>S2:5</td>
</tr>
</tbody>
</table>

5.1 Evaluation Results

Low to high multiplicity of MVPs: Figure 6 shows the performance evaluation of Pig-Def, Pig-Opt and NTGA for two non-join MVPs with low (Product Type) and high (Product Feature) multiplicity of 6 and 19 respectively. The star-join result of a Star-MVP with multiplicity \(M\) is represented as \(M\) tuples in Pig-Def, with repetition of non-MVP fields leading to data redundancy. This redundancy impacts the HDFS write of the star-join phase as well as the HDFS read in the subsequent MR cycles with a join involving the Star-MVP. Both Pig-Opt and NTGA, represent Star-MVP using a single record (nested tuple and RDFMap respectively). In general, the nested models show more benefit with increasing multiplicity of the MVP – NTGA has a performance gain of 69% and 71% over Pig-Def for Low-1Star and High-1Star respectively.

Varying number of Star-subpatterns: Figure 6 shows that Pig-Opt and NTGA have a performance gain of 23% and 70% over Pig-Def respectively for the one star subpattern (Low-1Star and High-1Star). The primary reason for difference in performance of PigOpt and NTGA (though both use nested models) comes from the fact that Pig-Opt’s SPLIT approach initiates 4 times the number of mappers (344) as NTGA (86) to vertically partition the input based on the 4 SPLIT predicates in the star pattern. The high number of mappers increases the map-side local writes as well as the shuffle time in the case of Pig-Opt. The case of two star sub patterns (Low-2Star and High-2Star) demonstrates the compounding I/O, sorting and data transfer costs due to the redundancy caused by the Star-MVP in Pig-Def. Pig-Opt and NTGA show a performance gain of 30 / 49% and 73 / 79% for Low-2Star / High-2Star with two star patterns.

Varying Cluster Size: The multiplicity of the MVP also impacts the number of nodes required to process the graph pattern queries involving Star-MVP. Figure 7 shows that for query pattern with two sub-patterns on BSBM-250K, Pig-Def successfully completed on a 5-node cluster for low multiplicity Star-MVP, while it failed for the high multiplicity query (High-2Star). However, both the nested model approaches (PigOpt and NTGA) completed on the 5-node cluster. This is because the space required for the successful execution of a query in MapReduce is equal to the sum of the sizes of the intermediate HDFS writes (between MR cycles) and the final output (!MR cycle).

Varying Data Size: Figure 8 (c) shows a comparative evaluation of the three approaches with increasing size of RDF graphs. For the small data set (22GB) the performance of all three approaches were comparable. In NTGA and NTGA-Opt, the fewer number of intermediate results after the star-join computation phase initiate fewer numbers of Mappers for the MVP-join phase. Pig-def approach has 70% more mappers for the MVP-join phase and catch up quickly with the other two approaches. However, with increasing data size, we see an overall performance gain of 23% and 30% with large data sizes of 43GB and 86GB respectively.

---

1 http://www4.wiwiss.fu-berlin.de/bizer/BerlinnSPARQLBenchmark/spec
2 http://research.csc.ncsu.edu/coul/RAPID/SWIM2012
Results on join MVP (MVJoin): Figure 8 (a) shows a detailed performance comparison of the map and reduce execution times for Pig-Def, NTGA and NTGA-Opt. The query tested was MV-5p consisting of a MVJoin among two star patterns. In Pig-Def, the result of the star-join is stored as multiple n-tuples, and the join on MVP’s object value is processed using the default relational-style join algorithm (Pig-Def MR4 in Figure 8 (a)). NTGA-naive requires a more complex map phase to generate n copies of the RDFMap containing the MVP with n object values. This explains why NTGA’s map phase in MR4 takes almost the same time as the corresponding map phase in Pig-Def. NTGA-Opt’s map phase for MVP-Join (MR4) integrates indirect hashing (N=8k) that results in fewer map output when compared to the other two approaches. The references of map output are shared across reducer tasks assigned to the same reducer partition, resulting in reduced shuffle bytes, reduced costs of sort and data transfer.

Varying density of star containing MVP: We studied the impact of the cardinality of the star containing the join MVP by evaluating 4 different query patterns (MV-2p to MV-5p). Recall that denser stars correspond to larger size of non-MVP components, which are redundant in the map output of both Pig-Def and NTGA. We observed that NTGA-Opt showed fastest reduce phases among the three approaches for all the 4 queries on BSBM-500k (10-node cluster) with N=4k, with overall performance gain increasing from 27% for MV-2p to 47% got MV-5p as shown in Figure 8 (b).

Varying partition factor (N): Table 3 shows the execution time for the MVJoin phase (MR4 in Pig-Def and MR5 in NTGA) for query MV-5p, along with the impact of the varying reduce groups. In both Pig-Def and NTGA, the number of reduce groups is equal to the number of distinct join keys (47884).

<table>
<thead>
<tr>
<th>Approach (Reduce Groups)</th>
<th>Map Output Records</th>
<th>MVP Rep (%)</th>
<th>Map (s)</th>
<th>Reduce (s) (avg. shuffle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pig-Def (47884)</td>
<td>4.905M</td>
<td>19.43</td>
<td>179</td>
<td>246 (121)</td>
</tr>
<tr>
<td>NTGA (47884)</td>
<td>4.905M</td>
<td>19.43</td>
<td>329</td>
<td>288 (179)</td>
</tr>
<tr>
<td>NTGAOpt (1000)</td>
<td>4.873M</td>
<td>19.30</td>
<td>273</td>
<td>266 (159)</td>
</tr>
<tr>
<td>NTGAOpt (100)</td>
<td>4.583M</td>
<td>18.14</td>
<td>249</td>
<td>257 (144)</td>
</tr>
<tr>
<td>NTGAOpt (80)</td>
<td>4.483M</td>
<td>17.74</td>
<td>237</td>
<td>255 (139)</td>
</tr>
<tr>
<td>NTGAOpt (60)</td>
<td>4.246M</td>
<td>16.80</td>
<td>223</td>
<td>245 (130)</td>
</tr>
<tr>
<td>NTGAOpt (40)</td>
<td>4.064M</td>
<td>16.06</td>
<td>217</td>
<td>244 (126)</td>
</tr>
<tr>
<td>NTGAOpt (20)</td>
<td>3.326M</td>
<td>13.11</td>
<td>185</td>
<td>212 (95)</td>
</tr>
</tbody>
</table>

However, we note that Pig-Def starts with higher number of map input records (due to flattened MVP in previous cycles) resulting in initiation of 28 mappers as compared to just 3 mappers in the case of NTGA. The issue of better estimation of mappers for MVJoins is a concern in the case of nested models and is pursued as future work. In this section, we focus on the impact of N for fixed mappers (NTGA and NTGA-Opt). With NTGA, the replication factor (MVP Rep in Table 3) in the map phase is equal to the multiplicity of the MVP (19.43).

A lower value of N increases the sharing of data references (13% replication for N=20), resulting in lower shuffle costs, but increases the average time taken by a reduce task since large number of records are processed by each reduce function. On the other hand, a higher value of N provides less sharing opportunities. The cost model for estimation of N also needs to capture the size of data that can be processed by a reducer based on its heap size. Further the impact of N on replication factor depends on the distribution of MVP (assignment of set of prodFeatures to a Product). Additional experiments on test queries MVJoin-First and MVJoin-Last demonstrated the impact of the join order in the selection of the partition factor.

Varying multiplicity of an MVP across instances: Figure 9 shows a comparison of execution times for the different unnesting strategies for the three test queries on DBInfobox. Though DBp2 had more number of MVPs than DBp1 (5 vs. 4), the MVPs in DBp1 had a higher average multiplicity than those in DBp1. At the end of the star-join computation phase in DBp1, the partial unnesting in Pig-Opt and the delayed unnesting in NTGA/NTGA-Opt resulted in 8% and 43% less intermediate tuples when compared to the Pig-Def approach. For DBp3, the partial and delayed unnesting strategies result in 55% and 67% less intermediate tuples than in Pig-Def. The limited benefit of delayed unnesting over Pig-Opt in the second case is due to the low average multiplicity of the join-MVP in DBp3.
Further, the varying degree of multiplicity across instances also impacts the selection of the partition factor $N$ in NTGA-Opt. For MVJoins in DBp2 and DBp2 with low average multiplicity of join-MVP, the execution times for NTGA and NTGA-Opt are comparable.

6. Conclusion and Future Work
This work in an on-going research on the efficient management of intermediate results while processing RDF graph patterns containing multi-valued properties. The proposed delayed and partial unnesting strategies were integrated with the Nested TripleGroup Algebra for MapReduce based processing of graph pattern queries. Future directions include developing a theoretical cost model to estimate the appropriate hash space required for the indirect hashing scheme proposed in this paper. We also require better estimation of the appropriate number of mappers that should be initiated to process MVJoins on nested representations of Star-MVPs.

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

8. REFERENCES