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).

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 $T_{xGO}$ and $T_{label}$ with property types $xGO$ and $label$, respectively, the subquery $S_{J2}$ can be evaluated using relational joins ($T_{xGO}$ $\bowtie$ $T_{label}$ $\bowtie$ $T$). Figure 1 (right) shows the subrelations of $T$ participating in $S_{J2}$ and a snapshot of the star-join result. An issue with intermediate results in such cases is redundancy. For example, the result for $S_{J2}$ 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-

(c) 2015. Copyright is with the authors. Published in Proc. 18th International Conference on Extending Database Technology (EDBT), March 23-27, 2015, Brussels, Belgium: ISBN 978-3-89318-067-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0
diately 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 triple-groups, 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 \( S_{J_1} \), \( S_{J_2} \), and a third cycle to join the stars) as shown in Figure 1, while the NTGA would compute both \( S_{J_1} \) and \( S_{J_2} \) 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-
lation operations, the nested triplegroup data model can concisely represent intermediate results with multi-valued properties, e.g.,

\[
\{ \text{gene9, xGO, (go1, go9)} \} \quad \text{A single triplegroup representing} \\
\{ \text{gene9, label, retinoid...) \} \quad \text{two n-tuples t1 and t4} \\
\{ \text{gene9, synonym, RCoR-1) } \} \quad \text{by nesting object component}
\]

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. 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 \( \text{TripleGroups} \). For example, triples in the database can be modeled as a set of Subject \( \text{TripleGroups} \), each consisting of triples that share a common subject. For example, triplegroups \( t_{g1} \) and \( t_{g2} \) in Figure 2 represent subject triplegroups corresponding to triples sharing common subjects \text{gene9} and \text{homod2}, 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 \( S_{t1} \{ \text{label, gene_sym} \} \) and \( S_{t2} \{ \text{label, xGO, xRef} \} \). 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 ( \( \sigma \) ) 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.
Consider a set of triplegroups $\mathbf{TG} = \{\mathbf{tg}_1, \mathbf{tg}_2\}$ such that

$$\mathbf{tg}_1 = (\text{gene9}, \text{go9}, \text{go9}), \quad \mathbf{tg}_2 = (\text{gene9}, \text{go9}, \text{go9})$$

Notation Semantics

**TripleGroup Filter**

Enforces structural constraints in a star subpattern by matching the set of bound properties $P_{\mathbf{tg}}$ and eliminating triplegroups in $\mathbf{TG}$ that violate the required join structure (structure-based validation)

\[ \sigma (\text{tg}_1 \bowtie \text{tg}_2) \]

**TripleGroup Join**

Joins triplegroups $\mathbf{tg}_1$ and $\mathbf{tg}_2$ based on the conditions specified by triple patterns $p_1$ and $p_2$ respectively.

\[ \mathbf{M}^*(\mathbf{tg}_1, \mathbf{tg}_2) \]

Figure 2: Example NTGA Operators

in the star subpattern. Triplegroup $\mathbf{tg}_1$ is a valid match for $S_1$ and is said to belong to the equivalence class $T\mathbf{G}(\text{label}, \text{gene}_{\text{symb}})$ 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 $Q^r$ can be computed as follows:

\[ \sigma_{\text{ntg}}^r(\mathbf{tg}) \]

Joins between star subpatterns can be computed using the join operator (\(\bowtie\)) that is semantically equivalent to the relational join operator but is defined on triplegroups. The object-subject join between triplegroups $\mathbf{tg}_1$ and $\mathbf{tg}_2$ in a nested triplegroup $\mathbf{ntg}$ whose root is the triplegroup $\mathbf{tg}_1$ and child triplegroup is $\mathbf{tg}_2$. Before proceeding, we review the notion of *content-equivalence* that enables lossless translation between relational algebra and NTGA plans.

**Relational Algebra \(\Rightarrow\) NTGA Plans.** Triplegroups are *content-equivalent* (represented as \(\bowtie\)) to the set of $n$-tuples computed using a set of relational-style joins. Let $\mathbf{StP}$ be a star subpattern comprising of the set of bound properties \(P_1, P_2, \ldots, P_k\), and $\mathbf{TStP}$ be the join result of vertically partitioned subset relations $T_{P_1}, T_{P_2}, \ldots, T_{P_k}$.

Let $T_{\mathbf{StP}}(x)$ represent the subset of $T_{\mathbf{StP}}$ with subject $\mathbf{StP} = x$.

\[ T_{\mathbf{StP}}(x) = \sigma_{\text{ntg}}(T_{P_1} \bowtie T_{P_2} \bowtie \ldots \bowtie T_{P_k}) \]

Each tuple in $T_{\mathbf{StP}}(x)$ is of $3k$ arity (each property in $\text{StP}$ is associated with 3 columns). Let $p_i^r$ denote the projection of the $(\text{Sub. Prop. Obj})$ columns corresponding to the parent relation $T_{P_i}$ with bound-property $P_i$. Let $t_{\mathbf{tg}_1}$ represent the set of $n$-tuples formed by the triple relation, i.e.

\[ t_{\mathbf{tg}_1} = p_1^r(T_{\mathbf{StP}}(x)) \cup p_2^r(T_{\mathbf{StP}}(x)) \cup \ldots \cup p_k^r(T_{\mathbf{StP}}(x)) \]

In summary, the tuples in $T_{\mathbf{StP}}(x)$ can be vertically partitioned into 'triples' whose union is equivalent to a subject triplegroup $\mathbf{tg}_1$ in the NTGA data model. For our example data in Figure 2.

\[ t_{\mathbf{tg}_1} \cong \sigma_{\text{ntg}}(T_{\text{subject}} \bowtie T_{\text{gene}_{\text{symb}}}) \]

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 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 $\text{StP}_2$ containing a multi-valued property $\text{xGO}$ are implicitly represented using a single triplegroup $\mathbf{tg}_2$ 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 (Q1a, Q1b, Q2a, Q2b) and Object-Object join (Q3a, Q3b) between star patterns. Queries Q1b, Q2b, Q3b are variations of Q1a, Q2a, Q3a respectively, where each 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 (Q3a, Q3b), Sel-SJ-first still requires 3 MR cycles, but more importantly has very high HDPS 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 ($\sigma^r$) 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 3: Evaluation of different groupings of star-joins (MR: No. of MapReduce cycles, FS: No. of Full Scans)
\[ T_{St1}(S1) = \sigma_{Sub=S1} (TP1 \bowtie TP2 \bowtie T) = S1 P1 O1 S1 P2 O2 S1 P1 O1 \]
\[ = S1 P1 O1 \ldots O1 S1 P2 O2 S1 P2 O2 S1 P2 O2 S1 P3 O3 S1 P4 O4 \]

Figure 4: Transformation: n-tuples to a triplegroup

Essentially, the tuples in \( T_{St1} \) 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 \( tg \) 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 \( tg \) represent matches to the bound and unbound-property triple patterns in \( St1 \). This process basically describes a sequence of translation steps from the relational algebra to NTGA. In other words,
\[ T_{St1(s)} = \sigma_{Sub=s}(TP1) \bowtie \ldots \bowtie \sigma_{Sub=s}(TP_k) \bowtie \sigma_{Sub=s}(TP_n) = tg_{p_1} \bowtie \ldots \bowtie tg_{p_k} \bowtie tg_{p_n} \]

Conversely, for our example star-pattern \( St1 \) in Figure 4, tuples in \( T_{St1(S1)} \) are implicitly represented in \( tg_{S1} \) and can be produced by \( (tg_{p_1} \bowtie tg_{p_2} \bowtie tg_{p_3}) \). 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., \( T = (TP_1 \cup TP_2) \). Then by the distributivity of join over union, we have:
\[ T_{Sub=\alpha}(TP_1 \cup TP_2) = (T_{Sub=\alpha}(TP_1) \cup T_{Sub=\alpha}(TP_2)) \]

Evaluating \( St1 \) 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 \( P_{unbnd} \) with each property in the database. For example, if \( P_{unbnd} = \{P_1, P_2\} \) is the set of bound-properties in the star pattern and \( P = \{P_1, P_2, \ldots, P_n\} \) represents the set of all properties in the database. Then, the \( \sigma^\gamma \) expression is:
\[ \sigma^\gamma_{\{P_1, P_2, \ldots, P_n\}}(TP_{unbnd} \bowtie TP') \equiv (TP_{unbnd} \bowtie TP_{unbnd}) \]

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 \( tg \) 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 \( tg \) that are exact matches for any of the required pattern combinations. This is achieved by extracting the subset of triples corresponding to \( P_{unbnd} \) and generating their union with each triple in the unbound-property subset \( tg_{unbnd} \). 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 \( St_a = \{P_{unbnd}, P_{unbnd}\} \) 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 \( P_{unbnd} \). Specifically,
\[ \sigma^{\beta \gamma}_{\{P_{unbnd}, P_{unbnd}\}}(TG) = \{tg_{i} \in TG \mid P_{unbnd} \subseteq tg_{i,props()} \} \]

Essentially, \( \sigma^{\beta \gamma} \) ensures that triplegroups contain a matching triple for each of the bound properties in \( P_{unbnd} \). Additionally, triplegroups...
may also contain tripleps matching other property types. For example, given \( P_{\text{bound}} = \{ \text{label}, xGO \} \), triplegraph \( ftg_1 \) forms a valid result for the \( \sigma^{\beta}_{\text{tg}_{\text{bound}}} \) 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 triplegraphs \( TG \) and an \( \beta \)-unbound-property star pattern \( St_u = \{ P_{\text{bound}}, P_{\text{unbound}} \} \), the \( \beta \)-unnest operator \( \mu^\beta \) creates a set of triplegraphs that are exact matches to \( St_u \). Specifically,

\[
\mu^\beta_{(P_{\text{bound}}, P_{\text{unbound}})}(TG) := \{ \text{tg} \in TG \mid \text{tg}_{\text{bound}} \cup \{ \text{tg} \}_{\text{unbound}} \subseteq \text{tg} \}
\]

In other words, the \( \beta \)-unnest operator extracts subsets of triples in a triplegraph \( \text{tg} \) that match the different pattern combinations corresponding to the unbound-property star-pattern. Figure 5(b) shows the 5 perfect triplegraphs that are produced by \( \beta \)-unnestting the triplegraph \( ftg_2 \) in Figure 5(a), each containing a subset of triples \( \text{tg}_{\text{bound}} \) matching the set of bound properties \( P_{\text{bound}} \), and a triple \( \text{tg} \) that matches the unbound-property triple pattern.

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

\[
(T_{P_1} \bowtie \ldots \bowtie T_{P_n} \bowtie T_{St_u}) \cong \mu^\beta_{(P_{\text{bound}}, P_{\text{unbound}})}(\sigma^{\gamma}_{(P_{\text{bound}}, P_{\text{unbound}})}(\gamma_s(T)))
\]

**Proof:** Let \( T_{St_u} \) and \( TG_{St_u} \) represent the set of tuples and triplegraphs 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 \( t_{\text{unbound}} \in T_{St_u} \) with subject \( s \) that cannot be produced using triples in \( t_{\text{tg}} \). This can happen only if \( \exists \) a triple \( t \in t_{\text{unbound}} \) such that \( t \notin t_{\text{tg}} \). Firstly, since \( t \in t_{\text{unbound}} \), we know that the subject of \( t \) is \( s \). If \( t_{\text{tg}} \) matches 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 \( TG_{\text{Join}} \) tags the triplegraphs based on the join key and partitions them to different reducers. We propose evaluation strategies to delay the \( \beta \)-unnesting of triplegraphs.
Lazy Map-side $\beta$-Unnesting: The $\beta$-unnest is a sequence of operations that can be delayed to a MR cycle that requires join on an unbound-property triple pattern, such as cycle $MR_{1}$ in Figure 6(b). Specifically, we push the $\beta$-unnest operator to the map phase of the corresponding $\text{TG}_1$ operator. We refer to the new physical operator as $\text{TG}_1\text{UnbJoin}$ (reduce phase remains same as $\text{TG}_1$). By delaying the $\beta$-unnest 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 $\text{TG}_1\text{UnbJoin}$, which impacts the shuffling costs. Assuming that $\text{TG}_1\text{UnbJoin}$ is assigned to the 4th MR cycle $MR_{4}$ in the workflow, then the redundancy in map output impacts ($MR_{1}, MR_{2}, MR_{3}, MR_{4}, MR_{5}$).

**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 Partial $\beta$-Unnesting: 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 $\text{AnnTG}$, that concisely represents annotated triplegroups. Example annotated triplegroup $\text{AnnTG}_{\text{gene9}}(o1, o2, o3, o4, o5)$ in Figure 7 represents the subject triplegroup $f_{tg1}$ (Figure 3(a)) which is a valid match for the unbound-property star subpattern $S_{J1}$ in query $Q1$. Annotated $\text{TG} \text{AnnTG}_{\text{gene9}}(o1, o2, o3, o4, o5)$ 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 $MR_{4}$. 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, $\text{AnnTG}_{\text{gene9}}(o1)$ and $\text{AnnTG}_{\text{gene9}}(o2)$ may be assigned to the same Reducer, e.g., $\text{Reducer1}$. The redundancy in the map output of $MR_{4}$ 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 $\beta$-unnest, we can reduce the size of map output, and hence reduce the shuffling costs. We propose a partial $\beta$-unnest strategy that creates a set of triplegroups that each contain the bound-property component $tg_{bnd}$, and a subset of the unbound-property component $tg_{unbnd}$.

**Definition 3. (Partial $\beta$-unnest)** Given a set of triplegroups $TG$, an unbound-property star pattern $S_{tg1} = \{P_{bnd}, P_{unbnd}\}$, and a partition function $\phi_{m}$ that partitions the triples in $tg_{unbnd}$ into $m$ partitions, the partial $\beta$-unnest operator $\mu^{\beta}_{tn}$ produces a set of triplegroups such that:

$$\mu^{\beta}_{tn}(\phi_{m})(TG) := \{tg' = \{tg_{bnd} \cup \text{partition}_{i}\} \}$$

where

- $\forall tg \in TG$, the bound-property subset $tg_{bnd} \subseteq tg$ such that $tg_{bnd}\text{props}() = P_{bnd}$.

- A function $\phi_{m}$ assigns a triple $tj \in tg_{unbnd}$ to $\text{partition}_{i}$, i.e., $\phi_{m} : tj \rightarrow \text{partition}_{i}$, where $i \in \{1, 2, ..., m\}$.

The function $\phi$ partitions the triples in $tg_{unbnd}$ into $m$ buckets based on the value of the join key. Essentially, $\mu^{\beta}_{tn}$ produces a maximum of $m$ triplegroups for each triplegroup $tg \in TG$. For example, a partial $\beta$-unnest on $AnnTG_{\text{gene9}}(o1, o2, o3, o4, o5)$ in Figure 6 using the partition function $\phi_{2}$ produces 2 triplegroups - $AnnTG_{\text{gene9}}(o1, o2, o3, o4)$ and $AnnTG_{\text{gene9}}(o2, o4)$ respectively. This implies that $\phi_{2}(o1) = \phi_{2}(o3) = \phi_{2}(o5) = k1$. Similarly, $AnnTG_{\text{gene9}}(o2)$ and $AnnTG_{\text{gene9}}(o4)$ 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, $\text{TOptUnbJoin}$. Figure 6(c) represents how the I/O footprint can be reduced by partial and delayed $\beta$-unnesting at map phase of $MR_{1}$.

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 $Job_{2}$, 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

Job1: Compute 'matching' triplegroup equivalence classes
Map:
\[ TG\_GroupByMap(Tuples T) \]
Reduce:
\[ TG' \leftarrow TG\_GroupByReduce(TG, \text{List <Tuples>}); \]
\[ TG'' \leftarrow TG\_UngrpFilter(TG', EC, \{P\_ond, P\_unbn\}); \]
Job2: Join between triplegroup equivalence classes
Map:
\[ TG\_OptUnbJoin.Map(TG)'/\text{partial } \beta\text{-unnest} \]
\[ \text{or } TG\_UnbJoin.Map(TG'')'/\beta\text{-unnest} \]
Reduce:
\[ TG'' \leftarrow TG\_OptUnbJoin.Reduce(TG ''); \]
\[ \text{or } TG'' \leftarrow TG\_UnbJoin.Reduce(TG ''); \]

The 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 \( \text{tempMap} \) 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 \( P\_ond \) 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

\[ \text{\betaGrpFilter}(tg, ECList, \{P\_ond, P\_unbn\}); \]
\[ \text{tempMap} \leftarrow \text{extract tuples in } tg; \]
\[ \text{matchECList} \leftarrow \text{match(tempMap, ECList);} \]
\[ P\_ond \leftarrow \text{extract bound properties in } EC; \]
\[ \text{if } P\_ond \subseteq \text{tempMap} \text{.keySet then } \]
\[ \text{if } EC \text{ contains unbound property then } \]
\[ \text{propMap} \leftarrow \text{\beta group filtering } \text{propMap} \text{ = tempMap; } \]
\[ \text{else } \]
\[ \text{ Extract only bound properties in } EC \]
\[ \text{propMap} \leftarrow \text{extract } P\_ond \text{ entries from tempMap; } \]
\[ \text{emit } \text{(AnnTG(Sub, EC, propMap)); } \]

Job2: 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 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 \( (right\_Hash) \) 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, 4GB memory and running Red Hat Linux. 60 and 80-node Hadoop clusters (block size set to 256MB, 1GB heap-size for child jvm’s) 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
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 β-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 β-unnesting reduces the shuffle
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 comparison 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
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 $M_{R1}$ and reduced scan costs and shuffling costs in $M_{R2}$, resulting in additional 18% performance gain over EagerUnnest in $M_{R2}$. 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 “nur77”, 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 (selective) 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 approximate 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 β-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 β-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 β-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


