# Scaling Unbound-Property Queries on Big RDF Data Warehouses using MapReduce*

Padmashree Ravindra
Department of Computer Science
North Carolina State University
pravind2@ncsu.edu

Kemafor Anyanwu
Department of Computer Science
North Carolina State University
kogan@ncsu.edu

## ABSTRACT

Semantic Web technologies are increasingly at the heart of many integrated scientific and general purpose data warehouses. Flexible querying of such diverse data collections with (partially) unknown structures can be enabled using triple patterns with 'unbound' properties (edges with don't care labels). When evaluating such queries using relational joins, intermediate results contain redundancy due to repeated combination of bound-property mappings with those of the unbound properties. However, in distributed-processing contexts, the footprint of intermediate results directly impacts I/O and communication costs. Given the popularity of MapReduce-based platforms for periodic on-demand scaling using Cloud resources, we propose an algebraic optimization technique that interprets unbound-property queries on MapReduce, using a non-relational algebra based on a TripleGroup data model. The approach enables shorter execution workflows and reduced costs for processing RDF queries on MapReduce. This paper introduces new logical and physical operators, and query rewriting rules for interpreting unbound-property queries using the TripleGroup-based data model and algebra. A key optimization strategy is to concisely represent intermediate results as far along an execution workflow as possible, thus minimizing the effects of redundancy. The proposed work is integrated into Apache Pig. Experiments conducted on real-world and synthetic benchmark datasets demonstrate their benefit over popular relational-style MapReduce systems.

## 1. INTRODUCTION

The successful adoption of Semantic Web technologies to interlink diverse (related) datasets has led to large semantically-integrated scientific (Uniprot [8], Bio2RDF [9]) and general purpose (DBpedia [7], Billion Triple Challenge [1]) RDF data warehouses. The heterogeneous and evolving nature of such data collections makes it difficult for users to be familiar with different kinds of relationships that exist in the data. Consequently, exploration of datasets in data-integration [23] and data archival [36] scenarios require flexibility in querying, i.e., the ability to use structural variables or "don't cares" in queries. SPARQL [28], the standard query language to specify graph pattern queries on the Semantic Web, enables flexible querying of datasets by allowing `OPTIONAL` substructures or substructures with missing edge labels. The latter are called *unbound-property* triple patterns and can be used to query unknown relationships ("*Scientists in some way associated to the same city*"), relationships with partial knowledge ("*Gene Ontology terms related to a gene Rxr*"), or to retrieve all available information about a resource ("*What is known about the Hexokinase gene?*").

Consider an example SPARQL query $Q1$ on Bio2RDF, a Life Sciences RDF dataset. $Q1$ is useful to analyse the Parkinson's disease and involves two unbound-property triple patterns (1) and (5).

| **Query Q1** | **Description** |
|---|---|
| `SELECT` ?s1, ?label1, ?s2, ?label2, ?o2 | *Retrieve gene ontology (GO) terms related to "rxr", a gene of interest in analyzing Parkinson's disease.* |
| `WHERE {` | *Q1 contains two star subpatterns,* |
| **?s1** ?p1 ?o1 . (1) | $SJ_1$ *(1-2) and* $SJ_2$ *(3-5).* |
| `FILTER` regex(?o1, "rxr") | *(1) matches triples whose object* |
| **?s1** label ?label1 . (2) | *contains string "rxr" (any property).* |
| **?s2** xGO ?o2 . (3) | *(5) specifies an unknown* |
| **?s2** label ?label2 . (4) | *relationship connecting the two* |
| **?s2** ?p2 ?s1 . (5) | *star subpatterns in the query.* |
| } | |

Other than querying scenarios in integrated data warehouses, subqueries with unbound-property triple patterns are also generated while optimizing ontological queries by rewriting them as a union of conjunctive queries. Examples of unbound-property queries can be found in real [23] and synthetic Semantic Web benchmarks [11], as well as other studies [22, 36]. In fact, 84% of queries in [2] involve unbound-property triple patterns.

Given a triple relation $T$ and subset relations $T_{xGO}$ and $T_{label}$ with property types $xGO$ and $label$, respectively, the subquery $SJ_2$ can be evaluated using relational joins ($T_{xGO} \bowtie T_{label} \bowtie T$). Figure 1 (right) shows the subrelations of $T$ participating in $SJ_2$ and a snapshot of the star-join result. An issue with intermediate results in such cases is redundancy. For example, the result for $SJ_2$ in Figure 1(top right) contains repeated occurrences for matches of the bound properties – $xGO$ and $label$, with each match of the unbound-property triple pattern. The numbers of matches for the unbound-property triple pattern could be large if properties in the input dataset have high multiplicity ($gene9$ is associated with multiple $xRef$), further aggravating the issue of redundancy. High-multiplicity properties are common in real-world social networks as well as biological datasets such as Uniprot and Bio2RDF, e.g., some Uniprot properties have multiplicity as high as 13K.

For applications with periodic scale–up requirements, the growing trend is to employ cloud-processing platforms, e.g., Hadoop [10], Dryad [16], Hive [37], Pig [26], that are based on the MapReduce [12] computing model. However, any redundancy in interme-

*The work presented in this paper is partially funded by NSF grant IIS-1218277.





10.5441/002/edbt.2015.16

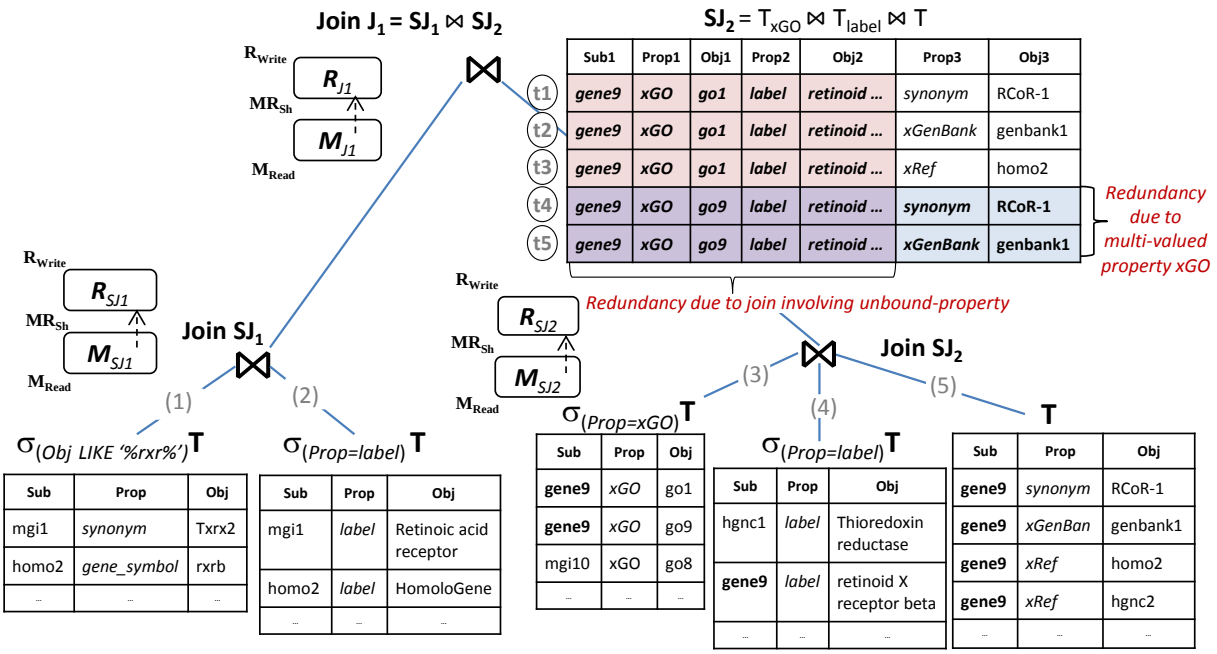



**Figure 1: A MapReduce workflow for an unbound-property graph pattern query $Q1$ with two star subqueries $SJ_1$ and $SJ_2$; Join result of unbound-property star subpattern $SJ_2$ contains redundant information related to bound properties (xGO, label)**

diate results impacts query processing costs, particularly for MapReduce based distributed processing platforms that involve shipping of intermediate results across the network. The intermediate result footprint also impacts additional costs associated with sorting phases, materialization between the 2-steps of a MapReduce (MR) execution cycle, and total disk space requirements to store all intermediate states for fault-tolerance purposes. Hence, it is critical to minimize the footprint of intermediate results.

## 1.1 Related Work

*Optimizing Relational Query Plans on MapReduce:* There have been several efforts to shorten the length of MR workflows [6, 40, 15, 5, 27] to minimize the overall costs of MapReduce-based processing, sharing scans [24, 25, 39] and computations [24, 13] across MR workflows, cost-based and transformation-based MR workflow optimizer [20], and data skew problems [19]. Multi-way join algorithms [6, 40] cluster multiple joins into a single [6] or few [40] MR cycles, but have not been applied to join-intensive workloads. Amongst the MapReduce-based RDF processing systems, SHARD [32] uses initial MR cycles to cluster triples into star subgraphs, followed by separate MR cycles to process each clause in the SPARQL query. HadoopRDF [15] pre-processes triples using the vertical-partitioning (VP) [4] approach, and uses heuristics to greedily group non-conflicting joins in a query to minimize the required number of MR cycles. However, unbound-property queries would require processing a union of all VP property relations. The HadoopDB-based extension [14] uses a hybrid database-Hadoop architecture that exploits the partitioning scheme to push part of the execution into the database/RDF-3X. Hash partitioning on Subject can enable local evaluation of unbound-property star subpatterns. However, once the execution is handed over to Hadoop the redundancy in intermediate results impacts the disk I/O, sorting, and communication costs for the rest of the execution workflow. In order to minimize the data shuffle costs, MRShare [24] enables sharing of map output data across grouping operations on a common input relation. Some other works proposed a value-partitioning scheme [21] to manage reducer-unfriendly groups during the cube computation process, and a reducer-routing strategy [38] that groups intermediate keys to balance the data across reducers. The evaluation strategies proposed in this paper, i.e., lazy $\beta$-unnesting strategies, are in similar spirit.

*Optimizing unbound-property queries:* Earlier studies [35, 34] have shown that the vertical-partitioning (VP) [4] storage model may be inefficient for unbound-property queries. Such queries result in multiple joins and large unions of VP relations, which gets worse for data containing large number of property types. The multi-indexing schemes in systems such as RDF-3x [22] could benefit single-star unbound-property queries. However, such systems may not scale well for large RDF graphs, particularly for queries with low selectivity and unbound objects [15]. There have been efforts [36] to optimize simple unbound-property queries to RDF views over relational databases. Since naive translation of an unbound-property query into SQL results in unions of multiple subqueries, the proposed Group Common Term transformer [36] exploits common terms in complex disjunctive SQL queries and rewrites them into a smaller number of queries. Our work proposes a scalable solution for processing unbound-property queries on MapReduce-based parallel processing platforms.

**Prior Work.** A previous work explored the use of a non-relational data model and algebra, i.e., the *Nested TripleGroup Data Model and Algebra* (NTGA) [30, 17], for efficient RDF query processing on MapReduce. The NTGA allows an alternative interpretation of queries in terms of a "grouping" operation and a set of *triplegroups*, that enables shorter execution workflows when compared to relational query plans in systems such as Hive and Pig. For example, query $Q1$ requires 3 MR cycles altogether (two cycles for computing star-joins $SJ_1$, $SJ_2$, and a third cycle to join the stars) as shown in Figure 1, while the NTGA would compute both $SJ_1$ and $SJ_2$ in a single cycle using a "grouping" operation, followed by a second cycle to compute the join between the stars.

***Comparison with Redundancy due to Multi-valued Properties.*** Unlike the normalized representation of intermediate results of re-

lational operations, the nested triplegroup data model can concisely represent intermediate results with multi-valued properties, e.g.,

{ ($gene9$, *xGO*, **{go1, go9}**) // A single triplegroup representing
($gene9$, *label*, retinoid...) // two n-tuples $t1$ and $t4$
($gene9$, *synonym*, RCoR-1)} // by nesting object component

Though the "nested object" model and *nesting-aware physical operators* [31, 29] reduce the I/O footprint of execution workflows, a join involving an unbound-property triple pattern would still produce '$n$' triplegroups (assuming $n$ triples with subject $gene9$). More importantly, all $n$ triplegroups contain redundant bound-property component. In this paper, we generalize the concept of triplegroup nesting to allow *nesting of property-object components*, to implicitly represent intermediate results while evaluating unbound-property queries. However, such an implicit representation involves triples playing multiple roles, i.e., a triple may match the bound and the unbound component of a query, which needs to be incorporated into the "unnest" process, referred here after as $\beta$-`unnest`. Additionally, there are implications of when and what portion of a triplegroup is $\beta$-unnested during the different phases of an execution workflow, resulting in choices for evaluation strategies. Specifically, this paper makes the following contributions:

- We introduce new logical operators and query rewrite rules that allow the translation of unbound-property queries into NTGA-based logical plans. The correctness and sufficiency of query rewrite rules is also presented.
- We introduce new physical operators that offer different evaluation strategies - *eager vs. lazy $\beta$-unnesting* of intermediate results during query processing.
- Extensive evaluation using large RDF graphs, both Semantic Web synthetic benchmark and real-world biological datasets, demonstrates the efficiency of our approach over relational-style processing of unbound-property queries in Pig and Hive.

# 2. PRELIMINARIES

## 2.1 MapReduce and Data Processing

In the MapReduce programming model, data processing tasks are encoded as *map* and *reduce* functions, that are executed in parallel across a cluster of computing nodes. Relational operations such as a join between two relations, maps to a processing cycle consisting of two phases – the *Map* phase and the *Reduce* phase. In the *Map* phase, a set of slave nodes (*mappers*) execute the *map* function that tags each tuple based on the join key. Map output tuples are partitioned on the join key and *shuffled* across the network to another set of slave nodes (*reducers*). In the *Reduce* phase, each reducer receives a collection of tuples with the same join key, and computes the join. The output of the Reduce phase is written onto the *Hadoop Distributed File System* (HDFS) and read back in a subsequent cycle. Each MapReduce (MR) cycle involves costs associated with initial input data reads in the map phase ($M_{Read}$), the data shuffling costs between mappers and reducers that involve local disk writes at the mappers ($M_{Write}$), sort-merge costs ($MR_{Sort}$) as well as network transfer costs ($MR_{TR}$), and finally the cost of writing the reduce output to the HDFS ($R_{Write}$).

To evaluate graph pattern queries on MapReduce, one can exploit the fact that graph pattern queries often consist of multiple star-structured subqueries e.g., $SJ_1$ and $SJ_2$ rooted at variables *?s1* and *?s2* in query $Q1$, that can be evaluated using a multi-way join algorithm. For a graph pattern query with $l$ star subpatterns, the typical MapReduce execution plan generated by relational-like platforms such as Hive and Pig consist of a sequence of MapReduce cycles $MR_1, MR_2, .., MR_n$ such that $1 \leq n \leq (l-1)$ cycles are used for executing the $l$ star-joins, and $(n-l)$ MapReduce cycles for the remaining joins in the query. Our example query $Q1$ can be evaluated in 3 MR cycles as shown in Figure 1: $MR_{SJ1}$ and $MR_{SJ2}$ to compute star subpatterns $SJ_1$ and $SJ_2$ respectively, followed by a third cycle $MR_{J1}$ to join the stars. Given such a MR workflow $W$, the overall processing cost of $W$ is:

$$\text{Cost}(W) = \text{cost}(MR_1) + \text{cost}(MR_2) + ... + \text{cost}(MR_n)$$

where the I/O, sorting, and network transfer costs of each cycle compound across multiple cycles of a lengthy workflow. Furthermore, the portion of redundant data in the intermediate results directly impacts the HDFS writes ($R_{Write}$) for the current MR cycle, and the scan costs ($M_{Read}$) and shuffle costs ($MR_{Sh}$) of subsequent MR cycles. Hence, the redundancy has a ripple effect on the costs of reads, writes, sorting and the data transfer costs across a workflow with multiple MR cycles. Thus, lengthy workflows lead to performance inefficiency and an important optimization goal is to *minimize the length of an MR execution workflow* [6, 15, 40].

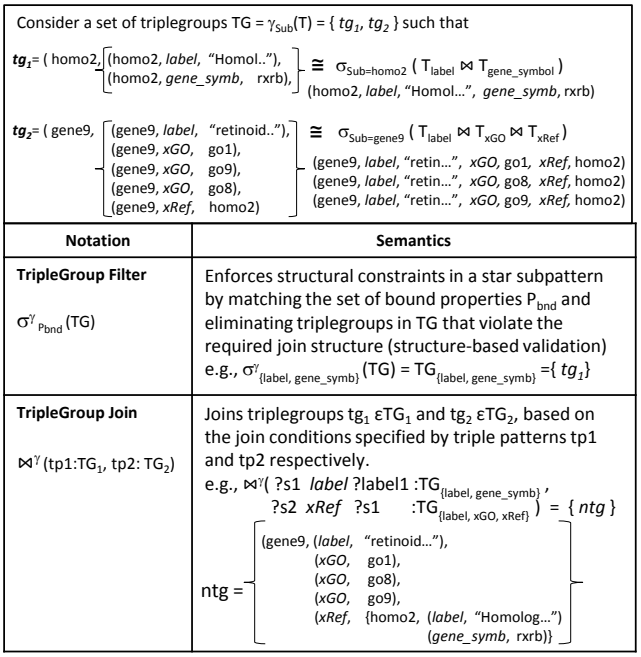
However, grouping of joins based on star structures does not necessarily result in the typical join order generated using traditional cost-based optimization. One challenge is that most cloud processing platforms are used in an on-demand model, where precomputed statistics for cost-based optimization may not be available or take too long to compute, resulting in long lead times. More importantly, ordering joins in terms of their costs may generate some linear subplans requiring one input as the full triple relation, which in the absence of an index is a full scan. Such plans may incur larger overhead due to HDFS reads, which outweighs the savings achieved by pushing selective joins ahead.

Our previous work [30, 17] explored an algebraic optimization technique that rewrites graph pattern queries using operators that are more MapReduce-cognizant. It has been demonstrated that the underlying data model and algebra called the *Nested TripleGroup Data Model and Algebra* (NTGA), not only results in short execution workflows [30, 17], but also enable scan-sharing [18] across star subpatterns, while reducing the I/O footprint of intermediate results [31, 29]. In the next section, we overview the data model and algebraic operators in NTGA that enable nimble execution workflows while evaluating RDF graph pattern queries on MapReduce.

## 2.2 TripleGroup-based Processing of Graph Pattern Queries on MapReduce

The NTGA data model represents the RDF database as sets of related "group of triples" or *TripleGroups*. For example, triples in the database can be modeled as a set of *Subject TripleGroups*, each consisting of triples that share a common subject. For example, triplegroups $tg_1$ and $tg_2$ in Figure 2 represent subject triplegroups corresponding to triples sharing common subjects gene9 and homo2, respectively. Given such a data model, answering graph pattern queries translates to manipulation of triplegroups. Some of the most relevant triplegroup operators are summarized in Figure 2 and discussed below.

**Algebraic Operators.** Consider a query $Q'$ with two star subpatterns $St_1$={$label$, $gene\_symb$} and $St_2$={$label$, $xGO$, $xRef$}. NTGA's grouping operator ($\gamma$) computes a set of subject triplegroups $TG$ based on the subject column as shown in Figure 2. Given such a set of triplegroups $TG$, a match to a star subpattern is a selection operation ($\sigma^{\gamma}$) that extracts a subset of triplegroups that match the required join structure, i.e., a valid triplegroup must contain at least one triple corresponding to each of the property types




| Notation | Semantics |
|---|---|
| **TripleGroup Filter**<br>$\sigma^{\gamma}_{P_{bnd}}(TG)$ | Enforces structural constraints in a star subpattern by matching the set of bound properties $P_{bnd}$ and eliminating triplegroups in TG that violate the required join structure (structure-based validation)<br>e.g., $\sigma^{\gamma}_{\{label,\ gene\_symb\}}(TG) = TG_{\{label,\ gene\_symb\}} = \{ tg_1 \}$ |
| **TripleGroup Join**<br>$\bowtie^{\gamma}(tp1{:}TG_1,\ tp2{:}\ TG_2)$ | Joins triplegroups $tg_1 \in TG_1$ and $tg_2 \in TG_2$, based on the join conditions specified by triple patterns tp1 and tp2 respectively.<br>e.g., $\bowtie^{\gamma}$( ?s1 *label* ?label1 :$TG_{\{label,\ gene\_symb\}}$ ,<br>?s2 *xRef* ?s1 :$TG_{\{label,\ xGO,\ xRef\}}$ ) = { *ntg* }<br>ntg = { (gene9, (*label*, "retinoid..."), (*xGO*, go1), (*xGO*, go8), (*xGO*, go9), (*xRef*, {homo2, (*label*, "Homolog...") (*gene_symb*, rxrb)} ) } |

**Figure 2: Example NTGA Operators**

in the star subpattern. Triplegroup $tg_1$ is a valid match for $St_1$ and is said to belong to the equivalence class $TG_{\{label,gene\_symb\}}$ that defines its join structure. Further, matching multiple star subpatterns translates to a disjunctive selection based on the set of properties in each star subpattern. For example, the two star subpatterns in $Q'$ can be computed as follows:

$$\sigma^{\gamma}_{(\{label,gene\_symb\}\vee\{label,xGO,xRef\})}(TG)$$

Joins between star subpatterns can be computed using the join operator ($\bowtie^{\gamma}$) that is semantically equivalent to the relational join operator but is defined on triplegroups. The object-subject join between triplegroups $tg_1$ and $tg_2$ results in a nested triplegroup $ntg$ whose root is the triplegroup $tg_1$ and child triplegroup is $tg_2$. Before proceeding, we review the notion of *content-equivalence* that enables lossless translation between relational algebra and NTGA plans.

**Relational Algebra ↔ NTGA Plans.** Triplegroups are '*content-equivalent*' (represented as $\cong$) to the set of n-tuples computed using a set of relational-style joins. Let $Stp$ be a star subpattern comprising of the set of bound properties $\{P_1, P_2, ..., P_k\}$, and $T_{Stp}$ be the join result of vertically partitioned subset relations $T_{P_1}, T_{P_2}, ..., T_{P_k}$. Let $T_{Stp(s)}$ represent the subset of $T_{Stp}$ with subject Sub = s.

$$T_{Stp(s)} = \sigma_{Sub=s}(T_{P_1} \bowtie T_{P_2} \bowtie ... \bowtie T_{P_k})$$

Each tuple in $T_{Stp(s)}$ is of 3k arity (each property in $Stp$ is associated with 3 columns). Let $\pi_{P_i}$ denote the projection of the (Sub, Prop, Obj) columns corresponding to the parent relation $T_{P_i}$ with bound-property $P_i$. Let $tg_s$ represent the set union of triples formed by the 3 columns, i.e.

$$tg_s = \pi_{P_1}(T_{Stp(s)}) \cup \pi_{P_2}(T_{Stp(s)}) \cup ... \cup \pi_{P_k}(T_{Stp(s)})$$

In summary, the tuples in $T_{Stp(s)}$ can be vertically partitioned into 'triples' whose union is equivalent to a subject triplegroup $tg_s$ in the NTGA data model. For our example data in Figure 2,

$$tg_1 \cong \sigma_{Sub=homo2}(T_{label} \bowtie T_{gene\_symb})$$
$$tg_2 \cong \sigma_{Sub=gene9}(T_{label} \bowtie T_{xGO} \bowtie T_{xRef})$$

**Benefits of NTGA Query Plans.** For a query with '$n$' star subpatterns, NTGA can compute ALL star subpatterns concurrently using a single 'grouping' operation, by first 'grouping' the triples

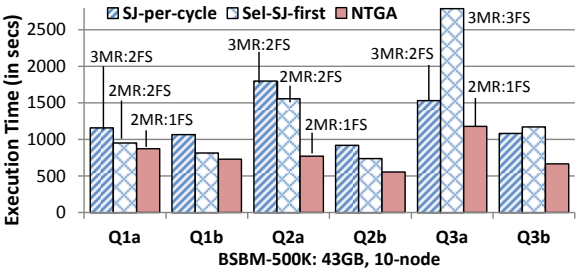



**Figure 3: Evaluation of different groupings of star-joins (MR: No. of MapReduce cycles, FS: No. of Full Scans)**

into subject triplegroups and then applying a disjunctive selection based on the multiple star subpatterns. This is in contrast to the relational-style approach where each star subpattern is evaluated as a relational-style join. The grouping-based star-join computation naturally fits the *map-group-reduce* theme in MapReduce, and translates to just one MR cycle for computing all star-joins in the query (as opposed to '$n$' MR cycles using relational-style plans). In addition to the reduction in the number of required MR cycles, NTGA also results in reduced size of intermediate results. Multiple related n-tuples resulting from relational-style joins involving a multi-valued property are implicitly represented as a single triplegroup in NTGA. For example, the 3 n-tuples corresponding to $Stp_2$ containing a multi-valued property *xGO* are implicitly represented using a single triplegroup $tg_2$ as shown in Figure 2. This is specifically important in minimizing the I/O footprint of long MapReduce execution workflows while processing RDF graph pattern queries.

Consider a case study using 6 test queries (each with two star subpatterns) using the BSBM synthetic benchmark dataset (43GB) on a 10-node Hadoop cluster, as shown in Figure 3. The test queries have varying join structures with Object-Subject join ($Q$1a, $Q$1b, $Q$2a, $Q$2b) and Object-Object join ($Q$3a, $Q$3b) between star patterns. Queries $Q$1b, $Q$2b, $Q$3b are variations of $Q$1a, $Q$2a, $Q$3a respectively, where one of the two star-joins is highly selective due to an additional filter on the object column. Additional details about the evaluated queries are available on the project website [3]. We evaluated three different groupings of star subpatterns in a query, (i) a star-join per cycle approach (*SJ-per-cycle*), (ii) most selective grouping of joins first but preserving star structure as much as possible to minimize MR cycles (*Sel-SJ-first*), and (iii) concurrent evaluation of star-joins using the grouping-based approach in NTGA. *SJ-per-cycle* approach requires 3 MR cycles for all queries (2 of 3 cycles require full scan of triple relation). For Object-Subject joins, *Sel-SJ-first* approach can group joins into just 2 MR cycles (both cycles scan entire triple relation). For the Object-Object join ($Q$3a, $Q$3b), *Sel-SJ-first* still requires 3 MR cycles, but more importantly has very high HDFS reads due to full scan of triple relation in all 3 cycles. In contrast, the *NTGA* approach is able to minimize the number of MR cycles (2 cycles for all queries), as well as minimize the required number of full scans of the triple relation, thus outperforming the other two approaches for all test queries.

Earlier work on NTGA captures basic graph patterns. In this work, we build on the advantages of the TripleGroup data model and algebra for efficient evaluation of unbound-property graph pattern queries on MapReduce. Specifically, the semantics of the group-filter operator ($\sigma^{\gamma}$) requires all properties in the query structure to be bound. However, to capture more complex patterns, the algebra and the set of rewrite rules need to be extended. The following section introduces a number of extensions which allow us to relax the above constraint to provide an extended group-filter semantics for

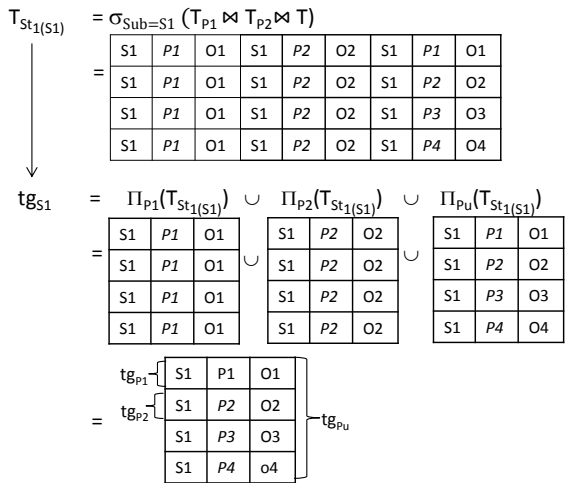



**Figure 4: Transformation: n-tuples to a triplegroup**

evaluating unbound-property queries.

# 3. REWRITING UNBOUND-PROPERTY QUERIES USING NTGA

Consider an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ such that $P_{bnd} = \{P_1, P_2, ..., P_k\}$ represents the set of bound properties and $P_{unbnd}$ represents an unbound property. Let $T_{St_u}$ be the star-join result of relation $T(Sub, Prop, Obj)$ with vertically partitioned subset relations $T_{P_1}$, $T_{P_2}$,...,$T_{P_k}$, and let $T_{St_u(s)}$ represent a subset of $T_{St_u}$ with subject $Sub = s$.

$$T_{St_u(s)} = \sigma_{Sub=s}(T_{P_1} \bowtie T_{P_2} \bowtie ... \bowtie T_{P_k} \bowtie T)$$

The tuples in $T_{St_u(s)}$ have arity 3($k$+1), where each property in $St_u$ is associated with 3 columns in $T_{St_u(s)}$. Figure 4 represents the tuples in $T_{St_1(S1)}$ for a star-pattern $St_1$ with bound properties $P_1, P_2$ and an unbound property. To determine how $St_u$ will be evaluated using NTGA, it will be useful to develop some correspondence between $T_{St_u(s)}$ and a subject triplegroup in NTGA. Note that a single triple may play multiple roles (occur multiple times) in the result of an unbound-property star pattern – one as a match for the bound property and the other as a match for the unbound property. For example, $(S1, P1, O1)$ in Figure 4 occurs once for the join with $T_{P_1}$ and once for the join with $T$. In the NTGA data model, such multiple occurrences are implicitly represented once, which must be accounted for in the transformation process.

Continuing with the transformation process, let $\pi_{P_i}$ and $\pi_{P_u}$ denote the projection of the (Sub, Prop, Obj) columns corresponding to parent relation $T_{P_i}$ with bound property $P_i$, and unbound property $P_{unbnd}$ respectively. Let $tg_s$ represent the set union of triples formed by the 3 columns, i.e.

$$tg_s = \pi_{P_1}(T_{St_u(s)}) \cup ... \cup \pi_{P_k}(T_{St_u(s)}) \cup \pi_{P_u}(T_{St_u(s)})$$

Figure 4 denotes the triples in $tg_{S1}$ for our example star-pattern $St_1$, formed by the set union of the partitions $\pi_{P_1}$, $\pi_{P_2}$, and $\pi_{P_u}$. $tg_s$ has the following properties:

(i) $\forall t_i, t_j \in tg_s$, the triples $t_i, t_j$ agree on the subject column $s$.

(ii) $\exists$ non-empty subset of triples $tg_{P_i} \subseteq tg_s$ such that $tg_{P_i} = \pi_{P_i}(T_{St_u(s)})$, for each bound property $P_i \in P_{bnd}$.

(iii) $\exists$ a non-empty subset of triples $tg_{P_u} \subseteq tg_s$ such that $tg_{P_u} = \pi_{P_u}(T_{St_u(s)})$ and $tg_{P_u} \cap (tg_{P_1} \cup ... \cup tg_{P_k})$ may be non-empty.

Essentially, the tuples in $T_{St_u}$ can be horizontally partitioned into sets of tuples with the same Subject column, and each element in the partition can be vertically partitioned into 'triples' whose union is equivalent to a subject triplegroup $tg_s$ in the NTGA data model. The use of set union instead of bag union ensures that we have a triplegroup. Further, subsets of triples in $tg_s$ represent matches to the bound and unbound-property triple patterns in $St_u$. This process basically describes a sequence of translation steps from the relational algebra to NTGA. In other words,

$$\begin{aligned} T_{St_u(s)} &= \sigma_{Sub=s}(T_{P_1}) \bowtie ... \bowtie \sigma_{Sub=s}(T_{P_k}) \bowtie \sigma_{Sub=s}(T_{P_u}) \\ &= tg_{P_1} \bowtie ... \bowtie tg_{P_k} \bowtie tg_{P_u} \end{aligned}$$

Conversely, for our example star-pattern $St_1$ in Figure 4, tuples in $T_{St_1(S1)}$ are implicitly represented in $tg_{S1}$ and can be produced by $(tg_{P_1} \bowtie tg_{P_2} \bowtie tg_{P_u})$. A useful property is to distribute the join with the unbound-property triple pattern across a union of subset relations of $T$. In other words, if the triple relation $T$ can be partitioned into two subset relations, i.e., $T = \{T_{P'_u} \cup T_{P''_u}\}$. Then by the distributivity of join over union, we have:

$$T_{P_{bnd}} \bowtie (T_{P'_u} \cup T_{P''_u}) \equiv (T_{P_{bnd}} \bowtie T_{P'_u}) \cup (T_{P_{bnd}} \bowtie T_{P''_u})$$

Evaluating $St_u$ using NTGA requires applying group filter ($\sigma^{\gamma}$) to match the required query structures. Recall that $\sigma^{\gamma}$ is defined in terms of a set of bound properties. One might consider evaluating an unbound-property star-pattern query using $\sigma^{\gamma}$ with a disjunction of concrete pattern combinations. Each such combination will consist of the set of bound properties $P_{bnd}$ with each property in the database. For example, if $P_{bnd} = \{P_1, P_2\}$ is the set of bound-properties in the star pattern and $P = \{P_1, P_2, ..., P_{10}\}$ represents the set of all properties in the database. Then, the $\sigma^{\gamma}$ expression is:

$$\sigma^{\gamma}_{(\{P_1,P_2,P_1\} \vee \{P_1,P_2,P_2\} \vee ... \vee \{P_1,P_2,P_{10}\})}(TG)$$

This would filter out triplegroups that do not match any of the required pattern combinations. However, the approach of enumerating all possible pattern combinations may be inefficient depending on the number of properties in the database. Additionally, the subject triplegroup $tg_s$ may contain additional triples relevant to other patterns, and hence may not exactly match a single pattern combination. Hence, there is a need to relax the $\sigma^{\gamma}$ to restrict the matching of structural constraints to the bound properties of the unbound-property star pattern. This means that triplegroups that contain all the bound properties (may contain additional properties), should be produced as part of the result for $\sigma^{\gamma}$. Once this is done, we need to extract subsets of triples in $tg_s$ that are exact matches for any of the required pattern combinations. This is achieved by extracting the subset of triples corresponding to $P_{bnd}$ and generating their union with each triple in the unbound-property subset $tg_{P_u}$. In the following section, we provide the formal definitions for a specialized group-filter operator ($\sigma^{\beta\gamma}$) and the unnest operator ($\beta$-`unnest`) that extracts the perfect matches to the unbound-property star-pattern. From here on, we assume the convenience function $tg.props()$ ($st.props()$) to retrieve the set of properties in a triplegroup $tg$ (star pattern $st$).

DEFINITION 1. *($\beta$ **Group-filter**) Given a set of subject triplegroups $TG$ and a star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ containing an unbound property, the $\beta$ **group-filter** operator $\sigma^{\beta\gamma}$ returns the subset of triplegroups in $TG$ that contain a non-empty subset of triples matching all bound properties $P_{bnd}$. Specifically,*

$$\sigma^{\beta\gamma}_{(P_{bnd},P_{unbnd})}(TG) := \{\, tg_i \in TG \mid P_{bnd} \subseteq tg_i.props() \}$$

Essentially, $\sigma^{\beta\gamma}$ ensures that triplegroups contain a matching triple for each of the bound properties in $P_{bnd}$. Additionally, triplegroups

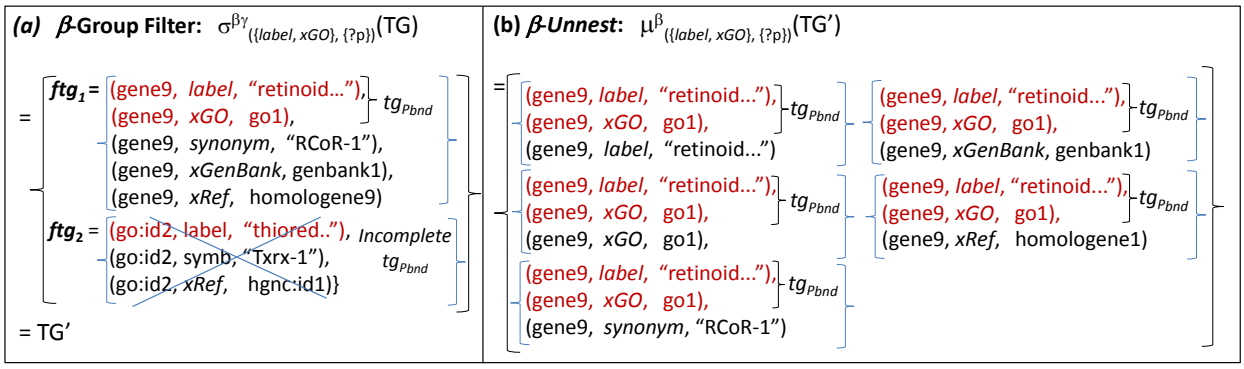



**Figure 5: NTGA logical operators to evaluate unbound-property star-patterns**

may also contain triples containing other property types. For example, given $P_{bnd} = \{label, xGO\}$, triplegroup $ftg_1$ forms a valid result for the $\sigma^{\beta\gamma}$ expression in Figure 5(a). However, $ftg_2$ does not contain a matching triple for the bound property $xGO$ and hence gets filtered out.

DEFINITION 2. *($\beta$ **unnest**) Given a set of triplegroups $TG$ and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$, the **unnest** operator $\mu^{\beta}$ creates a set of triplegroups that are exact matches to $St_u$. Specifically,*

$$\mu^{\beta}_{(P_{bnd},P_{unbnd})}(TG) := \{\ tg_i = \{tg_{P_{bnd}} \cup t_i\} \mid tg_{P_{bnd}}, t_i \subseteq tg, \\ tg_{P_{bnd}}.props() = P_{bnd}, tg \in TG\ \}$$

In other words, the $\beta$-unnest operator extracts subsets of triples in a triplegroup $tg$ that match the different pattern combinations corresponding to the unbound-property star-pattern. Figure 5(b) shows the 5 perfect triplegroups that are produced by $\beta$-unnesting the triplegroup $ftg$ in Figure 5(a), each containing a subset of triples $tg_{P_{bnd}}$ matching the set of bound properties $P_{bnd}$, and a triple $t_i$ that matches the unbound-property triple pattern.

LEMMA 1. *Given a triple relation $T$ and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ such that the set of bound properties $P_{bnd} = \{P_1, P_2, ..., P_k\}$ and $P_{unbnd}$ represents a single unbound property, the following equivalence holds:*

$$(T_{P_1} \bowtie ... \bowtie T_{P_k} \bowtie T) \cong \mu^{\beta}_{P_{bnd}}(\ \sigma^{\beta\gamma}_{(P_{bnd},P_{unbnd})}(\ \gamma_s(T)))$$

**Proof:** Let $T_{St_u}$ and $TG_{St_u}$ represent the set of tuples and triplegroups produced by evaluating an unbound-property star-pattern $St_u$ using relational joins and NTGA respectively. We need to prove that all tuples in $T_{St_u}$ are produced using NTGA. We prove by contradiction. Let us assume that there exists a tuple $tup_s \in T_{St_u}$ with subject $s$ that cannot be produced using triples in $tg_s$. This can happen only if $\exists$ a triple $t_i \in tup_s$ such that $t_i \notin tg_s$. Firstly, since $t_i \in tup_s$, we know that the subject of $t_i$ is $s$. If $t_i.\texttt{props}() \in P_{bnd}$, since $t_i$'s subject is $s$, the $\sigma^{\beta\gamma}$ ensures that $t_i \in tg_s$. If $t_i.\texttt{props}() \notin P_{bnd}$, then $\sigma^{\beta\gamma}$ still retains $t_i$ since its subject is $s$. Hence, $t_i \in tg_s$. The only other case is when a triple $t_i$ plays multiples roles (matches both bound and unbound parts) which are implicitly represented in our data model. We rely on the correctness of the $\mu^{\beta}$ operator (illustrated earlier but proof omitted for brevity) to complete the proof.

**Generalization to Multiple Unbound Properties.** The $\beta$-unnest operator can be generalized to star-patterns containing multiple unbound-property triple patterns. Let $P_\alpha$, $P_\beta$,...,$P_m$ represent the $m$ unbound properties in a star-pattern. Then the $\beta$-unnest operator results in a set of triplegroups $\{tg_{\alpha\beta...m}\}$ each containing the bound-property subset $tg_{P_{bnd}}$ and $m$ triples, one each matching the unbound properties $P_\alpha$, $P_\beta$,...,$P_m$.

$$\mu^{\beta}_{(P_{bnd},\{P_\alpha,P_\beta,..,P_m\})}(TG) := \{\ \{tg_{P_{bnd}} \cup t_\alpha \cup t_\beta \cup ... \cup t_m\}\ \}$$

such that $tg_{P_{bnd}} \subseteq tg$ is the bound-property subset, i.e., $tg_{P_{bnd}}.props()$ = $P_{bnd}$, and triples $t_\alpha, t_\beta, ...t_m \subseteq tg \in TG$.

# 4. TRANSLATION TO MAPREDUCE PLANS

The logical operators proposed in the previous section are integrated into RAPID+ [17] (an NTGA-based extension of Apache Pig). The query compilation process in RAPID+ begins with plans of logical operators, which are compiled to plans of physical operators, which could either be a single function or a function pair corresponding to the map and reduce phases of the logical operator. The MR plan is an assignment of physical operators to MR cycles.

The MR plan for an unbound-property query, executes the $\beta$-group-filtering using the `TG_UnbGrpFilter` ($\sigma^{\beta\gamma}$) operator in the reduce of the `TG_GroupBy`. This is followed by the $\beta$-`unnest` ($\mu^{\beta}$) operator that produces a set of perfect triplegroups. Thus, both `TG_UnbGrpFilter` and `unnest` can be executed in the reduce of `TG_GroupBy` in a single MR cycle ($MR_1$). We call this as *eager* $\beta$-unnesting of triplegroups, represented in Figure 6(a). The joins between the triplegroups matching the different subpatterns can be computed using NTGA's `TG_Join` operator in the subsequent $MR$ cycles. At the end of $MR_1$ for this strategy, we have intermediate results (perfect triplegroups for the star pattern subqueries) that contain redundancy with respect to the bound-properties. This increases the cost of $MR_1.R_{Write}$ and HDFS read ($MR_i.M_{Read}$) and shuffle costs ($MR_i.MR_{Shuffle}$) for subsequent cycles $MR_i$ that process the output of $MR_1$. Therefore, optimization strategies to minimize the redundancy in intermediate results of the star-join computation phase would be useful to generate cost-effective MapReduce workflows.

## 4.1 Optimization using $\beta$-Unnesting Strategies

The intuition is to concisely represent the result of an unbound-property star-pattern as far along the MR workflow as possible. Unbound-property query structures such as $B4$ in Figure 8 do not involve further joins based on the bindings of the unbound-property triple pattern, and thus can remain in its (nested) implicit representation till the end of the MR workflow. Query structures such as our example query $Q1$ participate in joins based on the Object column of the unbound-property triple pattern. Hence, the star-join results for such star subpatterns need to be $\beta$-unnested before the join, since the map phase of `TG_Join` tags the triplegroups based on the join key and partitions them to different reducers. We propose evaluation strategies to delay the $\beta$-unnesting of triplegroups.

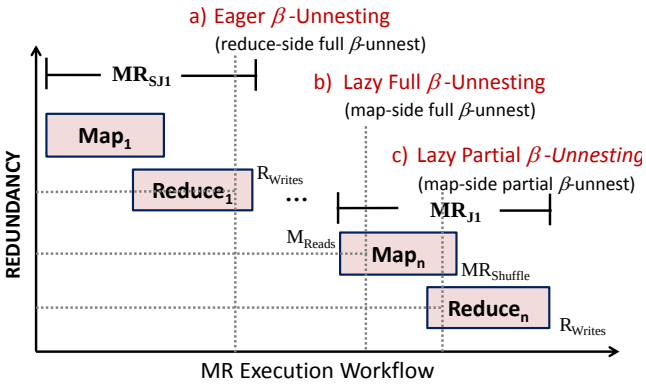



**Figure 6: (a) eager $\beta$-unnest of a triplegroup during star-join, (b) lazy full and (c) lazy partial $\beta$-unnest in later join phase**

**Lazy Map-side $\beta$-Unnest:** The $\beta$-unnesting of triplegroups can be delayed to a MR cycle that requires join on an unbound-property triple pattern, such as cycle $MR_{J1}$ in Figure 6(b). Specifically, we push the $\beta$-`unnest` operator to the map phase of the corresponding `TG_Join` operator. We refer to the new physical operator as `TG_UnbJoin` (reduce phase remains same as `TG_Join`). By delaying the $\beta$-unnesting of triplegroups, we can minimize the redundancy in results of the star-join computation phase, and hence avoid unnecessary writes, reads, and shuffle costs for all subsequent intermediate MR phases. However, the $\beta$-`unnest` operator expands the map output of `TG_UnbJoin`, which impacts the shuffling costs. Assuming that `TG_UnbJoin` is assigned to the $k$th MR cycle $MR_k$ in the workflow, then the redundancy in map output impacts $(MR_k.M_{Write} + MR_i.MR_{Sort} + MR_i.MR_{TR})$.

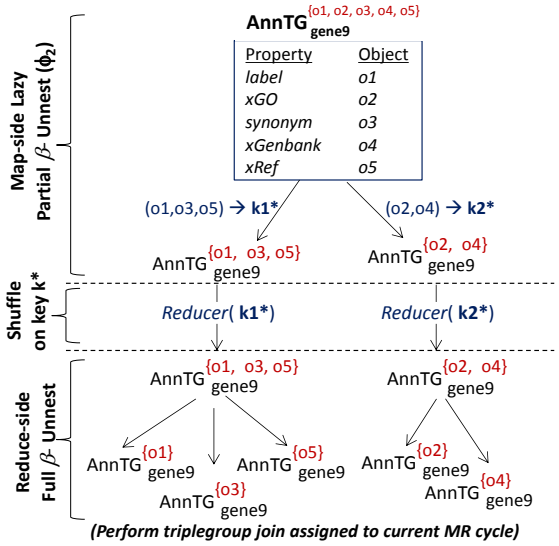



**Figure 7: Lazy partial $\beta$-unnesting ($\phi_2$)**

**Lazy Map-side Partial $\beta$-Unnest:** We illustrate this strategy using Figure 7 . In order to support efficient look-up of (Property, Object) pairs in a triplegroup, we use an optimized internal representation scheme (extended multi-map) represented here as $AnnTG$, that concisely represents annotated triplegroups. Example annotated triplegroup $AnnTG^{\{o1,o2,o3,o4,o5\}}_{gene9}$ in Figure 7 represents the subject triplegroup $ftg_1$ (Figure 5(a)) which is a valid match for the unbound-property star subpattern $SJ_2$ in query $Q1$. Annotated TG $AnnTG^{\{o1,o2,o3,o4,o5\}}_{gene9}$ contains 2 bound-property triples (matching $label$ and $xGO$) and 5 triples matching the unbound-property triple pattern. A $\beta$-unnest operation produces 5 triplegroups (all containing the same bound-property component) that form a part of the map output for $MR_k$. The default partitioning scheme in Hadoop assigns the map output tuples to a reducer $r$ based on the hash value of the join key, i.e., $hash(joinKey)\%r$. In the case that we have just 2 reducers, it is possible that triplegroups containing redundant bound-property component are partitioned and assigned to the same reducer based on the join keys (object of triples in the unbound-property component). For example, $AnnTG^{\{o1\}}_{gene9}$ and $AnnTG^{\{o3\}}_{gene9}$, may be assigned to the same Reducer, e.g., *Reducer1*. The redundancy in the map output of $MR_k$ can be minimized if triplegroups that are eventually assigned to the same reducer are concisely represented during the shuffle phase, i.e., they are not $\beta$-unnested completely. By avoiding a part of the $\beta$-unnesting, we can reduce the size of map output, and hence reduce the shuffling costs. We propose a partial $\beta$-unnesting strategy that creates a set of triplegroups that each contain the bound-property component $tg_{P_{bnd}}$, and a subset of the unbound-property component $tg_{P_{unbnd}}$.

DEFINITION 3. ***(partial $\beta$-unnest)*** *Given a set of triplegroups TG, an unbound-property star-pattern $St_u$=$\{P_{bnd}, P_{unbnd}\}$, and a partition function $\phi_m$ that partitions the triples in $tg_{P_{unbnd}}$ into $m$ partitions, the **partial-$\beta$-unnest** operator $\mu^{\beta'}$ produces a set of triplegroups such that:*

$$\mu^{\beta'}_{(P_{bnd},\phi_m)}(TG) := \{\ tg^i = \{tg_{P_{bnd}} \cup partition_i\}\ \}$$

where

- $\forall\ tg \in TG$, the bound-property subset $tg_{P_{bnd}} \subseteq tg$ such that $tg_{P_{bnd}}.props() = P_{bnd}$.
- A function $\phi_m$ assigns a triple $t_j \in tg_{P_u} \subseteq tg$ to $partition_i$, i.e., $\phi_m : t_j \rightarrow partition_i$, where $i \in \{1, 2, ..., m\}$.

The function $\phi$ partitions the triples in $tg_{P_{unbnd}}$ into $m$ buckets based on the value of the join key. Essentially, $\mu^{\beta'}$ produces a maximum of $m$ triplegroups for each triplegroup $tg \in TG$. For example, a partial $\beta$-unnest on $AnnTG^{\{o1,o2,o3,o4,o5\}}_{gene9}$ in Figure 7 using the partition function $\phi_2$ produces 2 triplegroups - $AnnTG^{\{o1,o3,o5\}}_{gene9}$ and $AnnTG^{\{o2,o4\}}_{gene9}$ respectively. This implies that $\phi_2(o1) = \phi_2(o3) = \phi_2(o5) = k1*$. Similarly, $AnnTG^{\{o2\}}_{gene9}$ and $AnnTG^{\{o4\}}_{gene9}$ are assigned to the same partition and hence remain implicitly represented as a single triplegroup. The redundant content in the map output is now a function of the partition range $m$. The partially $\beta$-unnested triplegroups are tagged and assigned to the reducers based on the partition key $k*$. Triplegroup join with lazy partial $\beta$-unnest is implemented as a new physical operator, `TG_OptUnbJoin`. Figure 6(c) represents how the I/O footprint can be reduced by partial and delayed $\beta$-unnesting at map phase of $MR_{J1}$.

### 4.1.1 Algorithms For Physical Operators:

Algorithm 1 gives an overview of the job workflow for two key phases in the NTGA plan – $Job_1$, that computes 'matching' triplegroup equivalence classes that match all star subpatterns in the query, and $Job_i$, that computes the join between the triplegroup equivalence classes.

$Job_1$**: *Compute 'matching' TG equivalence classes*.** The input to this job is a set of 3-tuples (triples) in the RDF database, and the output is a set of annotated triplegroups $AnnTG$ that match the star subpatterns in the query. In the map phase, each tuple is tagged based on the Subject component. In the reduce phase, all

**Algorithm 1:** MR job workflow for NTGA plan

$Job_1$: Compute 'matching' triplegroup equivalence classes
**Map:**
$TG\_GroupBy$.Map(Tuples $T$);
**Reduce:**
$TG \leftarrow TG\_GroupBy$.Reduce($Sub$, List <Tuples>);
$TG' \leftarrow TG\_UnbGrpFilter(TG, <EC, \{P_{bnd}, P_{unbnd}\}>)$;
$Job_i$: Join between triplegroup equivalence classes
**Map:**
$TG\_OptUnbJoin$.Map($TG'$) //partial $\beta$-unnest
or $TG\_UnbJoin$.Map($TG'$) //$\beta$-unnest
**Reduce:**
$TG'' \leftarrow TG\_OptUnbJoin$.Reduce($TG'$);
or $TG'' \leftarrow TG\_UnbJoin$.Reduce($TG'$);

tuples corresponding to the same Subject component $Sub$ are processed in the same reduce(), producing subject triplegroups. This is followed by a group-filtering phase to filter out triplegroups that violate the structural constraints in the query. Algorithm 2 shows the pseudocode for the $\beta$ group-filtering operator, `TG_UnbGrpFilter`. The (Property, Object) pairs in a triplegroup (`tempMap` in line 1), are matched with all equivalence classes (star subpatterns) in the query (line 2). For each matching equivalence class $EC$, the bound properties $P_{bnd}$ are extracted (line 4). The tuples in the group are considered relevant to the query only if they contain all bound properties (lines 5-9). If the matched equivalence class contains an unbound-property, the resultant $AnnTG$ contains all the (Property, Object) pairs for subject $Sub$ (lines 6-7). If the matched equivalence class does not contain any unbound-property, only the relevant (Property, Object) pairs that match the bound properties are retrieved into the resultant triplegroup (line 8). Essentially, a group of tuples that does not contain the required set of bound properties for any of the star subpatterns in the query is filtered out.

**Algorithm 2:** `TG_UnbGrpFilter`

$\beta$-**GrpFilter** *(tg, ECList:<EC,*$\{P_{bnd}, P_{unbnd}\}$*>)*;
1 $tempMap \leftarrow$ extract triples in $tg$;
2 $matchedECList \leftarrow match(tempMap, ECList)$;
3 **foreach** $EC \in matchedECList$ **do**
4 $P_{bnd} \leftarrow$ extract bound properties in $EC$;
5 **if** $P_{bnd} \subseteq tempMap.keySet$ **then**
6 **if** $EC$ *contains unbound property* **then**
//$\beta$ group filtering
7 $propMap \leftarrow tempMap$ ;
**else**
//Extract only bound properties in EC
8 $propMap \leftarrow$ extract $P_{bnd}$ entries from $tempMap$;
9 emit $\langle AnnTG(Sub, EC, propMap)\rangle$;

$Job_i$: ***Join between TG equivalence classes.*** The input to this phase is a set of annotated triplegroups, belonging to the two equivalence classes whose join is to be computed. The output is a set of annotated triplegroups, representing the joined result between the two equivalence classes. Based on the amount of redundancy in intermediate results due to the unbound-property star subpattern, a decision is made to either enable a partial or full $\beta$-unnest of the map output. Star subpatterns where the unbound-property is associated with a (partially) bound object, are not likely to cause redundancy, and hence a full $\beta$-unnest is enabled (`TG_UnbJoin` operator). For all other cases, the `TG_OptUnbJoin` operator is used.

Algorithm 3 shows the map-reduce functions for the operator `TG_OptUnbJoin` that integrates lazy partial-$\beta$-unnest operation. In the map phase, the annotated triplegroups that join on Subject are tagged using the Subject's partition key $k$* computed using $\phi_m$ (lines 1-3). For joins on Object, the AnnTG is partially $\beta$-unnested

**Algorithm 3:** `TG_OptUnbJoin`

**Map** *(key:null, val: AnnTG atg)* ;
1 **if** *join on* $Sub$ **then**
2 $k* \leftarrow \phi_m(atg.Sub)$;
3 emit $\langle k*, atg\rangle$ ;
**else if** *join on* $Obj$ **then**
//Partially $\beta$-unnest $atg$ using $\phi_m(Obj)$
4 $atgList \leftarrow$ **partial-$\beta$-unnest** $(atg, \phi_m)$ ;
5 **foreach** $partialMap \in atgList$ **do**
6 $k* \leftarrow$ extract $k*$ for $partialMap$ ;
7 emit $\langle k*, partialMap\rangle$ ;
**Reduce** *(key:*$k^*$*, val:List of AnnTGs* $TG'$*)* ;
8 $leftList \leftarrow \beta$-unnest leftEC AnnTGs from $TG'$;
9 $rightHash \leftarrow \beta$-unnest rightEC AnnTGs from $TG'$;
10 **foreach** $leftAnnTG \in leftList$ **do**
11 **foreach** $prop \in leftAnnTG.propMap$ **do**
//Handle multi-valued property
12 $objList \leftarrow$ extract $prop$'s objects from $leftAnnTG$ ;
13 **foreach** $joinKey \in objList$ **do**
14 $rightAnnTG \leftarrow rightHash$.get($joinKey$) ;
15 emit $\langle$ joinTGs($leftAnnTG, rightAnnTG$)$\rangle$;

using the `partial-`$\beta$`-unnest` operation. The `partial-`$\beta$`-unnest` operator splits the (Property, Object) pairs in the triplegroup $atg$ based on the Object's partition key resulting in a list of partially-unnested AnnTGs ($atgList$ in line 4). A map output tuple is generated for each partially-unnested AnnTG, tagged by its partition key $k$*(lines 5-7). The replication factor $Rep$ is now a function of $\phi_m$. In the reduce phase, all AnnTGs corresponding to the same group key $k$* but different join keys are processed in the same reduce(). In order to selectively join them based on the original join key, the AnnTGs corresponding to the right relation ($rightEC$) are $\beta$-unnested into perfect triplegroups and hashed based on the join key ($rightHash$ in line 9). The algorithm iterates through each AnnTG in the left relation ($leftEC$ in line 8), and probes the hashed relation ($rightHash$) based on the Object value (join key) for each property (lines 10-14). Multi-valued properties have multiple Object values and the probing is done for each value (lines 12-13). When a match is found, the two AnnTGs are joined (line 15) as per the definition of `TG_Join`. The partition factor used by $\phi$ depends on the size of input, potential redundancy factor, and average number of tuples that can be processed by a reducer.

# 5. EVALUATION

We evaluated the proposed algebraic optimization techniques on both real-world and synthetic datasets, and compared it with two popular relational-style MapReduce systems, Apache Pig and Hive. For NTGA, we evaluated two approaches for processing unbound-property graph pattern queries – *EagerUnnest* (Section 4), and the optimized *LazyUnnest* with map-side lazy $\beta$-unnesting. Experiments were conducted on NCSU's VCL [33], where 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. 60 and 80-node Hadoop clusters (block size set to 256MB, 1GB heap-size for child jvms) were used with Pig release 0.11.1, Hive 0.10.0 and Hadoop 0.20.2. Only 20GB disk space was available per node, requiring large clusters to support large scale data, i.e., the 80-node Hadoop cluster made available ∼1.6TB HDFS disk space. Results recorded were averaged over three trials.

**Choice of Systems:** Both Pig and Hive evaluate star-joins in a single MR cycle (one-star-join-per-cycle), resulting in same length workflows for all queries. Hive enables shared-scan of input relations within an MR cycle, thus minimizing the overall HDFS

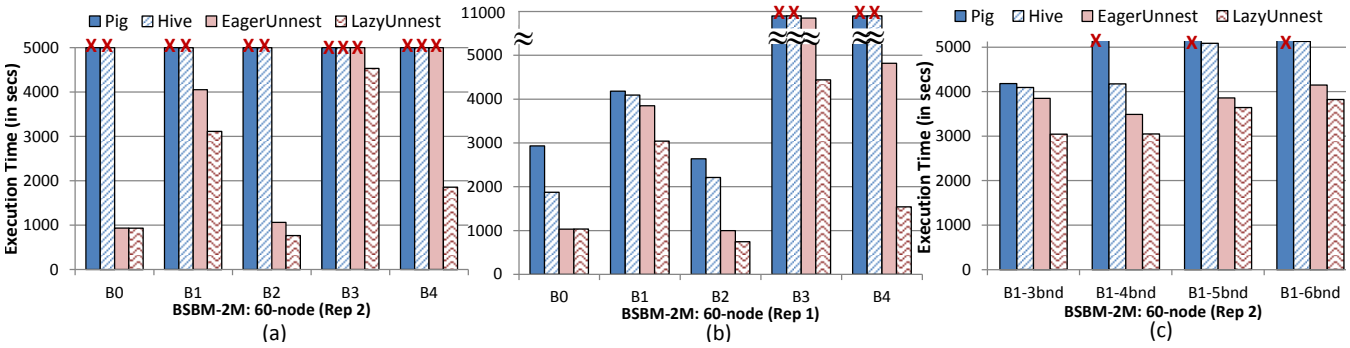



**Figure 9: Performance with varying unbound-property star patterns with (a) replication factor 2 (b) replication factor 1 (c) Performance with varying size of bound-property component (BSBM-2M, 172GB, 60-node)**

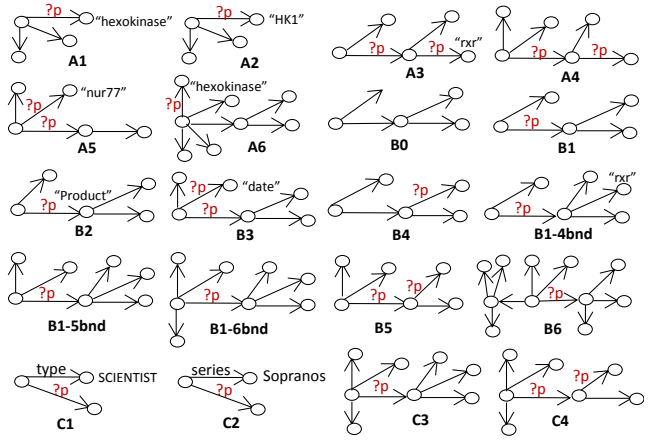



**Figure 8: Testbed unbound-property RDF queries**

reads. Pig can execute independent MR cycles concurrently, which is beneficial while evaluating multiple star subpatterns. NTGA approaches produce shorter workflows (all-star-joins in single MR cycle) when compared to Hive/Pig for queries with multiple star subpatterns. A triple relation is loaded as a 3-column table in Hive, where as Pig (and NTGA) process them as flat files. In Pig, the `SPLIT` operator is used to generate vertically-partitioning relations. HadoopRDF [15] does not currently support unbound-property queries and is not included for evaluation. Systems such as HadoopDB [14] scale well but rely on a heavy pre-processing phase that is more suitable for private clusters and less-evolving data. We focus on on-demand and pay-as-you-go workloads that involve quick exploration of datasets to get a sense of the data.

**Testbed - Dataset and Queries:** Real-world life sciences data from Bio2RDF [9] was used for evaluation. The queried biological data warehouse integrated 24 datasets, consisting of a total of ∼4.7 billion triples (615GB in n-triple format). Two other real-world datasets, DBPedia Infobox (DbInfobox) [7] dataset of size 4.4GB (33.74M triples: 20.5M properties, 13.23M types) and the Billion Triple Challenge 2009 dataset (BTC-09) [1] of size 193GB (1.5B triples), were also used for evaluation. More than 45% of properties in both datasets are multi-valued with varying multiplicity. Two synthetic datasets generated by the BSBM [11] data generator tool – BSBM-1M (85GB dataset with 1 million Products, total ∼370 million triples) and BSBM-2M (172GB dataset with 2 million Products, total ∼700 million triples) were used for scalability study. The evaluation tested unbound-property queries with varying selectivity, varying join structures (single join to more complex structures with multiple star subpatterns) that are represented in Figure 8. Graph patterns in queries `A1`-`A6` have been extracted from Bio2RDF demo queries [2]. Additional details about the evaluated queries, along with the Pig / Hive scripts, are available on the project website [3].

**Varying join structures (`B1`-`B6`):** Scalability experiments were conducted to evaluate different join structures with varying number of unbound-property triple patterns, and varying arity of star subgraphs. Figures 9(a) and (b) show a performance comparison of Pig, Hive, and the NTGA approaches for two-star queries with no unbound properties (`B0`), one unbound-property triple pattern with join on unbound object (`B1`), one unbound property associated with a partially-bound object (`B2`), two unbound-property triple patterns in the same star with one partially-bound object (`B3`), and an unbound-property triple pattern (`B4`). Pig / Hive evaluate all three queries using 3 MR jobs (one per star-join), while NTGA evaluates them in 2 MR jobs. The queries involve a multi-valued property *prodFeature* that impacts redundancy.

In order to avoid data loss during node failure, fault-tolerant systems such as Hadoop rely on replication of data blocks on multiple nodes using a configurable parameter (dfs.replication). Initial set of experiments were conducted using a replication factor of 2 for the larger dataset BSBM-2M on a 60-node cluster (1.6TB disk space, 20GB per node). The results, shown in Figure 9(a), demonstrate how critical it is to concisely represent intermediate results and eliminate redundancy when possible. Missing bars marked with 'X' represent failed execution. Pig / Hive approaches failed during the last job (join between stars) for all 5 queries due to shortage of disk space. While *EagerUnnest* successfully executed for `B0`, `B1`, and `B2` by concisely representing subgraphs involving multi-valued properties, it failed for queries `B3` and `B4`. This is because the double unbound-property triple patterns in `B3` result in materialization of large intermediate results during the star-join computation phase, and we see the benefit of pushing the $\beta$-unnesting to a later phase (*LazyUnnest*) in executing this query. Similarly, for query `B4`, *LazyUnnest* successfully executes by materializing concise intermediate results, while other approaches fail.

In order to analyze the performance of the different approaches on the larger dataset, the same set of queries were repeated after reducing the HDFS replication factor to 1. Figure 9(b) shows the results comparing the performance of the approaches for BSBM-2M on the same 60-node cluster. In general, we see the benefit of the NTGA approaches for all queries. Query `B0` shows a baseline case with all bound properties where Hive and NTGA approaches outperform Pig due to scan-sharing. Further, NTGA approaches concisely represent results containing multi-valued property which leads to I/O savings. For query `B1` (join on unbound-property triple pattern), lazy partial $\beta$-unnesting reduces the shuffle

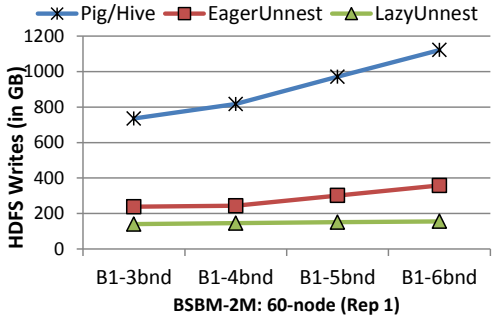



**Figure 10: Total HDFS writes with varying size of bound-property component (BSBM-2M, 60-node)**

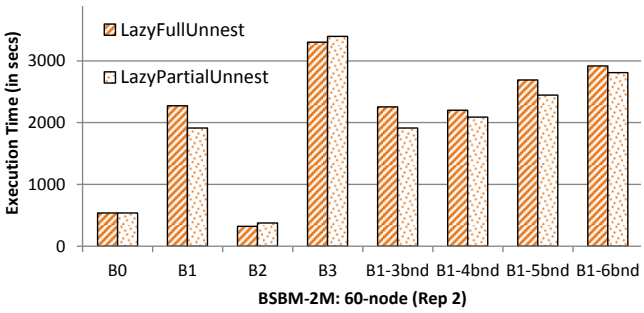



**Figure 11: Lazy Full vs. Lazy Partial Unnesting: A comparative study of savings and overhead in MR cycle $MR_{J1}$**

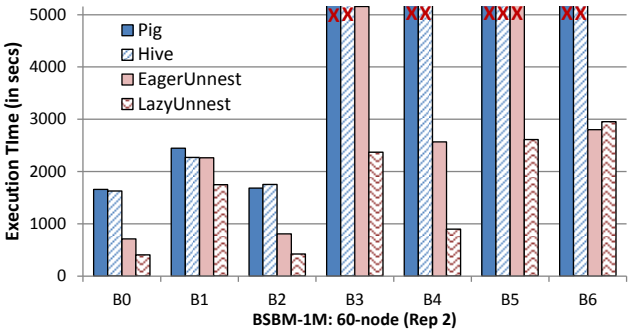



**Figure 12: Performance comparison (BSBM-1M, 85GB)**

costs and is 21% faster than eager $\beta$-unnesting (27% faster than Pig and 26% faster than Hive). For query `B2`, all approaches evaluate the filter on the partially-bound object associated with the unbound-property triple pattern in the initial map phase, and from there on, the execution is similar to the baseline query `B0`. As in the case of replication factor 2, Hive and Pig failed again for `B3` and `B4`. The star subpattern with double unbound-property triple patterns (one with partially-bound object) in `B3`, is concisely represented in *LazyUnnest* with 80% less HDFS writes than *EagerUnnest*. In queries, such as `B4`, where the unbound-property triple pattern does not participate in join between stars, the lazy $\beta$-unnesting strategy keeps the result compact till the end, thus saving on intermediate disk reads / writes as well as final writes. Lazy $\beta$-unnesting using *LazyUnnest* results in 61% less HDFS writes than *EagerUnnest*, and overall has a 68% gain in performance times over the eager $\beta$-unnesting approach.

**Choice of Lazy $\beta$-Unnesting Strategies:** Testbed queries consist of varying structure of unbound-property triple patterns. For example, unbound-property triple patterns in queries `B2` and `B3` have partially-bound objects, i.e., the user does not know the exact property relationship but knows something about the object. In such cases, it is likely that the number of triples matching the unbound-property triple pattern are reduced and hence the associated star-join is more selective, i.e., results in less number of pattern combinations when compared to same triple pattern with an unbound object. Other queries such as `B1` consist of unbound-property triple pattern with an unbound object. Though lazy $\beta$-unnesting is beneficial for all cases, we wanted to study benefits and overhead of lazy full and lazy partial $\beta$-unnest strategies. Figure 11 shows execution times for the last MR cycle ($MR_{J1}$) where the join involving the unbound-property triple pattern is computed. Since the size of input for $MR_{J1}$ is same for both approaches, this analysis allows us to zoom into the map-side overhead for full and partial $\beta$-unnest, savings in shuffle costs, and analysis of reduce-side overhead in the case of partial-$\beta$-unnest. Our experiments show that a lazy full $\beta$-unnest may be sufficient for unbound-property queries with partially bound objects (queries `B2` and `B3`). However, unbound-property queries with an unbound object ($B1$ series), benefit from partial-$\beta$-unnest. Other experiments were corroborative to these findings, and hence the *LazyUnnest* approach reported in this section evaluate lazy full-$\beta$-unnest for unbound-property queries with partially-bound-object patterns, and lazy partial-$\beta$-unnest for those with unbound-object patterns.

**Varying number of bound-property edges:** Unbound-property queries with bound-property triple patterns varying from 3 (`B1-3bnd`) to 6 (`B1-6bnd`) were evaluated. Figure 10 shows the total amount of HDFS writes for Pig, Hive and the NTGA approaches for the test queries evaluated on a 60-node cluster with BSBM-2M. In general, the increase in the number of bound-property components results in a gradual increase in the size of reduce output for Pig and Hive, while lazy $\beta$-unnesting keeps the result concise till the end of map phase of the last MR job ($Job_2$). The relational approaches produce 10 combinations of the bound component for the test queries since the relational arity of the subgraph that matches the unbound-property subpattern is 10. However, *LazyUnnest* compactly captures all the required combinations, resulting in approx. 80 to 86% less HDFS writes than Hive / Pig for queries `B1-3bnd` to `B1-6bnd`, respectively. Additionally, the reduce output for the NTGA approaches remain almost constant for such query patterns, which allows more flexible exploration of large datasets. Figure 9(c) shows a comparion of the execution times for all approaches. Note that Pig failed for all queries beyond three bound-property subpatterns. *LazyUnnest* ($\phi_{1K}$) consistently outperformed the other approaches, running about 25% faster than Hive.

**Varying size of RDF graphs:** Figure 12 shows the evaluation of the BSBM queries using BSBM-1M (85GB) on the 60-node cluster (HDFS replication factor 2). NTGA approaches successfully executed for all datasets, with up to 80% less HDFS writes after the star-join computation phase for query `B1` when compared to Hive. Once again it was observed that both Pig and Hive failed for queries `B3` and `B4` due to insufficient disk space. This is due to the high redundancy in star-join result that ripples into the next MR job, impacting the scan and I/O costs. For query `B2`, *LazyUnnest* outperforms all other approaches, executing about 75% faster than both Pig and Hive. *LazyUnnest* reduces the redundancy in intermediate results, and thus improves the execution time of the eager $\beta$-unnesting approach (*EagerUnnest*) by 54% (65%) for query `B3` (`B4`). Hive / Pig failed to execute for more complex queries such as `B5` and `B6`. These sets of experiments demonstrate the benefit of the proposed strategies in mitigating the effect of redundancy on MapReduce processing costs.

**Real-world Unbound-property Queries (`A1-A6`):** Figure 13

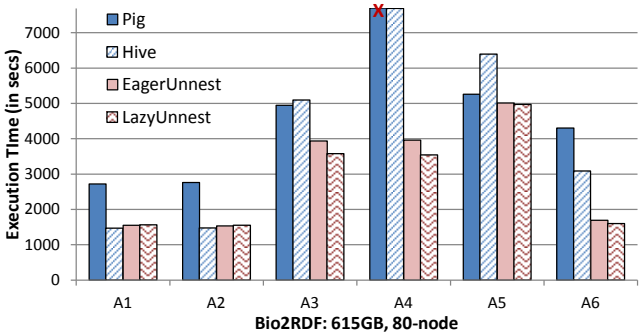



**Figure 13: Evaluation of real-world unbound-property queries (Bio2RDF Life Sciences Dataset)**

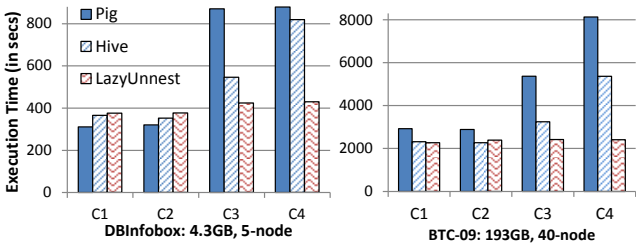



**Figure 14: Evaluation of real-world unbound-property queries (DBInfobox and Billion Triple Challenge'09)**

shows a performance comparison of Pig, Hive, and the two NTGA approaches for Bio2RDF queries `A1`-`A6` on a 80-node Hadoop cluster. Queries `A1` and `A2` have one star subpattern with one unbound-property triple pattern associated with partially-bound objects. For query `A1`, while Hive / Pig approaches produce all combinations of subtuples matching the bound property with triples matching the unbound property (∼63K tuples), *EagerUnnest* produces ∼7K triplegroups that concisely represent subtuples with multi-valued properties. *LazyUnnest* achieves more concise representation of all combinations corresponding to the unbound-property star pattern and produces only ∼3K triplegroups. The impact of the savings in HDFS writes due to elimination of redundancy in intermediate results, becomes more clear with the two-star queries (`A3`-`A6`).

Queries `A3` and `A4` contain an unbound-property in each of the two star subpatterns (one with partially-bound object). While Pig / Hive materialize 26GB of intermediate results in the star-join computation phase for query `A3`, the NTGA approaches write only about 1.3GB of data to the HDFS, contributing to the 32% performance gain over Hive while computing the star subpatterns. The *LazyUnnest* results in reduced HDFS writes in $MR_1$, and reduced scan costs and shuffling costs in $MR_2$, resulting in additional 18% performance gain over *EagerUnnest* in $MR_2$. For query `A4`, Pig initiates 4 MR jobs (initial map-only job to read entire input and compress it, 2nd and 3rd MR jobs to compute the two star patterns, and the 4th job to join the stars). However, Pig approach failed (marked as 'X') due to lack of HDFS space while executing the last job. Again, there is a huge savings in terms of HDFS writes, with *EagerUnnest* and *LazyUnnest* producing only 1.8GB and 0.6GB of intermediate results, respectively, after the initial star-join phase, as opposed to 152GB of writes in Hive. An important factor that results in large intermediate results with relational-style processing, is the redundancy due to the presence of large number of high multiplicity properties in biological datasets (representative of real-world datasets). For `A4`, *EagerUnnest* and *LazyUnnest* approaches are 48% and 53% faster than Hive, respectively.

Query `A5` contains a star pattern with two unbound-property triple patterns – one whose object matches a gene "nurr77", and the other with an unbound object, connecting the star to a single edge retrieving the *label* property type. Hive executes `A5` using 2 MR jobs, with both jobs requiring a full-table scan. NTGA approaches also execute using 2 MR jobs but with one full-table scan, resulting in overall savings of about 1400s (22% gain) over Hive. The single unbound-property triple pattern in query `A6` partially binds the object to "hexokinase". While Hive uses 3 MR jobs, including 2 for the star-join computation, Pig uses an extra map-only job to compress the input (total 4 jobs). NTGA's *LazyUnnest* approach shows a benefit of up to 48% over Hive.

**DBPedia Queries (`C1-C4`):** Additional experiments were conducted on varying sizes of real-world datasets, 4.3GB DBInfobox dataset (5-node cluster) and 193GB BTC-09 dataset (40-node cluster) as shown in Figure 14. Four different query structures were used. `C1` and `C2` are simple queries with single join that retrieve all information about Scientists (unselective) and Sopranos TV series (selective). In the case of DBInfobox dataset, since the data processed is quite small, the benefit of the NTGA approach is not seen for the first two queries. However, Pig does better than Hive since it processes two copies of the input relation, and hence initiates double the number of mappers and reducers. `C3` and `C4` represent real-world scenarios during exploration where the relationship between entities (star subpatterns) is unknown. NTGA approaches showed a performance gain of 20-22% and 50% over Hive and Pig respectively for query `C3`, and resulted in approx. 80% less HDFS writes than Hive. All four queries had redundancy factor greater than 0.6. In particular, `C4` which involved an unbound-property in each of the two star patterns showed a redundancy factor close to 0.89, and hence showed major improvement (50% gain over both Pig and Hive) with the lazy $\beta$-unnesting strategy.

Unbound-property queries on the BTC-09 dataset resulted in very large HDFS reads which negatively impacted Pig the most, due to its multiple scans per star-join. The scan-sharing across star patterns in NTGA resulted in 50% less HDFS reads for the two star queries. NTGA approaches resulted in 54% (25%) gains over Pig (Hive) for query `C3` with 1 unbound-property. The result of the star-join phase for `C4` (2 unbound properties) has redundancy factor of 0.93 (0.75GB) and increases to 0.98 (14GB) in the final output for Pig/Hive. The lazy $\beta$-unnesting strategy results in 98% less HDFS writes, and have 70% (55%) performance gain over Pig (Hive) for `C4`. In general, real-world data contained multiple multi-valued properties with varying multiplicity, and highly benefited by the generalized nested representation of triplegroups and lazy $\beta$-unnesting strategies while processing unbound-property queries.

# 6. CONCLUSION

We propose a scalable solution for processing unbound-property graph pattern queries on MapReduce, by minimizing the redundancy in intermediate results that adds avoidable costs while processing long execution workflows. The proposed approach uses a nested triplegroup model to implicitly represent the intermediate results and lazily 'unnest' them only when necessary. A combination of the two result in significant savings in intermediate HDFS reads and writes, which form a major portion of query processing costs on MapReduce. Additional savings in intermediate map-reduce data shuffling costs can be achieved by delaying a portion of the 'unnest' to the reduce phase. Experiments show promising results for different query join structures with varying selectivities. Future directions include exploring more complex structures with multiple unbound-property patterns as well as unbound-property queries with aggregation constraints.

## 7. REFERENCES


[1] Billion triple challenge. *http://challenge.semanticweb.org/*.

[2] Bio2RDF Demo Queries. `http://sourceforge.net/apps/mediawiki/bio2rdf/index.php?title=Demo_queries`.

[3] Scaling Unbound-Property Queries using MapReduce. `http://research.csc.ncsu.edu/coul/RAPID/UnboundPropQueries`.

[4] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[5] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2009.

[6] F. Afrati and J. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

[7] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. *ISWC/ASWC*, 2007.

[8] A. Bairoch, L. Bougueleret, S. Altairac, V. Amendolia, A. Auchincloss, G. A. Puy, K. Axelsen, D. Baratin, M.-C. Blatter, B. Boeckmann, et al. The universal protein resource (uniprot). *Nucleic Acids Research*, 36:D190–D195, 2008.

[9] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716, 2008.

[10] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene.apache.org/hadoop*, 11, 2005.

[11] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 2009.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm. ACM*, 2008.

[13] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *VLDB*, 2012.

[14] J. Huang, D. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *VLDB*, 4(11), 2011.

[15] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *TKDE*, 23(9), 2011.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, 2007.

[17] H. Kim, P. Ravindra, and K. Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *VLDB*, 4(12), 2011.

[18] H. Kim, P. Ravindra, and K. Anyanwu. Scan-sharing for optimizing rdf graph pattern matching on mapreduce. In *CLOUD*, 2012.

[19] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In *The Semantic Web–ISWC*. 2012.

[20] H. Lim, H. Herodotou, and S. Babu. Stubby: a transformation-based optimizer for mapreduce workflows. *VLDB*, 2012.

[21] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, 2011.

[22] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB*, 2010.

[23] M.-A. Nolin, P. Ansell, F. Belleau, K. Idehen, P. Rigault, N. Tourigny, P. Roe, J. M. Hogan, and M. Dumontier. Bio2rdf network of linked data. In *Semantic Web Challenge; ISWC*, 2008.

[24] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *VLDB*, 2010.

[25] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. Rao, V. Sankarasubramanian, S. Seth, et al. Nova: continuous pig/hadoop workflows. In *SIGMOD*, 2011.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[27] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *WWW*. ACM, 2012.

[28] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf, W3C Recommendation, 2008. *URL http://www.w3.org/TR/rdf-sparql-query*, 2010.

[29] P. Ravindra and P. Anyanwu. Nesting strategies for enabling nimble mapreduce dataflows for large rdf data. *Int. J. Semantic Web Inf. Syst.*, 10(1), 2014.

[30] P. Ravindra, H. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. *The Semantic Web: Research and Applications*, 2011.

[31] P. Ravindra, H. Kim, and K. Anyanwu. To nest or not to nest, when and how much: representing intermediate results of graph pattern queries in mapreduce based processing. In *SWIM*, 2012.

[32] K. Rohloff and R. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *PSI EtA*, page 4, 2010.

[33] H. Schaffer, S. Averitt, M. Hoit, A. Peeler, E. Sills, and M. Vouk. Ncsu's virtual computing lab: a cloud computing solution. *Computer*, 42(7), 2009.

[34] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. *ISWC*, 2008.

[35] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *VLDB*, 2008.

[36] S. Stefanova and T. Risch. Optimizing unbound-property queries to rdf views of relational databases. In *SSWS*, 2011.

[37] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2(2), 2009.

[38] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[39] X. Wang, C. Olston, A. Sarma, and R. Burns. Coscan: cooperative scan sharing in the cloud. In *SOCC*, 2011.

[40] S. Wu, F. Li, S. Mehrotra, and B. Ooi. Query optimization for massively parallel data processing. In *SOCC*, 2011.