ABSTRACT

SPARQL basic graph pattern (BGP) (a.k.a. SQL inner-join) query optimization is a well researched area. However, optimization of OPTIONAL pattern queries (a.k.a. SQL left-outer-joins) poses additional challenges, due to the restrictions on the reordering of left-outer-joins. The occurrence of such queries tends to be as high as 50% of the total queries (e.g., DBPedia query logs).

In this paper, we present Left Bit Right (LBR), a technique for well-designed nested BGP and OPTIONAL pattern queries. Through LBR, we propose a novel method to represent such queries using a graph of supernodes, which is used to aggressively prune the RDF triples, with the help of compressed indexes. We also propose novel optimization strategies – first of a kind, to the best of our knowledge – that combine together the characteristics of acyclicity of queries, minimality, and nullification, best-match operators. In this paper, we focus on OPTIONAL patterns without UNIONs or FILTERs, but we also show how UNIONs and FILTERs can be handled with our technique using a query rewrite. Our evaluation on RDF graphs of up to and over one billion triples, on a commodity laptop with 8 GB memory, shows that LBR can process well-designed low-selectivity complex queries up to 11 times faster compared to the state-of-the-art RDF column-stores as Virtuoso and MonetDB, and for highly selective queries, LBR is at par with them.

Categories and Subject Descriptors

H.2.4 [Systems]: Query Processing

Keywords

Query optimization; SPARQL OPTIONAL patterns; Left-outer-joins; Semi-joins; Compressed bitvectors.

*Initial part of this work was completed when the author was at Rensselaer Polytechnic Institute and University of Pennsylvania.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SIGMOD'15, May 31–June 4, 2015, Melbourne, VIC, Australia.
Copyright 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2746483.

1. INTRODUCTION

Resource Description Framework (RDF) [8] is a widely accepted standard for representing semantically linked data on the web, and SPARQL [9] is a standard query language for it. RDF data is a directed edge-labeled multi-graph, where each unique edge (S P O) is called a triple – P is the label on the edge from the node S to node O.

SPARQL query language, like SQL, provides various syntactic constructs to form structured queries for RDF graphs. Two types of queries of our interest: are Basic Graph Pattern (BGP) and OPTIONAL pattern (OPT) queries. Any SPARQL BGP query can be methodically translated into an equivalent SQL inner-join query [20], and an OPT pattern query can be methodically translated into an SQL left-outer-join query [19]. Like SQL, the SPARQL grammar allows nested OPT queries, i.e., a query composed of an internmix of BGPs and OPT patterns. Since SPARQL BGP queries are same as SQL inner-joins (denoted by symbol ∞) and OPT queries are same as SQL left-outer-joins (denoted by ⊘), we will use these terms, acronyms, and symbols interchangeably in the rest of the text. BGP and OPT queries tend to be performance intensive, especially for very large RDF data. An extensive amount of research has gone in the optimization of SQL inner-join queries. The database and semantic web communities have taken these optimizations further along with the novel ideas of indexing and BGP query processing over RDF data [15,28,29,33,42,44,45].

Why OPT queries? RDF is a semi-structured data, and adherence of RDF “instance data” to its Ontology specification (a.k.a. schema), and in turn its completeness, is not always enforced especially for the data published on the web (e.g., DBPedia [2]), and that which is compiled from many diverse sources of RDF graphs (e.g., Linked Open Data[4]). This makes OPT queries a crucial tool for the end users. For example, consider an RDF network describing movie and TV sitcom/soap actors. Not all the actors may have their contact info, such as email and telephone numbers listed. Then an OPT query (Q1) as given below fetches all the actors with their respective name and address. Along with that, it gets email and tele numbers of those who have them listed, and for the rest it marks email and tele values by NULLs.

```
  ?actor :name ?name .
  ?actor :address ?addr .
  OPTIONAL {
    ?actor :email ?email .
    ?actor :telephone ?tele .}}
```

1http://linkeddata.org/
Query logs of SPARQL endpoints on the web indeed concur with this intuition, e.g., DBPedia query logs show as high as 50% occurrence of OPT queries, with as many as eight OPT patterns in a query. These statistics make OPT queries a non-negligible component of SPARQL “join” query optimization. Now consider a modified version (Q2) of the previous query:

```sql
SELECT ?friend ?sitcom
WHERE {
  :Jerry :hasFriend ?friend .
  OPTIONAL {
  }
}
```

This query asks for all friends of :Jerry that have acted in a sitcom located in the :NewYorkCity. In this query, let (John :hasFriend Mary) be tp1, (?friend :actedIn ?sitcom) tp2, and (?sitcom :location :NewYorkCity) tp3. Then the query can be expressed as (Query = tp1 \LeftJoin (tp2 \LeftJoin tp3)), where tp1 forms a BGP (say P1) with only one triple pattern, and (tp2 \LeftJoin tp3) forms another BGP (say P2).

BGP queries (ε) are associative and commutative, i.e., a change in the order of triple patterns and joins between them does not change the final results. But for a nested OPT query, \LeftJoin operator is not associative and commutative, e.g., in the case of Q2 above, left-outer-join between tp1 and tp2 cannot be performed before the inner-join P2 = (tp2 \LeftJoin tp3). This limits the number of query plans an optimizer can consider.

Let us assume that this actor network has thousands of actors, and many of them have acted in sitcoms located in the :NewYorkCity. But :Jerry has just two friends, :Larry and :Julia, and among them only :Julia has acted in :Seinfeld, which has :NewYorkCity as the location. So the final results of this query are just two with the respective values of ?friend and ?sitcom set as ( (:Larry, NULL), (:Julia, :Seinfeld) ). However, the inner-join P2 = (tp2 \LeftJoin tp3) has to be evaluated before the left-outer-join P1 \LeftJoin P2, due to the restrictions on the reorderability of left-outer-joins. Since there are several actors who have acted in the sitcoms located in :NewYorkCity, these two triple patterns have a low selectivity[^1], which increases the evaluation time and cost of their inner-join. To overcome such a limitation, for conventional databases, Ra et al. and Galindo-Legaria, Rosenthal[^2] have proposed ways of reordering nested inner and left-outer joins using additional operators as nullification and best-match[^3] (Generalized Outerjoin in [^2]).

On this background, we propose Left Bit Right (LBR) and make the following main contributions.

- We propose a novel way to represent a nested BGP-OPT query with a graph of supernodes (Section 2).
- We extend the previously known properties of acyclicity of queries, minimality of triples, and nullification, best-match operations, to propose novel optimization strategies—first of a kind, to the best of our knowledge—for acyclic and cyclic well-designed OPT queries (Sections 5.2, 5.3, 5.4). We mainly focus on the “join” component of SPARQL, i.e., OPT patterns without UNIONS, FILTERs, or Cartesian products. Nevertheless, in Section 5.2 we show how these constructs can be handled using our technique.
- Finally, we show LBR’s performance using a commodity laptop of 8 GB memory over three popular RDF datasets, Uniprot[10], LUBM[1], and DBpedia[2], with up to and over a billion triples, in comparison with the state-of-the-art RDF column-stores like Virtuoso v7.1.0 and MonetDB v11.17.21. Through our evaluation on queries with varying degrees of selectivity, complexity, and running times, we show that for complex, low selectivity “well-designed” queries, LBR is up to 11 times faster than Virtuoso and MonetDB, and for highly selective queries, it is at par with them (Section 6).

2. QUERY GRAPH OF SUPERNODES

An OPT query with an intermix of Basic Graph Patterns (BGPs ε) and OPT patterns (⊗) establishes restrictions on the order of join processing, e.g., Q1, Q2 in Section 1.

In this section we introduce a novel idea of constructing a graph of supernodes (GoSN) to capture the nesting of OPT patterns in a query. GoSN makes an important part of our query processing techniques discussed in Sections 3 and 4.

2.1 GoSN Construction

Supernodes: In a SPARQL OPT pattern of the form (P1 ε P2), P1 may in turn have nested BGPs and OPT patterns inside it, e.g., P1 = (P1 ε P2). P2 may have nested BGPs and OPT patterns inside it too, or either of P1 and P2 can be OPT-free. Generalizing it, if a pattern P does not have any OPT pattern nested inside it, we call P to be an OPT-free Basic Graph Pattern. From a given nested OPT query, first we extract all such OPT-free BGPs, and construct a supernode (SNi) for each P. The triple patterns (TPs) in P are encapsulated in SNi.

Next, we serialize a nested OPT query using its OPT-free BGPs, ε (inner-join), ⊗ (left-outer-join) operators, and proper parentheses. E.g., we serialize Q2 in Section 1 as (P1 ε P2), where P1 and P2 are OPT-free BGPs, SN1 of P1 encapsulates just tp1, and SN2 of P2 encapsulates tp2 and tp3 (see Figure 2.12).

Unidirectional edges: From the serialized query, we consider each OPT pattern of type Pm ε Pn. Pm or Pn may have nested OPT-free BGPs inside them. Using the serialized-parenthesized form of the query, we identify the leftmost OPT-free BGPs nested inside Pm and Pn each. E.g., if Pn = ((Pm ε Pn) ε (Pm ε Pn)) and Pm = (Pm ε (Pm ε Pn)), Pm and Pn are the leftmost OPT-free BGPs in Pm and Pn respectively, and SNm and SNn are their respective supernodes. We add a directed edge SNm \rightarrow SNn. If either Pm or Pn does not nest any OPT-free BGPs inside it, we treat the very pattern as the leftmost for adding a directed edge. With this procedure, we can treat OPT patterns in a query in any order. But for all practical purposes, we start from the innermost OPT patterns, and recursively go on considering the outer OPT patterns using the parentheses in the serialized query. E.g., if a serialized query is ((Pm ε Pn) ε ((Pm ε Pn) ε Pn)) ε Pn, with Pm...Pn as OPT-free BGPs, we add directed edges as follows: (1) SNm \rightarrow SNn, (2) SNm \rightarrow SNd, (3) SNn \rightarrow SNf, (4) SNd \rightarrow SNn.

Bidirectional edges: Next we consider each inner-join of type Pm ε Pn in a serialized query. If Pm or Pn has nested OPT-free BGPs inside it, we add a bidirectional edge between the supernodes of leftmost OPT-free BGPs. E.g., if Pm = (Pm ε Pn), and Pn = (Pm ε Pn), we add a bidirectional edge SNm \leftrightarrow SNn. If Pm or Pn does not nest any OPT-free BGPs inside it, we consider the very pattern to be the leftmost for adding a bidirectional edge. We add bidirectional edges starting from the innermost inner-joins (⊗)

[^1]: Selectivity of a triple pattern is high if it has fewer number of triples associated with it and vice versa.
[^2]: E.g., if a serialized query is ((Pm ε Pn) ε ((Pm ε Pn) ε Pn)) ε Pn, with Pm...Pn as OPT-free BGPs, we add directed edges as follows: (1) SNm \rightarrow SNn, (2) SNm \rightarrow SNd, (3) SNn \rightarrow SNf, (4) SNd \rightarrow SNn.
using the parentheses in the serialized query, and recursively go on considering the outer ones, until no more bidirectional edges can be added. Considering the same example given under unidirectional edges, we add a bidirectional edge between SN_a \leftrightarrow SN_c. The graph of supernodes (GoSN) for this example is shown in Figure 2.11.

Thus we completely capture the nesting of BGP and OPT patterns in a query using this GoSN, and establish an order among the supernodes, which is described in Section 2.2.

Figure 2.1: GoSN for – (a) Q2 in Section 1, (b) ((P_a \ni P_b) \ni (P_c \ni P_d)) \ni (P_e \ni P_f)

2.2 Nomenclature

In this section, we highlight the nomenclature that we use in the context of GoSN, OPT patterns, and RDF graphs.

Master-Slave: In an OPT pattern P_j \ni P_k, we call pattern P_j to be a master of P_k, and P_k a slave of P_j. This master-slave relationship is transitive, i.e., if a supernode SN_j is reachable from another supernode SN_k by following at least one unidirectional edge in GoSN, then SN_k is called a master of SN_j (see Figure 2.11).

Peers: We call two supernodes to be peers if they are connected to each other through a bidirectional edge, or they can be reached from each other by following only bidirectional edges in GoSN, e.g., SN_a and SN_c in Figure 2.11.

Absolute masters: Supernodes that are not reachable from any other supernode through a path involving at least one unidirectional edge are called the absolute masters, e.g., SN_a and SN_c in Figure 2.11. These master-slave, peer, and absolute master nomenclatures and relationships apply to any triple patterns enclosed within the respective supernodes too.

Well-designed patterns: As per the definition given by Pérez et al. [34], a well-designed OPT query is – for every sub-pattern of type P_j \ni P_k in the query, if a join variable “?j” in P_j appears outside P_k, then “?k” also appears in P_k. A query that violates this condition is said to be non-well-designed. In this paper, we have focused on well-designed queries to “prune” the triples for acyclic well-designed patterns using GoSN, and in Section 3.3 we discuss pruning for cyclic well-designed patterns. For the discussions in Sections 3.1, 3.2, 3.3 we do not take into consideration the underlying indexes on RDF graphs, because the optimization strategies are agnostic to them. Then in Section 4 we describe our indexes, and in Section 5 we show how to prune the triples using our indexes, and generate the final query results using multi-way-pipelined join for both, acyclic as well as cyclic well-designed queries.

3. OPTIMIZATION STRATEGIES

In Figure 3.1 we point out that LBR can process all the nested BGP–OPT queries, but not all of them can avoid nullification and best-match, and we mainly focus on the well-designed queries in this paper. Our technique can be applied to non-well-designed queries too, but we have not focused on them due to lack of evidence of such queries in practice, disparity between pure SPARQL and SPARQL-over-SQL engines for joins over NULLs (refer to Appendices B and C), and space limitations.

Figure 3.1: Classification of OPT queries

In Section 3.1 briefly ignoring GoSN, we describe the conceptual foundation of our technique by combining the properties of nullification, best-match, acyclicity of queries, and minimality of triples. Then in Section 3.2 we present our optimization strategies to “prune” the triples for acyclic well-designed patterns using GoSN, and in Section 3.3 we discuss pruning for cyclic well-designed patterns. For the discussions in Sections 3.1, 3.2, 3.3 we do not take into consideration the underlying indexes on RDF graphs, because the optimization strategies are agnostic to them. Then in Section 4 we describe our indexes, and in Section 5 we show how to prune the triples using our indexes, and generate the final query results using multi-way-pipelined join for both, acyclic as well as cyclic well-designed queries.

3.1 Preliminaries

Typically, for a pairwise join processing plan, if nested inner and left-outer joins are reordered, nullification and best-match (a.k.a. minimum-union) operations are required. We explain them here with a brief example for the completeness of the text, and refer the reader to [24][39] for the details.

Nullification, Best-match: Consider the same query given in Figure 2.1a along with the sample data associated with it in Figure 3.2. NewYorkCity has been the location for a lot of American sitcoms, and a lot of actors have acted in them (they are not shown in the sample data for conciseness). But, among all such actors, :Jerry has only two friends, :Julia and :Larry. Hence, tp_1 is more selective than tp_2 and tp_3. A left-outer-join reordering algorithm as proposed in [20][39] will typically reorder these joins as (tp_1 \ni tp_2) \ni tp_3. Due to this reordering, all four sitcoms that Julia has acted in show up as bindings of :sitcom (see Res1 in Fig. 3.2), although only :Seinfeld was located in the :NewYorkCity. To fix this, nullification operator is used, which ensures that variable bindings across the reordered joins are consistent with the original join order in the query (see Res2 in Figure 3.2).

We can see that the nullification operation caused results that are subsumed within other results. A result r_1

\footnote{www.w3.org/TR/REC-rdf-syntax/#section-Syntax-blank-nodes}
is said to be subsumed within another result \( r_2 \) \((r_1 \sqsubseteq r_2)\), if for every non-null variable binding in \( r_1 \), \( r_2 \) has the same binding, and \( r_2 \) has more non-null variable bindings than \( r_1 \). Thus results 3–5 in Res2 are subsumed within result 2. The best-match operator removes all the subsumed results (see Res3). Final results of the query are given as best-match(nullification)(\( \{ tp_1 \bowtie tp_2 \bowtie tp_3 \} \)).

Semi-join (\( \bowtie \)) is a well-known concept in databases, with which triples associated with a triple pattern (TP) can be removed due to restrictions on the variable bindings coming from another TP, without actually performing a join between the two. \( tp_2 \bowtie \pi_j t_p = \{ t | t \in tp_2, t.j \in (\pi_j (tp_1) \cap \pi_j (tp_2)) \} \). Here \( t \) is a triple matching \( tp_2 \), and \( t.j \) is a variable binding (value) of variable \( ?j \) in \( t \). After this semi-join, \( tp_2 \) is left with only triples whose \( ?j \) bindings are also in \( tp_1 \), and all other triples are removed. Now let triple patterns, \( tp_1, tp_2, ..., tp_n \), all share join variable \( ?j \). Then we define a clustered-semi-join over them as follows.

**Definition 3.1.** A clustered-semi-join(\( ?j \), \( \{ tp_1, tp_2, ..., tp_n \} \)) is performed as follows: Let \( \bar{J} = \{ \pi_j (tp_1) \cap ... \cap \pi_j (tp_n) \} \). For each triple pattern \( tp_i, 1 \leq i \leq n \), \( tp_i = \{ t | t \in tp, t.j \in \bar{J} \} \).

This definition shows that a clustered-semi-join is nothing but many semi-joins performed together for TPs that share a join variable.

**Example-1:** Now let us evaluate the query in Figure 3.2 using semi-joins and clustered-semi-joins. We do a semi-join \( tp_2 \bowtie \pi_j \text{friend} tp_1 \), because \( tp_1 \) does a left-outer-join with \( ?j \) over \('\text{friend}'\). That keeps only \('\text{Larry'}\) and \('\text{Julia'}\) bindings of \('_\text{friend}'\) in \( tp_2 \), and removes any other bindings, and in turn triples generating those bindings from \( tp_2 \) (they are not shown in the figure for conciseness). Followed by it, we do a clustered-semi-join of \('_\text{ sitcom}'\), \( \{ tp_2, tp_3 \} \). That removes \('_\text{Curb Your Enthusiasm'}', '_\text{Veep}' \), and \('_\text{New Adventures of Christine'}\) bindings of \('_\text{ sitcom}'\), and the respective triples from \( tp_2 \). Notice that this clustered-semi-join also removes the \('_\text{ Larry'}\) binding of \('_\text{ friend}'\) from \( tp_2 \) as a ripple effect of the removal of \('_\text{Curb Your Enthusiasm'}\) binding of \('_\text{ sitcom}'\). At the end, \( tp_1 \) and \( tp_2 \) have the same set of triples, but \( tp_2 \) now has only one triple (\('_\text{Julia'} :\text{actedIn} :\text{Seinfeld'}\) ). Now if we evaluate the original join \( tp_1 \bowtie (tp_2 \bowtie tp_3) \) or a reordered one \( (tp_1 \bowtie tp_2) \bowtie tp_3 \) on these reduced set of triples, we do not need nullification to ensure consistent bindings of \('_\text{ sitcom}'\), and there are no subsumed results, because each TP has a minimal set of triples.

Minimality of triples is defined as follows.

Let \( R \) be the final results of a query \( Q \), and \( tp \) be a TP in \( Q \). Let \( tp.s, tp.p, tp.o \) be the respective subject, predicate, and object positions in \( tp \). Let \( Q \) be such that it SELECTs all the variables as well as fixed positions in all the TPs in \( Q \). E.g., for the query in Figure 3.2 “SELECT :Jerry :hasFriend ?friend :actedIn :sitcom :location :NewYorkCity WHERE...” selects everything in the query, and the final results will be \{(:Jerry, :hasFriend, :Larry, :actedIn, NULL, :location, :NewYorkCity), (:Jerry, :hasFriend, :Julia, :actedIn, :Sefield, :location, :NewYorkCity)\}. Note that this assumption of SELECTion is just for the ease of definition of minimality, and not a required condition. Let \( \Delta_{tp} \) be the triples associated with a \( tp \) after a semi-join or clustered-semi-join. Then minimality of \( \Delta_{tp} \) is defined as follows.

**Definition 3.2.** Let \( R_{tp} = (\text{non-NUL}_{tp.s,tp.p,tp.o}(R)) \), i.e., \( R_{tp} \) is a projection of the respective distinct bindings of \( tp.s, tp.p, tp.o \) from \( R \) without any NULLs. Then \( \Delta_{tp} \) is said to be minimal, if \( \Delta_{tp} = R_{tp} \) (\( \Delta_{tp} \) and \( R_{tp} \) can be empty as well). In short, in a BGP-OPT query, the set of triples associated with a triple pattern is minimal, if every triple creates one or more variable bindings in the final results. There does not exist any triple which may get eliminated as a result of an inner or left-outer-join.

Next we see why minimality of triples is important in the context of an OPT pattern query.

**Lemma 3.1.** If every triple pattern in an OPT pattern has a minimal set of triples associated with it, nullification and best-match operations are not required if an original query \( tp_1 \bowtie (tp_2 \bowtie tp_3) \) is reordered as \( (tp_1 \bowtie tp_2) \bowtie tp_3 \).

The proof of Lemma 3.1 is given in Appendix A.1. Example-1 and Lemma 3.1 together interest us to find if the set of triples associated with each TP in an OPT pattern can be reduced to minimal through semi-joins and clustered-semi-joins alone, because then nullification and best-match can be avoided even if the joins are reordered.

**Acylicity:** Bernstein et al. [16, 17] and Ullman [19] have proved previously that if a “graph of tables” (GoT) of an inner-join query is a “tree” (i.e., it is acyclic), a bottom-up followed by a top-down pass with semi-joins at each table in this tree, reduces the set of triples in each table to a minimal. In the context of a SPARQL query, GoT is a graph of TPs, where each TP is treated as a unique table, and two TPs can be avoided even if the joins are reordered.

**Acyclicity:** Bernstein et al. [16, 17] and Ullman [19] have proved previously that if a “graph of tables” (GoT) of an inner-join query is a “tree” (i.e., it is acyclic), a bottom-up followed by a top-down pass with semi-joins at each table in this tree, reduces the set of triples in each table to a minimal. In the context of a SPARQL query, GoT is a graph of TPs, where each TP is treated as a unique table, and two TPs can be avoided even if the joins are reordered.

**Lemma 3.2.** For a join query, if the GoT is acyclic, then the GoJ is acyclic too.

Acylicity of GoJ follows from its construction. We have given the proof of Lemma 3.2 in Appendix A.2. From Lemma 3.2 we also observe the following property.

\[\text{Redundant cycles may occur if multiple TPs join over same variable.}\]
Property 3.1. For a BGP query without OPT patterns (inner-joins only), if the GoJ is acyclic (is a tree), a bottom-up followed by a top-down pass on the GoJ with “clustered-semi-joins” performed at each jvar-node, reduces the set of triples associated with each TP in the query to a minimal.

Note: We use the query and GoSN in Figure 3.1 and GoJ in Figure 3.2, together as a running example in the rest of the text in this paper without mentioning it explicitly every time.

### 3.2 Acyclic well-designed OPT patterns

In Section 3.1 through Example 1 and Lemma 3.1, we showed that if each TP in an OPT query has minimal triples, nullification and best-match are not required. We showed how minimality of triples can be achieved for an acyclic OPT-free BGP through Lemma 3.2 and Property 3.1. But can we combine all these observations to use the acyclicity of GoJ of a nested OPT query using its graph of supernodes (GoSN)? We explore it next.

Let the GoJ of a nested OPT pattern without Cartesian products be acyclic, i.e., the GoJ is connected and is a tree. Cyclic GoJ and Cartesian products are discussed separately in Sections 3.4 and 3.2 respectively. We choose the least selective jvar node that appears in an absolute master supernode, and fix it as the root of the GoJ tree. Jvar-nodes can be ranked for selectivity as follows: A jvar-node $?j_1$ is considered more selective than $?j_2$, if the most selective TP having $?j_1$ has fewer triples than the most selective TP having $?j_2$, and so on.

With the root fixed as given before, this GoJ tree has all the jvars in absolute masters towards the top of the tree (root and internal nodes), and jvars in the slaves towards the bottom. This observation follows from the facts that there are no Cartesian products, GoJ is a tree, and we chose the root from an absolute master. We can traverse this tree bottom-up with semi-joins and clustered-semi-joins performed at each jvar-node $?j$ as follows:

- We perform a clustered-semi-join among the TPs that contain $?j$, and which appear either in the same supernode, or their supernodes are peers of each other.
- We perform a semi-join between pairs of TPs that are in a master-slave relationship.

We can do a top-down pass following the same rules. E.g., we choose $?friend$ as the root of GoJ tree in our example, because it appears in absolute master $SN_1$. In the bottom-up pass, we perform a clustered-semi-join over $?sitcom$ among the peers $tp_2$ and $tp_3$, no semi-join for $?sitcom$, because there are no master-slave TPs with $?sitcom$. We perform a semi-join $tp_2 \times ?friend \times tp_1$, no clustered-semi-join for $?friend$ because there are no peer TPs with $?friend$. Next, in the top-down pass, we process $?friend$ first followed by $?sitcom$. This process leaves each TP with a minimal set of triples.

But does this give us an optimal order of processing jvars? No, because a bottom-up pass on the GoJ tree is same as processing OPT patterns per the order imposed in the original query – recall that all the jvars in slaves appear towards the bottom of the tree, and all the jvars in masters towards the top of the tree. So this hardly fetches us any benefits of the selectivity of the master TPs. We want to find an optimal order of processing jvars, which exploits the selectivity of the masters to aggressively prune the triples.

For that we use `get_jvar_order` (Alg. 3.1). From the GoJ tree of an OPT pattern, we consider an induced subtree consisting only of jvars that appear in the absolute master supernodes (ln 4). Since there are no Cartesian products, this induced subtree is connected. We choose a jvar with the least selectivity as the root of this induced subtree, do a bottom-up pass on it, and store the order in `order_{bu}` (ln 5). Choosing a jvar with least selectivity as the root ensures that it is processed last.

```
Algorithm 3.1: get_jvar_order

input: GoSN, GoJ
output: order_{bu}, order_{td}
1 if GoJ is cyclic then
2 \( order_{greedy} = greedy-jvar-order(GoSN, GoJ); \)
3 return order_{greedy}, order_{greedy}
4 \( J_m = \) jvars in absolute master supernodes;
5 root = $?j \in J_m$ with least selectivity;
6 \( T_{root} = \) get-tree($J_m$, root);
7 order_{bu} = bottom-up($T_{root}$);
8 \( SN_{ss} = \) order remaining slaves with masters first;
9 for each slave supernode $SN_i$ in $SN_{ss}$ do
10 \( \{ J_s = \) jvars in $SN_i$;
11 root = $?j \in masters(SN_i)$;
12 \( T_s = \) get-tree($J_s$, root);
13 order_{td} = \text{order}_{td}\ append\text{-top-down}(T_s);
14 order_{td} = \text{order}_{td}\ append\text{-bottom-up}(T_{root});
15 for each slave supernode $SN_i$ in $SN_{ss}$ do
16 \( \{ J_s = \) jvars in $SN_i$;
17 root = $?j \in masters(SN_i)$;
18 \( T_s = \) get-tree($J_s$, root);
19 order_{td} = \text{order}_{td}\ append\text{-top-down}(T_s);
20 return order_{bu}, order_{td};
```

We order the remaining supernodes as – masters before their respective slaves, and among any two peer supernodes, a supernode with a more selective triple pattern is ordered first. This order is $SN_{ss}$ (ln 5). Note that such an ordering of supernodes favours selective masters to be processed before their non-selective peers and slaves, thus benefiting
the pruning process. For each supernode \( S_N \) in \( SN_{as} \), we consider an induced subtree of GoJ consisting only of jvars in \( S_N \), and choose a root jvar such that it also appears in a master of \( S_N \) [11]. Note that since GoJ is a connected tree, a slave supernode shares at least one jvar with a master. We make a bottom-up pass on this induced subtree of \( S_N \), and append it to \( order_{bu} \) (in [14]). For a top-down pass, we reverse the above procedure. Starting with the induced subtree of absolute masters, we do a top-down pass, and store it in \( order_{td} \) (in [1]). Using the same order of supernodes, \( SN_{as} \), for an induced subtree of each \( S_N \) in \( SN_{as} \), we do a top-down pass, and append it to \( order_{td} \) (in [15][19]).

With \( order_{bu} \) and \( order_{td} \) of jvar-nodes, we prune the triples associated with TPs in a query using prune_triples (Alg. 3.2). For each jvar \( ?j \) in \( order_{bu} \), first we do a semi-join between TPs that are in a master-slave relationship (in [2]). Then we do a clustered-semi-join among TPs having \( ?j \), and which appear in the same supernode, or are peers of each other (in [5]). Recall that the master-slave or peer relationship among TPs is determined by GoSN. We repeat the same process, but now by following \( order_{td} \) of jvars (in [9][16]). Notice that in this process, semi-joins transfer the restrictions on variable bindings from master TPs to the slaves without actually performing left-outer-joins, and clustered-semi-joins transfer restrictions on variable bindings among all the peers without actually doing inner-joins. Through \( order_{bu} \), \( order_{td} \) of pruning we ensure that jvars in masters always get pruned before those in slaves.

Example-2: Recalling our running example, which has an acyclic GoJ, we get \( order_{bu} = [(?friend), (?city), (?friend)] \), and \( order_{td} = [(?friend), (?friend), (?city)] \) from get_jvar_order, and with these we use prune_triples to do semi-joins and clustered-semi-joins among the TPs in the master-slave and peer relationships respectively.

**Lemma 3.3.** For a well-designed OPT query with an acyclic GoJ, Algorithm 3.7 followed by Algorithm 3.2 leaves a minimal set of triples for each triple pattern. □

The proof of Lemma 3.3 is given in Appendix A.3

### 3.3 Cyclic well-designed OPT patterns

For OPT-free BGPs, i.e., pure inner-joins, with cyclic GoJ, minimality of triples cannot be guaranteed using clustered-semi-joins [16][17][43]. This result carries over immediately to cyclic OPT patterns too. For a cyclic OPT pattern, minimality of triples cannot be guaranteed, so we simply return \( order_{greedy} \) from get_jvar_order (in [5]), which is a greedy order of jvars, i.e., all the jvars are ranked in the descending order of their selectivity. Recall from Section 3.2 that the relative selectivity between two jvars can be determined from the selectivity of the TPs which have those jvars. In prune_triples, we use \( order_{greedy} \) in place of \( order_{bu} \) and \( order_{td} \), and follow the rest of the procedure as is. Since minimality of triples in each TP is not guaranteed, we need to use the nullification and best-match operations in a reordered query to ensure consistent variable bindings, and to remove any subsumed results.

This observation in general holds for all cyclic OPT queries, but we identify a subclass of cyclic OPT queries that can avoid nullification and best-match by just using \( order_{greedy} \) in place of \( order_{bu} \) and \( order_{td} \) in prune_triples — in such cyclic OPT queries, each slave supernode has only one jvar in it (slaves can have one or more non-join variables).
P-dimensional (O-S BitMats are nothing but transpose of the respective S-O BitMats), (2) P-O BitMats by slicing the S-dimension, and (3) P-S BitMats by slicing the O-dimension. Altogether we store $2 \times |V_o| + |V_s| + |V_e|$ BitMats for any RDF data. Figure 4.1 shows 2D S-O BitMats that we can get by slicing the predicate dimension (others are not shown for conciseness). Intuitively, a 2D S-O or O-S BitMat of predicate `hasFriend` represents all the triples matching a triple pattern of kind (`?a :hasFriend ?b`), a 2D P-S BitMat of O-value `:Seinfeld` represents all triples matching triple pattern (`?c :actedIn ?sitcom`), and so on.

Each row of these BitMats is compressed using run-length-encoding. A bit-row like “1100111110” is represented as “[1 3 2 4 1]”, and “0010010000” is represented as “[0] 1 2 1 4”. Notably, in the second case, the bit-row has only two set bits, but it has to use five integers in the compressed representation. So we use a hybrid representation in our implementation that works as follows – if the number of set bits in a bit-row are less than the number of integers used to represent it, then we simply store the set bit positions. So “0010010000” will be compressed as “3 6” (3 and 6 being the positions of the set bits). This hybrid compression fetches us as much as 40% reduction in the index space compared to using only run-length-encoding as done in [1].

Fold operation is represented as ‘fold(BitMat, RetainDimension) returns bitArray’. It takes a 2D BitMat and folds it by retaining the RetainDimension. More succinctly, a fold operation is nothing but projection of distinct values of the particular BitMat dimension, by doing a bitwise OR on the other dimension. It can be represented as:

$$\text{fold}(BM_{tp}, \text{dim}_j) \equiv \pi_{\text{dim}_j}(BM_{tp})$$

$BM_{tp}$ is a 2D BitMat holding the triples matching $tp$, and $\text{dim}_j$ is the dimension of BitMat that represents variable $j$ in $tp$. E.g., for a triple pattern `(?friend :actedIn ?sitcom)`, if we consider the O-S BitMat of predicate `:actedIn`, `?friend` values are in the “column” dimension, and `?sitcom` values are in the “row” dimension of the BitMat.

Unfold operation is represented as ‘unfold(BitMat, MaskBitArray, RetainDimension)’. For every bit set to 0 in the MaskBitArray, unfold clears all the bits corresponding to that position of the RetainDimension of the BitMat. Unfold can be simply represented as:

$$\text{unfold}(BM_{tp}, \beta_j, \text{dim}_j) \equiv \{ t | t \in BM_{tp}, j \in \beta_j \}$$

$t$ is a triple in $BM_{tp}$ that matches $tp$. $\beta_j$ is the MaskBitArray containing bindings of $j$ to be retained. $\text{dim}_j$ is the dimension of $BM_{tp}$ that represents $j$. If $t \in \beta_j$ in triple $t$. In short, unfold keeps only those triples whose respective bindings of $j$ are set to 1 in $\beta_j$, and removes all other.

Reasons for choice of BitMats: RDF stores that achieve high compression ratio using variable length or delta encoding, have to decode/decompress the variable length IDs, and get them in 4-byte integers for performing joins. This ends up being an overhead for queries that have to access a large amount of data. In BitMats, IDs and in turn triples are represented by bits compressed with 4-byte run-lengths, which are manipulated through the fold-unfold procedures without decompressing them. Hence we have chosen BitMats as the base index structure in our implementation.

5. QUERY PROCESSING

Up to this point, we saw how to construct a GoSN to capture the nesting of OPT patterns in a query in Section 2.1 In Sections 3.2 and 3.3 we presented our optimization strategies to “prune” the triples associated with TP in acyclic and cyclic well-designed queries, without considering the underlying index structure for RDF data. Then in Section 4 we described BitMat – our index structure.

Now, in this section, we “connect the dots”, i.e., we present how triples associated with TPs in a query are pruned using BitMats, how semi-join and clustered-semi-join in prune_triples (Alg 3.2) are achieved through BitMats and fold-unfold by consulting GoSN, and finally how output results are generated using a multi-way pipelined join for any acyclic or cyclic well-designed query. We put all these concepts together in Algorithm 5.1.

Algorithm 5.1: Query processing

input : Original BGP-OPT query
output : Final results
1 GoSN = get-graph-supernodes(Orig BGP-OPT query);
2 GoJ = get-graph-jvars(Orig BGP-OPT query);
3 for each tp_j in GoSN do
4 \{ tp_j = BM_{tp_j} = init();
5 bool NB-reqd = decide-best-match-reqd(GoSN, GoJ);
6 \{ order_u, order_id = get_jvar_order(GoSN, GoJ);
7 prune_triples(order_u, order_id, GoSN); // Alg 3.2
8 sorted-tps = sort TPs in master-slave hierarchy;
9 allres = multi-way-join(vmap, sorted-tps, visited, NB-reqd);
10 if NB-reqd then
11 \{ finalres = best-match(allres);
12 else
13 \{ finalres = allres;
14 \} return finalres;

Algorithm 5.2: semi-join

input : $t_j, tp_j, tp_i$
1 $\beta_j = \text{fold}(BM_{tp_j}, \text{dim}_j) \text{ AND } \text{fold}(BM_{tp_i}, \text{dim}_j)$;
2 $\text{unfold}(BM_{tp_j}, \beta_j, \text{dim}_j)$;

Algorithm 5.3: clustered-semi-join

input : $t_j, \{tp_1, ..., tp_k\}$
1 $\beta_{t_j} = \text{bitarray with all bits set to 1}$;
2 for each $tp_i$ in $\{tp_1, ..., tp_k\}$ do
3 $\beta_j = \beta_j \text{ AND } \text{fold}(BM_{tp_i}, \text{dim}_j)$;
4 for each $tp_i$ in $\{tp_1, ..., tp_k\}$ do
5 $\text{unfold}(BM_{tp_i}, \beta_j, \text{dim}_j)$;

In Algorithm 5.1 we first construct the GoSN and GoJ (in [4.2]). Then with init, we load a BitMat for each TP in the query that contains the triples matching that TP (in [4]). Recall from Section 4 that we have, in all, four types of BitMats. We choose an appropriate BitMat for each TP as follows. If the TP in the query is of type `(?var :fx1 :fx2)`, i.e., with two fixed positions, we load only one row corresponding to :fx1 from the P-S BitMat for :fx2. Similarly for a TP of type `(?var :fx1 :fx2 ?var)`, we load only one row corresponding to :fx2 from the P-O BitMat for :fx1. E.g., for `(?comment :location :NewYorkCity)` we load only one row corresponding to :location from the P-S BitMat of :NewYorkCity. If the TP is of type `(?var1 :fx1 ?var2)`, we load either the S-O or O-S BitMat of :fx1. If ?var1 is a join variable and ?var2 is not, we load the S-O BitMat and vice versa. If both,
?var1 and ?var2 are join variables, then we check which of ?var1 and ?var2 appears in ordervars before the other. If ?var1 comes before ?var2, we load the S-O BitMat and vice versa.

Recalling Example-2 from Section 3.2 for (?friend :actedIn \sitcom \& \friend) we load the S-O BitMat of :actedIn because ?friend comes before \sitcom in ordervars.

While loading the BitMats with init, we do active pruning using the TPs that may have been initialized previously. E.g., if we first load BitMat BM_{\beta_1} containing triples matching (:Jerry :hasFriend ?friend), then while loading BM_{\beta_2}, we use the bindings of ?friend in BM_{\beta_1} to actively prune the triples in BM_{\beta_2} while loading it. Then while loading BM_{\beta_2}, we use the bindings of \sitcom in BM_{\beta_2} to actively prune the triples in BM_{\beta_1}, where \sitcom and ?var1 are join variables, then we check which of \sitcom and ?var1 comes before ?var2, we load the S-O BitMat of \sitcom because \sitcom comes before ?var2.

Next we decide if nullification and best-match are required – they are required for a cyclic query where slaves have more than one jvars (ln 5). Then using get_jvar_order (ln 6) we get an optimal order of jvars for the bottom-up and top-down passes on GoJ. Recall that for a cyclic GoJ, get_jvar_order simply returns (order_greedy, order_greedy).

Next, we prune the triples in BitMats using prune_triples (ln 7 in Alg 5.1). This procedure is already been explained in Section 5.2 hence we refer the reader to it. Two important operations in prune_triples are semi-join and clustered-semi-join – to remove the triples from the BitMats. These operations make use of the fold and unfold primitives. We have shown how fold and unfold are used in semi-join and clustered-semi-join with Algorithms 5.2 and 5.3 respectively.

Recall from prune_triples (Alg 5.2) that we do a semi-join (Alg 5.2) between two TPs, if they are in a master-slave relationship over a shared jvar. The slave TP takes the restrictions on the variable bindings across all the slave TPs, and store the intersection result in \beta_j. Then we use \beta_j as the MaskBitArray in unfold to remove any triples whose respective bindings for ?j were dropped as a result of the intersection. Clustered-semi-join (Alg 5.3) is same as semi-join, except that we transfer the restrictions on the variable bindings across all the TPs that share a join variable and are peers of each other. Recall Example-2 from Section 3.2 which shows how semi-join and clustered-semi-join are used.

If an OPT query is acyclic, after prune_triples, each TP BitMat has a minimal set of triples (Lemma 5.3). In case of a cyclic OPT query, prune_triples only reduces the triples in the BitMats, but they may not be minimal. Note that using prune_triples, we prune the triples in BitMats, but we need to actually “join” them to produce the final results. For that we use multi-way-join (ln 8 in Alg 6.1). This procedure is described separately in Section 5.4. After multi-way-join, we use best-match to remove any subsumed results only if the query is cyclic and its slaves have more than one jvars (ln 11) – recall Lemmas 3.3 and 3.4. In best-match, we externally sort all the results generated by multi-way-join, and then remove the subsumed results with a single pass over them.

In the init and prune_triples processes, we do a “simple optimization” – if at any point, a TP in an absolute master supernode has zero triples, we take that as a hint of an empty result, and abandon any further query processing.

Currently Algorithm 5.1 is a main-memory process, i.e., all the TP BitMats are kept in memory during the query processing, and there is no disk spooling. This may pose some limitations on the total size of BitMats in a query. But as seen in our evaluation, LBR could handle low-selectivity queries with up to 13 TPs on a dataset with more than a billion triples on a machine with 8 GB memory, exhibiting the scalability of our technique. Presently LBR does not handle TPs with all variable positions (?a ?b ?c), and supporting them is currently under development.

5.1 Multi-way Pipelined Join

Before calling multi-way-join, we first sort all the TPs in the query as follows. Considering the TPs in absolute master supernodes, we sort them in an ascending order of the number of triples left in each TP’s BitMat. Then we sort remaining TPs in the descending order of master-slave hierarchy and selectivity. That is, among two supernodes connected as SN_1 \rightarrow SN_2, TPs in SN_1 and any peers of SN_1 are sorted before those in SN_2. Among the peer TPs, they are sorted in the ascending order of the number of triples left in their BitMats (ln 9 in Alg 5.1). This order is stps. In multi-way-join we use at most \sum_{i\in \text{stps}}\upsilon_{\text{vars}(tp_i)} additional memory buffer, where \upsilon_{\text{vars}(tp_i)} are the variables in every tp_i in the query Q. This is vmap in Alg 5.1. Thus we use negligible additional memory in multi-way-join.

At the beginning, multi-way-join gets an empty vmap for storing the variable bindings, stps, an empty visited list, and a flag nullreq indicating if nullification is required (depending on the cyclicity of the query). In multi-way-join, we go over each triple in BM_{\beta_1} of the first TP in stps, generate bindings for the variables in tp_1, and store them in vmap. We add tp_1 to the visited list, and call multi-way-join recursively for the rest of the TPs (ln 10). Note that in each recursive call, multi-way-join gets a partially populated vmap and a visited list that tells which TP’s variable bindings are already stored in vmap. Then we check if any variables in a non-visited tp are already mapped in vmap (ln 11). Recall that since the query does not have Cartesian products, we always find at least one tp which has one or more of its variables mapped in vmap. Also notice that stps order ensures that a master TP’s variable bindings are stored in vmap before its slaves. If there exist one or more triples t in BM_{\beta_1} consistent with the variable bindings in vmap, then for each such t we generate bindings for all the variables in tp_1, store them in vmap, and proceed with the recursive call to multi-way-join for the rest of the TPs (ln 12). Notice that, this way we pipeline all the BitMats, and do not do pairwise joins or use any other intermediate storage like hash-tables.

If we do not find any triple in BM_{\beta_1} consistent with the existing variable bindings in vmap, then – (1) if tp_1 is an absolute master, we rollback from this point, because an absolute master TP cannot have NULL bindings (ln 13), else (2) we map all the variables in tp_1 to NULLs, and proceed with the recursive call to multi-way-join (ln 14). When all the TPs in the query are in the visited list, we check if we require nullification to ensure consistent variable bindings in vmap across all the slave TPs, and output one result (ln 15). We continue this recursive procedure till triples in BM_{\beta_1} are exhausted (ln 16).
Algorithm 5.4: multi-way-join

```
input : vmap, stps, visited, nulreqd
output: all the results of the query

if visited.size == stps.size then
  nullification (vmap);
  output (vmap); // generate a single result
  return;

if visited is empty then
  tp1 = first TP from stps;
  visited.add(tp1);
  for each triple t ∈ BMtp1 do
    generate bindings for vars(tp1) from t, store in vmap;
    multi-way-join(vmap, stps, visited, nulreqd);
  then
    ttp = an absolute master then
      return;
      // This means tp1 is a slave
    set all vars(tp1) to NULL in vmap;
    visited.add(tp1);
    multi-way-join(vmap, stps, visited, nulreqd);
  end if
  continue;

if atleast-one-triple = false then
  for each triple t ∈ BMtp with same bindings do
    store vars(tp) bindings from t in vmap;
    visited.add(tp);
    multi-way-join(vmap, stps, visited, nulreqd);
    visited.remove(tp);
  end for
  if (atleast-one-triple = false) then
    if tp1 is an absolute master then
      return;
      // This means tp1 is a slave
    set all vars(tp1) to NULL in vmap;
    visited.add(tp1);
    multi-way-join(vmap, stps, visited, nulreqd);
  end if
end if
```

Intuitively, `multi-way-join` is reminiscent of a relational join plan with reordered left-outer-joins — that is, in `stps` we sort selective masters before their non-selective peers and slaves, and masters generate variable bindings before slaves in `vmap`. But note that we pipeline all the joins together, and we can skip nullification and best-match for acyclic queries and cyclic queries with only one year per slave, because of our optimization techniques, get_ıvar_order (Alg 5.3), prune_triples (Alg 5.2, and Lemmas 5.3 and 5.4).

Recall Example-2 from Section 5.2. After `prune_triples`, `tp1` has two triples, and `tp2`, `tp3` have one triple each in their BitMats. We sort the TPs as `stps = [tp1, tp2, tp3]`. In `multi-way-join`, we first generate a binding of `tp1.¿friend` and store it in `vmap`. In the recursive calls, we locate triples with the same `¿friend` binding in `BMtp2`. For each such triple, we generate `tp2.¿friend`, `tp2.¿friend` bindings in `vmap`, and proceed to `tp3`. In a recursive call, if we do not find any triple with the same variable bindings in `tp2` or `tp3`, we set variables in that TP to null in `vmap`. Since this is an acyclic query, we do not need `nullification`. While `outputting` a result, we pick variable bindings generated by masters over their slaves for common variables in `vmap`, e.g., we pick binding of `¿friend` from `tp1.¿friend` over `tp2.¿friend`.

### 5.2 Discussion

For our experiments, we assume that all the variables in a query — join as well as non-join — are SELECTed for projection, because analysis of DBPedia query logs shows that over 95% of the queries SELECT all the variables. As per W3C specifications, SPARQL algebra follows “bag semantics” [9], so SELECTion (projection) of particular variables from a query can be readily supported in LBR by just intercepting the output (vmap) statement at line 1 in `multi-way-join`, which will output only mappings of the SELECTed variables from `vmap`.

For the scope of this paper, we have focused on the “join” component of SPARQL, i.e., BGP-OPT patterns without UNIONS, FILTERs, or Cartesian products. But for the completeness of the text, here we discuss how our technique can be extended to handle them.

**UNION:** For this discussion, we assume well-designed UNIONS (UWD), which are — for every subpattern `P_r = (P_1 ∪ P_2)` in a query, if a variable `¿j` in `P_r` appears outside `P_r`, then it appears in both `P_1` and `P_2`. UWDs tend to have high occurrence (e.g., 99.97% as shown in [35]). Also UWDs remain unaffected by the difference between SPARQL and SQL over the treatment of NULLs [36], and following equivalences hold on them [34]: (1) `(P_1 ∪ P_2) → P_3 ∋ (P_1 ∪ P_3) ∪ (P_2 ∪ P_3)`, (2) `(P_1 ∪ P_2) ≡ (P_1 ∩ P_2) ∪ (P_2 ∩ P_1)`. Additionally, we also rewrite - (3) `P_1 ∪ (P_2 ∪ P_3)` as `(P_1 ∪ P_2) ∪ (P_1 ∩ P_3)` and remove any spurious results in the end.

Using these rules, we rewrite a well-designed BGP-OPT-UNION query in the UNION normal form (UNF) [54], i.e., a query of the form `P_1 ∪ P_2 ∪ ... ∪ P_n`, where each sub-pattern `P_i` (1 ≤ i ≤ n) is UNION-free. We evaluate each UNION-free `P_i` using the same LBR technique, remove any spurious results if rule (3) of rewrite is used, and add the results from all the `P_i`s. Note that we can add the output of all the `P_i`s without a conventional set-union, because SPARQL UNION follows “bag semantics” [8].

**FILTER:** For this discussion, we assume safe FILTERs [54, 55], i.e., for `P_r.F(R)`, where `R` is a filter to be applied on `P_r`, all the variables in `R` appear in `P_r`, `vars(R) ⊆ vars(P_r)`. Unsafe filters can alter the semantics of OPT patterns (ref [34]). In addition to the previous rewrite rules introduced inside UNION, for a well-designed BGP-OPT-UNION query with safe filters, following equivalences hold [34, 51]: (4) `(P_1 ∪ P_2).F(R) ≡ (P_1.F(R)) ∪ (P_2.F(R))", (5) `(P_1 ∪ P_2).F(R) ∪ (P_2.F(R))". Using these five rewrite rules, we push in filters, push out unions, and bring the query in the UNF, i.e., `P_1 ∪ P_2 ∪ ... ∪ P_n` where each sub-pattern `P_i` (1 ≤ i ≤ n) is union-free.

Now we can evaluate each `P_i`, using an augmented LBR technique as follows. We apply filters either by intercepting `init` (in 4) and `prune_triples` (in 5) or we apply them using a new filter-and-nullification (FaN) routine, in place of nullification (in 6 in `multi-way-join`). The decision depends on the type of filters, e.g., if a filter has variables from two or more TPs, then we apply it in FaN when variable bindings from all the TPs are generated in `vmap`. Through FaN, we apply filters as well as do nullification for a cyclic `P_i`. We apply `best-match` in the end if FaN nullifies at least one variable binding in `vmap` or `P_i` is cyclic. We remove any spurious results introduced due to rule (3) of UNION rewrite, and obtain the final results by adding the results from each `P_i` in the UNF — please recall our note about SPARQL “bag semantics” under UNION. We can use some “cheap” filter optimizations before rewriting a query, e.g., a filter `P_r.F(¿m = ¿n)` can be eliminated by replacing

---

6 Spurious results may get introduced if either `P_2` or `P_3` is empty, or `P_2` or `P_3` does not have any matching variable bindings to that of `P_1`. 

---
every ?n by ?m in all the TPs in P1. Extending LBR technique for OPT patterns with UNIONS and FILTERs is a part of our ongoing work.

Cartesian products (×): If a query is in the Cartesian normal form, i.e., P1 × P2 × . . . × Pi, where each sub-pattern Pi (1 ≤ i ≤ n) is ×-free, then we can evaluate each ×-free Pi using the same LBR technique presented in this paper, and generate the final results by taking a cross product of the results from each Pi sub-pattern. However, a query with an arbitrary nesting of ∃∀ and × poses challenges, because ∃∀ is not distributive over ×. A naïve way of handling such a query can be – use the technique presented in this paper only for the sub-patterns that are ×-free, and then resort to the standard relational technique of performing pairwise joins or ×-products to get the final results. As a part of the future work, we plan to extend LBR’s technique to handle such nested OPT-Cartesian queries using an augmented GoSN construction and modified multi-way-join (Alg 6.1). Note that queries with Cartesian products are rare for RDF data. Previously published SPARQL queries in the literature do not have Cartesian products [15, 28, 32, 33, 45, 46].

6. EVALUATION

We developed Left Bit Right (LBR) query processing technique with C/C++ language, compiled with g++ v4.8.2, -O3 -m64 flags, on a 64 bit Linux 3.13.0-34-generic SMP kernel (Ubuntu 14.04 LTS distribution). For running the experiments, we used a Lenovo T540p laptop with Intel Core i3-4000M 2.40GHz CPU, 8 GB memory, 12 GB swap space, and 1 TB Western Digital 5400RPM SATA hard disk.

6.1 Setup for Experiments

RDF Stores: To evaluate the competitive performance of LBR’s techniques, we used two columnstores Virtuoso v.7.1.0 [12] and MonetDB v11.17.21 [7], which are popular for RDF data storage. LBR’s core query processing algorithm works with the integer IDs assigned to the subjects, predicates, objects. Hence we loaded RDF triples with integer valued subjects, predicates, objects in Virtuoso and MonetDB’s native columnstore, and translated all the OPT pattern queries to their equivalent SQL counterpart. We setup all the required indexes, configuration, and “vectorized query execution” in Virtuoso as outlined in [13, 22]. For MonetDB, we created separate predicate tables with triples of only that predicate, ordered on S-O columns as outlined in [22], and also created an O-S index on each of these tables. However, due to a large number of predicates in the DBPedia dataset (see Table 6.1), MonetDB failed to create a separate table for each predicate. Hence we loaded the entire DBPedia data in one 3-column table, and created four indexes PSO, POS, SPO, OPS on that table – same as what LBR and Virtuoso use. Thus we ensured that both Virtuoso and MonetDB have an optimized setup. Current published sources of RDF-3X [33] and TripleBit [45] cannot process OPT queries. GH-RDF3X1, a third party system built using sources of RDF-3X for OPT pattern support, could not correctly process many OPT queries.

Datasets and Queries: We used three popular RDF datasets, (1) Lehigh University Benchmark (LUMB) – a synthetically generated dataset over 10,000 universities using their data generator program, (2) UniProt [10] – a real life protein network, and (3) DBPedia (English) 2014 [2] – a real life RDF dataset created from Wikipedia. The characteristics of these datasets are given in Table 6.1.

#### Table 6.1: Dataset characteristics

<table>
<thead>
<tr>
<th>Datasets</th>
<th>#triples</th>
<th># S</th>
<th># P</th>
<th># O</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUMB</td>
<td>1,335,081,176</td>
<td>217,206,845</td>
<td>18</td>
<td>161,413,042</td>
</tr>
<tr>
<td>UniProt</td>
<td>845,074,885</td>
<td>147,524,984</td>
<td>95</td>
<td>128,321,926</td>
</tr>
<tr>
<td>DBPedia</td>
<td>565,523,796</td>
<td>29,747,387</td>
<td>57,453</td>
<td>153,561,757</td>
</tr>
</tbody>
</table>

For a competitive evaluation of the three systems, we used a mix of well-designed OPT queries with varying degrees of selectivity, complexity, and running times. The LUBM data generator is popular for scalability analysis [13, 28, 44, 45]. However, its queries do not cover OPT patterns well. Therefore, to get a good coverage, we used UniProt queries from [5, 11], and LUBM queries from [6, 15], and introduced OPTIONAL patterns in them by studying the Ontology and graph structure. Similarly, we used DBPedia queries from [32] by removing union and filter clauses wherever necessary, because the main focus of our paper is on the nested BGP-OPT queries. Note that our approach is similar to that of other research systems, which focus on specific SPARQL components [15, 28, 33, 45, 46]. The current RDF benchmarks suffer from limitations as highlighted in [21]. Consequently, several systems use generated queries for specific SPARQL constructs. We evaluated all three systems over the same OPT queries, and they are given in Appendix A.

Evaluation Metrics: We used the following metrics for evaluation: (1) Time required for the init process in Alg 5.1 (Tinit), i.e., the time to load the BitMats associated with all the triple patterns in a query. (2) Time required for prune_triples (Alg 3.2) (Tprune). (3) Total query execution time (warm cache) for each system averaged over 5 runs, i.e., end-to-end clock-time (Ttotal, Tvert, Tmonet for LBR, Virtuoso, and MonetDB respectively). We ran each query 6 times by discarding the first runtime to warm up the caches. Also, for reporting this time fairly, we redicted output results of all the three systems to /dev/null, to eliminate any overhead of I/O latency in writing the results to the disk. Thus this reported time is the core query processing time spent by each system. Ttotal of LBR is Tinit + Tprune + Tmultiway. Tmultiway (Alg 5.3) is not explicitly shown in the evaluation for conciseness, as it can be computed by simply subtracting Tinit + Tprune from Ttotal. (4) Initial number of triples – the sum of triples matching each triple pattern in the query before the init and prune_triples procedures. (5) Sum of triples left in all the BitMats after prune_triples. (6) Number of final results. (7) Number of results, which have one or more NULL valued bindings, i.e., their result rows do not have all the variables bound due to the left-outer-joins in the query. These latter four metrics help us highlight the selectivity properties of the queries. (8) Whether nullification and best-match operations were required for LBR (see Section 5). Query processing times over all the three datasets are given in Tables 7.2, 8.3, and 9.4.

Note that the LBR system does not run in “server mode” as Virtuoso and MonetDB do, and it does not use any sophisticated cache management techniques of its own. It may only benefit from the underlying operating system kernel’s file-system cache management.

1https://github.com/gh-rdf3x/gh-rdf3x
3.4). For these three queries LBR gives several fold better performance than Virtuoso and MonetDB.

### 6.2 Analysis of the Evaluation

From Tables 6.2, 6.3, and 6.4, we can see that for queries with low selectivity, LBR’s technique clearly excels over Virtuoso and MonetDB.

Queries Q1–Q3 of LUBM need to access more than 50% of the data (see “#initial triples” column for the respective queries). These queries have a large number of triple patterns, have multiple OPT patterns, and generate a large number of results. Notably, these three queries have cyclic graphs of join-variables (GoJ), but only one join variable in each slave supernode. So LBR does not need to use nullification and best-match operations (see Section 3.2 and Lemma 3.4). For these three queries LBR gives several fold better performance than Virtuoso and MonetDB.

Queries Q4–Q6 of LUBM are short running, simpler queries with one OPT pattern. A closer look at these queries shows us that they have one or two highly selective triple patterns, with fixed values in predicate and object positions, and joins on S-S positions. This helps Virtuoso by way of fast merge-joins. Notice the column “#triples aft pruning”, which shows that these queries actually deal with a very small fraction of the entire data – about a million triples out of more than a billion triple dataset, and they produce very small number of results. Notably, for these queries, while Virtuoso performs much better, MonetDB suffers badly for Q1–Q4 because this master TP is not very selective and has only predicate position fixed. LBR’s pruning method with semi-joins shows a benefit here. For Q2, LBR’s init procedure with active pruning detects empty results of the query much earlier, and abandons further query processing (recall our “simple optimization” from Section 3). Whereas, Virtuoso and MonetDB detect empty results much later in their query processing. All seven UniProt queries are acyclic.

6.2 Analysis of the Evaluation

From Tables 6.2, 6.3, and 6.4, we can see that for queries with low selectivity, LBR’s technique clearly excels over Virtuoso and MonetDB.

Queries Q1–Q3 of LUBM need to access more than 50% of the data (see “#initial triples” column for the respective queries). These queries have a large number of triple patterns, have multiple OPT patterns, and generate a large number of results. Notably, these three queries have cyclic graphs of join-variables (GoJ), but only one join variable in each slave supernode. So LBR does not need to use nullification and best-match operations (see Section 3.2 and Lemma 3.4). For these three queries LBR gives several fold better performance than Virtuoso and MonetDB.

Queries Q4–Q6 of LUBM are short running, simpler queries with one OPT pattern. A closer look at these queries shows us that they have one or two highly selective triple patterns, with fixed values in predicate and object positions, and joins on S-S positions. This helps Virtuoso by way of fast merge-joins. Notice the column “#triples aft pruning”, which shows that these queries actually deal with a very small fraction of the entire data – about a million triples out of more than a billion triple dataset, and they produce very small number of results. Notably, for these queries, while Virtuoso performs much better, MonetDB suffers badly for Q1–Q4, because this master TP is not very selective and has only predicate position fixed. LBR’s pruning method with semi-joins shows a benefit here. For Q2, LBR’s init procedure with active pruning detects empty results of the query much earlier, and abandons further query processing (recall our “simple optimization” from Section 3). Whereas, Virtuoso and MonetDB detect empty results much later in their query processing. All seven UniProt queries are acyclic.

Similar to our previous observations, in case of DBPedia queries, for a query like Q1, which needs to handle larger amount of data, and produces more results, LBR comes out as a clear winner. Q2–Q6 access a much smaller fraction of the data, and generate a small set of results. In case of Q2–Q3, LBR’s initialization with active pruning detects an empty set of results early on and abandons further query processing. MonetDB suffers badly for Q1–Q4. We conjecture that this is because it was unable to create 57,453 distinct predicate tables. We successfully created four types of indexes on MonetDB’s DBPedia table, but its documentation revealed to us that MonetDB does not always honor user created indexes; it relies on its own policies of indexing and managing the data. All six DBPedia queries are acyclic.

Among UniProt queries, for Q1–Q4, LBR performs much better. These queries have multiple nestings of BGP-OPT patterns, and do not have highly selective triple patterns in them. Notably, for Q4 a semi-join between master and slave removes all the bindings in the slave. Virtuoso however could not exploit this situation (as it did for LUBM Q4–Q6), because this master TP is not very selective and has only predicate position fixed. LBR’s pruning method with semi-joins shows a benefit here. For Q2, LBR’s init procedure with active pruning detects empty results of the query much earlier, and abandons further query processing (recall our “simple optimization” from Section 3). Whereas, Virtuoso and MonetDB detect empty results much later in their query processing. All seven UniProt queries are acyclic.

### 6.2 Analysis of the Evaluation

From Tables 6.2, 6.3, and 6.4, we can see that for queries with low selectivity, LBR’s technique clearly excels over Virtuoso and MonetDB.

Queries Q1–Q3 of LUBM need to access more than 50% of the data (see “#initial triples” column for the respective queries). These queries have a large number of triple patterns, have multiple OPT patterns, and generate a large number of results. Notably, these three queries have cyclic graphs of join-variables (GoJ), but only one join variable in each slave supernode. So LBR does not need to use nullification and best-match operations (see Section 3.2 and Lemma 3.4). For these three queries LBR gives several fold better performance than Virtuoso and MonetDB.

Queries Q4–Q6 of LUBM are short running, simpler queries with one OPT pattern. A closer look at these queries shows us that they have one or two highly selective triple patterns, with fixed values in predicate and object positions, and joins on S-S positions. This helps Virtuoso by way of fast merge-joins. Notice the column “#triples aft pruning”, which shows that these queries actually deal with a very small fraction of the entire data – about a million triples out of more than a billion triple dataset, and they produce very small number of results. Notably, for these queries, while Virtuoso performs much better, MonetDB suffers badly for Q1–Q4, because this master TP is not very selective and has only predicate position fixed. LBR’s pruning method with semi-joins shows a benefit here. For Q2, LBR’s init procedure with active pruning detects empty results of the query much earlier, and abandons further query processing (recall our “simple optimization” from Section 3). Whereas, Virtuoso and MonetDB detect empty results much later in their query processing. All seven UniProt queries are acyclic.

Similar to our previous observations, in case of DBPedia queries, for a query like Q1, which needs to handle larger amount of data, and produces more results, LBR comes out as a clear winner. Q2–Q6 access a much smaller fraction of the data, and generate a small set of results. In case of Q2–Q3, LBR’s initialization with active pruning detects an empty set of results early on and abandons further query processing. MonetDB suffers badly for Q1–Q4. We conjecture that this is because it was unable to create 57,453 distinct predicate tables. We successfully created four types of indexes on MonetDB’s DBPedia table, but its documentation revealed to us that MonetDB does not always honor user created indexes; it relies on its own policies of indexing and managing the data. All six DBPedia queries are acyclic.

In general, observe that for low selectivity queries, Q1–Q3 (LUBM), Q1–Q4 (UniProt), and Q1 (DBPedia), LBR prunes a significant portion of the initial triples (see columns “#initial triples” and “#triples aft pruning”), in a relatively small amount of time ($T_{prune}$ is a very small portion of $T_{total}$). We believe that our novel technique of jvar-tree
traversal (see Section 3.2) along with the compressed index structure and fold-unfold operations (see Section 5) make the pruning procedure efficient. Also please notice that LBR processes long running queries several seconds or minutes faster than Virtuoso and MonetDB ($T_{Virt} - T_{total}$). Virtuoso does better for short running queries, but the difference ($T_{total} - T_{Virt}$) is just about a second. We feel that for such queries, in the future, LBR can probably use a more sophisticated cache management technique by running in server mode. We trust that for long running queries, the essence of LBR’s optimization technique gets highlighted.

To summarize, our evaluation shows that LBR works much better than the contemporary columnstores for queries with lower selectivity and higher complexity, e.g., queries Q1–Q3 (LUBM), Q1-Q4 (UniProt), and Q1 (DBPedia). The geometric mean of the presented queries for each dataset was as follows – for UniProt, 3.05 sec (LBR), 5.61 sec (Virtuoso), 4.35 (MonetDB), for LUBM, 10.18 sec (LBR) and 2.04 sec (Virtuoso), and for DBPedia 0.28 sec (LBR) and 0.07 sec (Virtuoso). The geometric mean of Virtuoso is lower than LBR for LUBM and DBPedia due to short running queries like Q4–Q6 (LUBM) and Q2–Q6 (DBPedia). We did not compute the geometric mean of MonetDB on LUBM and DBPedia as it took very long time for some of the queries.

**Index Sizes:** The on disk size of LBR indexes, i.e., $|V_1| + |V_2|$ BitMats, with our enhancements of “hybrid compression” technique is 20 GB, 32 GB, and 41 GB for DBPedia, UniProt, and LUBM datasets respectively. But note that we do not need to load all these indexes in memory. For any given query, we just load the BitMats associated with the triple patterns in the query, which are typically a smaller fraction of the total indexes. We ran our experiments on a machine with 8 GB memory, which could fit the BitMats associated with the triple patterns in the queries. For Virtuoso, the on disk size of the stored data and indexes was 5.5 GB, 9.5 GB, and 11 GB for DBPedia, UniProt, and LUBM datasets respectively, and for MonetDB it was 32 GB, 23 GB, 35 GB for DBPedia, UniProt, and LUBM respectively.

**7. RELATED WORK**

While the SPARQL OPT-free basic graph pattern (BGP) queries and their optimization have enjoyed quite bit of attention from the research community in the past few years, the discussion about OPTIONAL pattern queries has mostly remained theoretical. Previous work, such as [14, 22, 34, 41], has extensively analyzed the semantics of well-designed OPT patterns, as they have high occurrence [27, 35], bear desirable properties for the complexity analysis of the evaluation, and remain unaffected by the disparity between SPARQL and SQL algebra about joins over NULLs. Our query graph of supernodes (GoSN) is reminiscent of well-designed pattern trees (WDPT) [36, 37], but WDPTs are unidirected and unordered, whereas GoSN is directed, and establishes an order among the patterns (master-slave, peers), which is an integral part of our optimization technique. While the work on WDPTs focuses on the analysis of containment and equivalence of well-designed patterns and identifying the tractable components of their evaluation, we use GoSN to focus on the practical aspects of OPT pattern evaluation.

From the practical aspects of the optimization of SQL left-outer-joins, most prominently, Galindo-Legaria, Rosenthal [24, 25, 26] and Rao et al [38, 39] have proposed ways of achieving it through reordering inner and left-outer joins. Rao et al have proposed nullification and best-match operators to handle inconsistent variable bindings and subsumed results respectively (see Section 3.1). In their technique, nullification and best-match are required for each reordered query, as the minimality of tuples is not guaranteed (see Lemma 3.1). They do not use methods like semi-joins to prune the candidate tuples. Bernstein et al and Ullman [16, 17, 43] have proved the properties of minimality for acyclic inner-joins only. Through LBR, we have taken a step forward by extending these semi-join properties in the context of SPARQL OPT patterns, by analyzing the graph of jvar-nodes (GoJ), and finding ways to avoid overheads like nullification and best-match operations.

For inner-join optimization, RDF engines like TriAD [28], RDF-3X [29], gStore [16] take the approaches like graph summarization, sideways-information-passing etc for an early pruning of triples. Systems like TripleBit [15] use a variable length bitwise encoding of RDF triples, and a query plan generation that favors queries with “star” joins, i.e., many triple patterns joining over a single variable. RDF engines built on top of commercial databases such as DB2RDF [18] propose creation of entity-oriented flexible schemas and better data-flow techniques through the query plan to improve the performance of “star” join queries. Along with this, there are distributed RDF processing engines such as H-RDF-3X [29] and SHARD [10]. While many of these engines mainly focus on efficient indexing of RDF graphs, BGP queries, and exploiting “star” patterns in the queries, we have focused on the OPT patterns that may have multiple S-O joins, which cannot exploit the benefits of “star” query optimizers.

Our indexes and pruning procedure resemble the BitMat system [15]. However, BitMat only handles BGP queries. Their query graph cannot capture the structural aspects of a nested BGP-OPT query, and their query processing does not honor any left-outer-joins. Additionally, while using their index structure as a base in our system, we have enhanced it further to reduce as much as 40% of the index size compared to their original method [1].

**8. CONCLUSION**

In this paper, we proposed Left Bit Right (LBR), for optimizing SPARQL OPTIONAL pattern queries. We proposed a novel concept of a query graph of supernodes to capture the structure of a nested OPT query. We proposed optimization strategies – first of a kind, to the best of our knowledge – that extend the previously known properties of acyclicity, minimality, nullification, and best-match. We presented LBR’s evaluation in comparison with the two mainstream RDF columnstores, Virtuoso and MonetDB, and showed that LBR’s technique works much better for low-selectivity OPT queries with multiple OPT patterns, and for highly selective simpler queries, it gives at par performance with the other systems. In the future, we plan to extend our query processor to handle other SPARQL constructs such as unions, filters, and we intend to investigate methods for better cache management especially for short running queries.

**Acknowledgements**

We are grateful to Dr. Sujatha Das Gollapalli and Dr. Sudeepa Roy for their valuable inputs, and the three anonymous reviewers for their detailed feedback, which significantly helped us in improving the presentation of our paper.
9. REFERENCES

APPENDIX

A. PROOFS

A.1 Lemma 3.1

Proof. Let nullification be required with a minimal set of triples in each TP in a query. This means that there are one or more variable bindings and triples that will be removed as a result of an inner or left-outer-join that caused NULL values for the respective variable bindings. This is a contradiction to our assumption of minimality.

Let us assume that there are subsumed results with a minimal set of triples, and best-match is required to remove them. Let result $r_2$ be subsumed by $r_1$ ($r_2 \subseteq r_1$), and $non-null(r_1)$ and $non-null(r_2)$ be the non-NULL variable bindings in $r_1$ and $r_2$ respectively. Let $S_1 = non-null(r_1) \setminus non-null(r_2)$ and $S_2 = non-null(r_1) \cap non-null(r_2)$. Per the definition of subsumption, $non-null(r_2) \subset non-null(r_1)$. This means that $S_1$ are the variable bindings contributed by a “slave” TP, which is why they could be set to NULL in $r_2$ (see the definition of master-slave in Section 2.2). This also means that $S_2$ variable bindings had corresponding bindings of variables in $S_1$ in $r_1$ but not in $r_2$, implying that those $S_1$ bindings and the respective triples were removed in the process of joins. This is a contradiction to our assumption of minimality of triples. □

A.2 Lemma 3.2

Proof. Consider a query with three join variables, $?a$, $?b$, $?c$, that form a three-cycle in GoJ. Let us assume that the respective GoT of the query is acyclic. An edge between $?b$, $?c$, that form a three-cycle in GoJ. Let us assume that $A.3$ Lemma 3.3

A.3 Lemma 3.3

Proof. Let an OPT pattern be $P_k \sqsupseteq P_l$ with $P_k$ and $P_l$ as OPT-free BGP patterns. Let us temporarily ignore the left-outer-join between $P_k$ and $P_l$, and consider the two BGP patterns independently. $P_k$ and $P_l$ both have an acyclic connected GoJ. This observation follows from the fact that GoJ of $P_k$ and $P_l$ together is acyclic, and there are no Cartesian products in the query. Doing a bottom-up followed by top-down pass on the induced subtrees of $P_k$ and $P_l$ independently, with only clustered-semi-joins for each jvar, ensures that TPs in $P_k$ and $P_l$ are left with a minimal set of triples. This follows from Property 3.1 and is proved in [10]. Now consider $P_k \sqsupseteq P_l$. Let $J = jvars(P_l) \cap jvars(P_k)$, and $J \neq \emptyset$ because GoJ of $P_k \sqsupseteq P_l$ is connected. In Algorithm 3.1 line 11 chooses one of the jvars in $J$ to be the root of $P_l$’s induced subtree. Considering only jvars in $J$ and acyclicity, this means that $P_l$ has an induced subtree where jvar-nodes shared with its masters ($J$) appear as the ancestors of the jvars that appear only in $P_l$, but not in $P_k$.

A bottom-up pass over jvar subtrees of $P_k$ and $P_l$ (ln 11 in Alg 3.2), transfers the restrictions on the variable bindings across respective TPs in $P_k$ and $P_l$. Note that semi-join transfers the restrictions on variable bindings from $P_k$ to $P_l$, and clustered-semi-join transfers the restrictions on bindings among peers. Recall that we build order$_{id}$ such that induced subtree of $P_k$ is traversed before $P_l$‘s (ln 13–19 in Alg 3.1). A top-down pass on the jvar subtree of $P_k$ (ln 0–10 in Alg 3.2) leaves a minimal set of triples in the TP in $P_k$. Hence a top-down pass on the jvar subtree of $P_l$ after $P_k$‘s, leaves a minimal set of triples in the TPs in $P_l$ as a ripple effect. This analysis can be inductively applied for any nesting of BGP and OPT patterns, as long as their GoJ is acyclic and connected. Hence with the procedures in Algorithms 3.1 and 3.2 we can get a minimal set of triples in each TP in a nested acyclic BGP-OPT query. □

A.4 Lemma 3.4

Proof. We need to use nullification and best-match if a slave supernode has more than one jvars, because removal of bindings of one jvar may change the bindings of another due to the ripple effect through GoJ. Let us consider an OPT pattern of kind $tp_1 \sqsupseteq tp_3$, $(tp_2 \sqsupseteq tp_3)$. Let $P_1 = (tp_1)$ and $P_2 = (tp_2 \sqsupseteq tp_3)$. $P_2$ has only one jvar $?j_2$, that it shares with the master $P_1$. In prune_triples (Alg 3.2), we transfer the restrictions on bindings of $?j_2$ from $tp_1$ to $tp_2$ and $tp_3$ through a semi-join. Through a clustered-semi-join we ensure that restrictions on bindings of $?j_2$ are transferred among $tp_2$ and $tp_3$. So even if the original OPT pattern is reordered as $(tp_3 \sqsupseteq tp_2) \sqsupseteq tp_3$, there will be no subsumed results. This is because, after a clustered-semi-join, $tp_3$ does not have any bindings of $?j_2$ that do not appear in $tp_2$ and vice versa.

An alternate way to look at it is: Consider a cyclic OPT pattern where slaves have more than one jvars. Then if a final result $r_4$ of the query is subsumed by $r_3$ ($r_4 \subseteq r_3$), $non-null(r_3) \setminus non-null(r_4)$ have at least one jvar that appears only in the slaves of the query, i.e., $non-null(r_3) \setminus non-null(r_4)$ variables do not appear in absolute masters — that is why they could be set to NULLs. Let $S_1 = non-null(r_3) \setminus non-null(r_4)$, $S_2 = non-null(r_3) \setminus non-null(r_4)$. Note that not all $S_1$ can be non-jvars from slaves, because it would mean that while variables in $S_2$ had the respective bindings for those in $S_1$ in $r_3$, they did not find any bindings for $S_1$ in $r_4$. As per our prune_triples procedure (Alg. 3.2), we only prune the bindings of jvars; bindings of non-jvars get removed as a side effect of the pruning of jvar bindings. So if $S_1$ has all non-jvars, this will be a contradiction. Hence, $S_1$ would have at least one join variable that appears only in a slave, for subsumed result $r_4$. But for a cyclic OPT pattern (without Cartesian products) with each slave having only one jvar, each jvar appears in an absolute master too. This in turn means that none of the jvars in the query can be set to NULL. $S_1$ does not have any jvars that appear only in slaves, which means that subsumed results cannot be generated.

Thus we do not need nullification and best-match, for a cyclic OPT pattern query, where each slave has only one jvar. □
B. NON-WELL-DESIGNED PATTERNS

As defined by Pérez et al in [34], a SPARQL nested BGP-OPT pattern query \( P \) is said to be well-designed (WD) if for every subpattern \( P' = (P, \bowtie P_j) \) of \( P \), if a join variable \( ?j \) occurs in \( P \) and outside \( P' \), then \( ?j \) occurs in \( P \) as well.

An OPT pattern that violates this condition is called a non-well-designed (NWD) pattern.

Before we discuss the evaluation of NWD patterns, it is important to take into consideration how NULLs are treated in left-outer-joins or inner-joins. There is a disparity between the SPARQL and SQL algebra over this, which mainly creates a problem for NWD queries. The same NWD query evaluated over a pure SPARQL engine, e.g., Jena, gives counter-intuitive results than a SPARQL-over-SQL engine, e.g. Virtuoso, when there are joins over NULLs. This issue is discussed in detail with an example in Appendix C.

However, using the same assumption of “null-intolerant” (or null-rejecting) joins in SQL as done in the previous literature by Rao et al and Galindo-Legaria, Rosenthal [39, 26], we give a way of simplifying NWD patterns. In a “null-intolerant” join evaluation, NULL values are not matched with anything including other NULL values. E.g., in a left-outer-join \( P_a \bowtie P_b \), if \( ?j \) has NULL values in \( P_b \) before the join, they get eliminated during the left-outer-join evaluation. NULLs are only introduced as a result of a left-outer-join for missing values in the right hand side pattern.

For any OPT query (WD or NWD), we first serialize it using OPT-free BGP\'s, join operators \( \bowtie, \bowtie \bowtie \), and proper parentheses; then build a graph of supernodes (GoSN) as described in Section 2.1. If the given query is NWD, we identify all OPT-free BGP\'s that violate the WD condition, and the corresponding OPT-free BGP\'s with which they do a violation. E.g., if the query is \( P_z \bowtie (P_y \bowtie P_x) \), where \( P_z \) has a join variable \( ?j \) that appears in \( P_y \) but not in \( P_z \), then we say, \( "P_z \ violates WD condition with \( P_y \), and \( (P_y, P_x) \) is a violation pair"\). We identify all such violation pairs in the query, and their corresponding violation supernode pairs in the GoSN. For this example \( (SN_z, SN_x) \) will be the violation pair of supernodes. Temporarily ignoring the directionality of edges in GoSN, we identify the undirected path between each such supernode violation pair. Note that according to GoSN’s construction, there is a unique undirected path between any pair of supernodes. Next, considering the original directionality of edges on this path, we convert any unidirectional edges into bidirectional edges.

Conversion of unidirectional edges into bidirectional edges is continued this way until all the violation pairs in GoSN are treated. Note that this is a monotonic process, and always converges – we always convert unidirectional edges into bidirectional edges, and never the other way round. This effectively means we convert one or more left-outer-joins in the original query into inner-joins (recall that a unidirectional edge represents a left-outer-join, and a bidirectional edge represents inner-join). E.g., consider a serialized query \( (P_z \bowtie P_y) \bowtie ((P_x \bowtie P_d) \bowtie (P_e \bowtie P_f)) \), where \( P_b \) violates WD condition over variable \( ?j \), with \( P_c \), and \( P_f \) also violates WD condition over the same variable \( ?j \), with \( P_c \). In turn, \( P_b \) and \( P_f \) too violate WD condition with each other over \( ?j \). In Figure B.1, we show the original GoSN and the transformed GoSN for this query. The dotted lines in the original GoSN indicate the pairs of respective supernodes that are in violation of the WD condition.

After this transformation process, we use the same query processing techniques given in Algorithms 3.1, 3.2, and 5.1 on this GoSN with null-intolerant join assumption.

C. TREATMENT OF NULLS

Unlike relational tables, RDF instance data does not contain NULLs. Hence the issue of treatment of NULLs predominantly arises in case of non-well-designed OPT queries. This is because of the difference between the semantics of left-outer-joins between the SPARQL specifications and SQL algebra. Borrowing the definitions from Pérez et al in [34], if we have a pattern \( P_i \bowtie P_j \) with \( \Omega_i \) and \( \Omega_j \), as the set of mappings (variable bindings) of \( P_i \) and \( P_j \) respectively, then a SPARQL OPT pattern is defined as:

\[
\Omega_i \bowtie \Omega_j = (\Omega_i \bowtie \Omega_j) \cup (\Omega_i \setminus \Omega_j)
\]

This definition allows \( P_i \bowtie P_j \) to have results which may have different arity. For example, considering the query in Figure 3.2 per above definition, the results will be \( \{(:Larry), (:Julia, :Seinfeld)\} \). Note that the first result has arity 1, whereas the second one has arity 2.

This causes a problem if we do an inner-join of these results (mappings) with another pattern over \(?sitcom\) (as may happen in case of a non-well-designed query). Per SPARQL semantics of inner-join (\( \bowtie \)), two mappings are considered join-compatible, if they match on the variables that are bound in the respective mappings. For any unbound variables, the mappings are still considered compatible. E.g., if we join the above mappings with \( \{ (:Friends) \} \) over the \(?sitcom\) variable, the result will be counter-intuitive – \( \{ (:Larry, :Friends) \} \). Note that mapping \( (:Julia, :Seinfeld) \) got removed, but \( (:Larry) \) was preserved, because it does not have an explicit NULL value for \(?sitcom\).

In relational databases, evaluation of joins over NULL values has had different interpretations in different contexts (see [34, 35] for a discussion). However, for all practical purposes, most mainstream relational database systems assume a “null-intolerant” join evaluation. Also relational algebra assumes that the two sets of mappings (tuples) that are unioned, are union compatible, i.e., they have the same arity and same attributes. Due to these differences in the join and union semantics of SQL and SPARQL, RDF stores that are built on top of relational databases, e.g., Virtuoso, give different results for a non-well-designed query, compared to a native SPARQL processing engine, when joining over NULLs.

We observed that pure SPARQL processing engines, such as ARQ/Jena [3], follow the join semantics which allow union of mappings with different arity, and a null-tolerant join, whereas relational RDF stores such as Virtuoso or MonetDB follow the SQL semantics of union-compatibility and null-intolerant joins.

---

8 The difference between NULLs and “blank nodes” is elaborated in Section 2.2.
D. CONCEPTUAL BITCUBE

Borrowing the description of conceptual bitcube construction from [15], here we elaborate on the process of mapping the unique S, P, O values in the RDF data to the each bitcube dimension. As described in Section 4 the dimensions of the 3D bitcube of an RDF dataset are \(V_s \times V_p \times V_o\). The unique values of S, P, O in the original RDF data are first mapped to integer IDs, which in turn are mapped to respective bitcube dimensions. Let \(V_s = V_\text{so} \times V_p \times V_o\). Set \(V_s\) is mapped to a sequence of integers 1 to \(|V_s|\). Set \(V_p - V_\text{so}\) is mapped to a sequence of integers \(|V_s| + 1\) to \(|V_p|\). Set \(V_o - V_\text{so}\) is mapped to a sequence of integers 1 to \(|V_o|\). The common S-O identifier assignment \(V_\text{so}\) is for the sake of S-O joins.

The 2D BitMats are created by slicing this bitcube along each dimension, and they are compressed using our “hybrid compression” scheme as described in Section 4. Other metadata information such as, the number of triples, and condensed representation of all the non-empty rows and columns in each bitMat, is also stored along with them. This information helps us in quickly determining the number of triples in each BitMat and its selectivity without counting each triple in it, while processing the queries.

E. QUERIES

E.1 LUBM Queries

```xml
PREFIX ub: <http://www.lehigh.edu/~shp2/2004/0401/univ-bench.owl#>


E.2 UniProt queries

PREFIX uni: <http://purl.uniprot.org/core/>


Q7: SELECT * WHERE {{ ?univ uni:DiseaseAnnotation ?an . ?an a uni:Annotation . }}

E.3 DBPedia queries

PREFIX dbpedia: <http://dbpedia.org/resource/>

PREFIX dbpowl: <http://dbpedia.org/ontology/>

PREFIX dbpprop: <http://dbpedia.org/property/>

PREFIX dbpyago: <http://dbpedia.org/class/yago/>

PREFIX dbpcat: <http://dbpedia.org/resource/Category/>

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

PREFIX dbpedia: <http://dbpedia.org/resource/>

PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

PREFIX geo: <http://www.geos.org/geos/>