Efficient Execution of SPARQL Queries with OPTIONAL and UNION Expressions

Lei Zou, Yue Pang
Peking University
Beijing, China
{zoulei,michelle.py}@pku.edu.cn

M. Tamer Özsu
University of Waterloo
Waterloo, Canada
tamer.ozsu@uwaterloo.ca

Jiaqi Chen
Peking University
Beijing, China
chenjiaqi93@pku.edu.cn

ABSTRACT

The proliferation of RDF datasets has resulted in studies focusing on optimizing SPARQL query processing. Most existing work focuses on basic graph patterns (BGPs) and ignores other vital operators in SPARQL, such as `UNION` and `OPTIONAL`. SPARQL queries with these operators, which we abbreviate as SPARQL-UO, pose serious query plan generation challenges. In this paper, we propose techniques for executing SPARQL-UO queries using BGP execution as a building block, based on a novel BGP-based Evaluation (BE)-Tree representation of query plans. On top of this, we propose a series of *cost-driven BE-tree transformations* to generate more efficient plans by reducing the search space and intermediate result sizes, and a *candidate pruning* technique that further enhances efficiency at query time. Experiments confirm that our method outperforms the state-of-the-art by orders of magnitude.

1 INTRODUCTION

The proliferation of knowledge graphs has generated many RDF (Resource Description Framework) data management problems. RDF is the de-facto data model for knowledge graphs, where each edge is a triple of ⟨subject, predicate, object⟩. SPARQL has been the focus of a significant body of research as the standard language for accessing RDF datasets. Most of the existing work focuses on basic graph pattern (BGP) execution ([18, 24, 25]), which is the basic building block of SPARQL. On the other hand, how to execute and optimize queries containing operators on graph patterns, such as `UNION` and `OPTIONAL`, has received much less attention.

`UNION` and `OPTIONAL` expressions are essential in SPARQL grammar. RDF is a semi-structured data model that does not enforce the underlying data to adhere to a predefined schema. This provides flexibility in integrating diverse sources of RDF data, but leads to challenges when issuing queries since the same information can be represented in many ways in RDF graphs. The `UNION` operator is crucial in this case since it groups diversely expressed information. For example, in DBpedia [15], an open-domain knowledge graph extracted from Wikipedia, persons' names are represented using the predicate ⟨`foaf:name`⟩ or ⟨`rdfs:label`⟩. Thus, to fully retrieve all the names of a group of persons (e.g., Presidents of the United States), it is necessary to use the `UNION` operator (Figure 1(a)).

In addition to the diversity of representation, incompleteness is another feature of RDF datasets. Specifically, an entity may lack some attributes or relationships of most other similar entities (which are most likely to be stored in the same table in a relational database). The OPTIONAL operator is crucial in this case since it allows attaching some attributes or relations as optional information. For example, the `OPTIONAL` query in Figure 1(b) fetches all the presidents of the United States, along with other references to them that are not on the same Wikipedia page (through the predicate `owl:sameAs`). Since not every president has multiple references in the database, the triple with the predicate `owl:sameAs` is enclosed in an `OPTIONAL` expression, so those presidents without alternative references are still retained in the results.

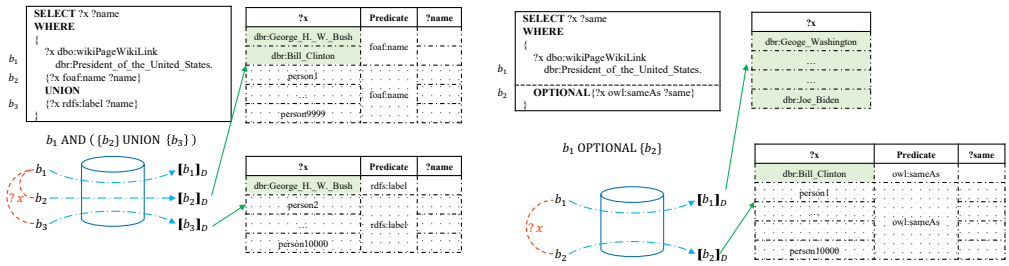
`UNION` and `OPTIONAL` expressions are widely used in real-world SPARQL workloads. Recent empirical studies [10] show that `UNION` and `OPTIONAL` expressions occur in 25.10% and 31.72% of the valid queries from real SPARQL query logs across a diverse range of endpoints, respectively. In this paper, we address the efficient execution of SPARQL queries with `UNION` and `OPTIONAL` expressions, which we abbreviate as SPARQL-UO queries.

Our Solution. Since BGP has been well studied, it is desirable to build SPARQL-UO query optimization on a well-performing BGP engine. Therefore, we first propose a BGP-based query evaluation scheme. Specifically, we propose a *B*GP-based *E*valuation (BE)-tree (Definition 8) for the evaluation plan of SPARQL-UO queries. However, if the BE-tree is evaluated as it is, some BGPs may generate large intermediate results. Therefore, we propose a BE-tree transformation method to generate a more efficient query plan.

We introduce two types of transformations, *merge* and *inject*, that target `UNION` and `OPTIONAL` operators, respectively. These transformations expose opportunities for reducing the cost during the evaluation of BGPs, `UNION` and `OPTIONAL` operators while preserving query semantics. Since there are many different ways to transform a BE-tree, we devise a cost model that accounts for the cost of evaluating both BGPs and complex expressions, and choose the transformation that most reduces the cost. Because of the vast space of possible transformations, we propose a greedy strategy to determine the transformation step-by-step. The transformed BE-tree is then evaluated by the BGP-based scheme, enhanced by the query-time optimization called *candidate pruning*, which prunes the search space of BGP evaluation on-the-fly whenever possible.

To summarize, we make the following contributions:

(1) We propose a novel BE-tree representation for the evaluation plan of a SPARQL-UO query and design two BE-tree transformation primitives, *merge* and *inject*, to generate more efficient SPARQL-UO query plans.
(2) We propose a cost model for SPARQL-UO queries and a cost-driven BE-tree transformation algorithm.
(3) We design a query-time optimization called *candidate pruning* that augments the BGP-based query evaluation scheme by pruning the search space.
(4) We conduct experiments on large-scale real and synthetic RDF datasets, which shows that our method outperforms existing techniques by orders of magnitude.



(a) UNION Clause (b) OPTIONAL Clause

Figure 1: An Example Query with a `UNION` and `OPTIONAL` Clause

2 RELATED WORK

Although the optimization of SPARQL queries has been extensively studied, most of the focus has been on evaluating BGPs [7], such as works on efficient data organization (e.g., property table [23] and vertical partitioning [4]), effective index strategies (e.g., RDF-3x [18], gStore [25] and BitMat [8]) and join order optimization (e.g., TripleBit [24], WCOJ [13]). Since the semantics of `UNION` and `OPTIONAL` differ fundamentally from joins, these techniques cannot be used on SPARQL-UO queries. As explained in later sections, our solution relies on BGP evaluation as a basic building block, and our proposed optimization techniques operate on a higher level than BGP evaluation techniques.

The existing research on SPARQL with `UNION` and `OPTIONAL` operators is primarily theoretical, studying their semantics and complexity [19]. For example, Letelier et al. [16] propose a WDPT (well-designed pattern tree), which focuses on the analysis of containment and equivalence of a class of SPARQL graph patterns called *well-designed patterns* and identifying the tractable components of their evaluation. However, no work has yet considered SPARQL-UO query optimization from the systems perspective, i.e., how to design an efficient SPARQL query processor to evaluate SPARQL-UO queries. To the best of our knowledge, LBR [7] is the only work that considers `OPTIONAL` query optimization. It designs a new data structure GoSN, which is reminiscent of WDPT but focuses on the practical aspects of `OPTIONAL` pattern evaluation. Concretely, it proposes a query rewriting technique to reduce intermediate results of left-outer joins, the join semantics represented by `OPTIONAL`. To remove inconsistent variable bindings, LBR uses the *nullification* and *best-match* techniques previously studied in SQL left-outer joins [21]. LBR also proposes a semijoin strategy to prune candidates, extending the operator for the minimality of acyclic inner joins [9]. However, it follows an execution strategy of two-pass semijoin scans following the graph of join variables, which introduces additional overhead during query execution. In this paper, we propose a more comprehensive approach that deals with `UNION` and `OPTIONAL`. Experiments also demonstrate that our techniques significantly outperform LBR on `OPTIONAL` queries.

An alternative choice is to rely on a relational DBMS and consider RDF graphs as three-column tables or other complex table organizations [5, 11]. Processing SPARQL queries is then mapped to its relational counterparts, as done in Apache Jena [23] and Virtuoso [3]. However, graph-based approaches can reduce the candidate vertex set size by building structure-aware indices (such as neighborhood-connection pruning [25]) and make use of efficient subgraph matching algorithms to reduce the intermediate results [17]. Experimental results confirm that graph-based approaches outperform relational counterparts significantly [14], especially for complex queries. Therefore, we develop graph-based techniques to process SPARQL-UO queries in this work.

Note that our techniques to optimize `UNION` expressions can also be applied to conjunctive relational queries with unions due to their semantic similarity. In fact, all SPARQL-UO queries can be equivalently mapped to SQL, but the mapping of `OPTIONAL` expressions involves sub-selects in SQL, so our techniques cannot be applied without major adaptations [12, 20]. We are aware of a recent demonstration [6] that proposes a *join pushing* technique on conjunctive queries with unions, which pushes the join condition into the unioned sets if a cost model deems it more efficient. This is principally similar to our approach when applied to relational queries, but no description of the employed cost model is provided, which renders further comparison impossible.

3 PRELIMINARIES

RDF Dataset. Table 1 is an example RDF dataset (defined as follows) containing 7 triples.

Definition 1 (RDF dataset). *Let pairwise disjoint infinite sets I, B, and L denote IRI, blank nodes and literals, respectively. An RDF dataset D is a collection of triples $D = \{t_1, t_2, ..., t_{|D|}\}$, where each triple is a three-tuple $t = \langle subject, property, object \rangle \in (I \cup B) \times I \times (I \cup B \cup L)$.*

SPARQL Query—Syntax. Assume that there is an infinite set V representing the variables that appear in the query. All variables differ from IRIs and literals by leading with a question mark (?), so the set V is disjoint with I and L. This work focuses on `SELECT` queries, which retrieve results by matching the graph pattern in the query with the dataset. We note that SPARQL provides other query forms for updating the database, constructing RDF datasets, asking whether a graph pattern exists in the database and describing resources, etc., that are beyond the scope of our consideration. A `SELECT` query is of the form "`SELECT` v_1 v_2 ... v_k `WHERE` {...}", in which the `SELECT` clause represents the query header, and the `WHERE` clause represents the query body (Figure 2(a)). The `SELECT` clause determines the projection variables that need to appear in the query results, and the `WHERE` clause gives the group graph pattern that

Table 1: An example RDF dataset

Subject	Predicate	Object
dbr:George_W._Bush	foaf:name	"George Walker Bush"@en
dbr:George_W._Bush	rdfs:label	"George W. Bush"@en
dbr:George_W._Bush	dbo:wikiPageWikiLink	dbr:President_of_the_United_States
dbr:Bill_Clinton	foaf:name	"Bill Clinton"@en
dbr:Bill_Clinton	dbo:wikiPageWikiLink	dbr:President_of_the_United_States
dbr:Bill_Clinton	dbp:birthDate	"1946-08-19"^^xsd:date
dbr:Bill_Clinton	owl:sameAs	fbp:Clinton_William_Jefferson_1946-

needs to be matched over the RDF dataset. Inside this graph pattern, many other types of graph patterns may appear, defined as follows.

Definition 2 (Triple Pattern). *A triple* $t \in (V \cup I) \times (V \cup I) \times (V \cup I \cup L)$ *is a* triple pattern.

In order to give a formal definition of BGPs, we need to first introduce the notion of *coalescability*.

Definition 3 (Coalescable triple patterns). *We say that the triple patterns* $t_1 = \langle s_1, p_1, o_1 \rangle$ *and* $t_2 = \langle s_2, p_2, o_2 \rangle$ *are coalescable if and only if* $\{s_1, o_1\}$ *and* $\{s_2, o_2\}$ *share at least one common variable.*

Intuitively, two triple patterns are coalescable if they have common variables at the subject or object positions. Since a BGP is composed of triple patterns, we can extend coalescability to BGPs, where we require some of their constituent triple patterns to be coalescable.

Definition 4 (Coalescable BGPs). *We say that the BGPs* b_1 *and* b_2 *are coalescable if there exist* $t_{i_1} \in b_1$ *and* $t_{i_2} \in b_2$ *such that* t_1 *and* t_2 *are coalescable triple patterns.*

Definition 5 (Basic Graph Pattern (BGP)). *A BGP is recursively defined as follows:*

(1) *A triple pattern t is a BGP;*
(2) *if* P_1 *and* P_2 *are coalescable BGPs,* P_1 AND P_2 *is also a BGP.*

Definition 6 (Graph Pattern, Group Graph Pattern). *A graph pattern is recursively defined as follows:*

(1) *if P is a BGP, P is a graph pattern;*
(2) *if P is a group graph pattern (defined below), P is a graph pattern;*
(3) *if* P_1 *and* P_2 *are both graph patterns,* P_1 AND P_2 *is also a graph pattern;*
(4) *if* P_1 *and* P_2 *are both graph patterns,* $\{P_1\}$ UNION $\{P_2\}$, P_1 OPTIONAL $\{P_2\}$ *are both graph patterns. Note that* $\{P_i\}$ *denotes a group graph pattern (defined below);*

A group graph pattern *P is recursively defined as follows:*

(1) *If P is a graph pattern,* $\{P\}$ *is a group graph pattern.*

Figure 2 is an example SPARQL query with six triple patterns ($t_{1..6}$), and UNION and OPTIONAL clauses.

SPARQL Query—Semantics. The semantics of any graph pattern can be uniquely determined, noting that the OPTIONAL clause is left-associative and the priority of operators is defined to be $\{\} \prec$ UNION $\prec$ AND $\prec$ OPTIONAL.

A graph pattern can then be converted to an expression containing triple patterns, built-in conditions, and binary operators AND, UNION and OPTIONAL. The AND, UNION, and OPTIONAL operators accept two graph patterns as their operands. Such an expression can

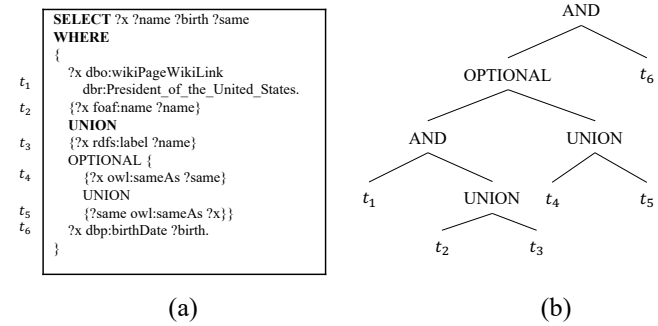


Figure 2: (a)An example SPARQL query and (b) Binary Tree Expression

be equivalently represented by a binary tree, where each leaf node represents a triple pattern or built-in condition, and each internal node represents a binary operator. Figure 2(b) shows such a binary tree expression of the outermost group graph pattern of the query in Figure 2(a).

A graph pattern P is matched on RDF dataset D (denoted by $[[P]]_D$) to produce a bag of mappings $\{\mu_1, \mu_2, \ldots, \mu_n\}$. Naturally, a bag (i.e., multi-set) may contain duplicate mappings. A mapping $\mu : V \mapsto U$ is a partial function from V to $(I \cup L)$, where V represents the variables that appear in the query, and I and L denote the sets of IRI and literals, respectively. The set of variables appearing in mapping μ is denoted by $dom(\mu)$. The two mappings μ_1 and μ_2 are defined to be *compatible* (denoted by $\mu_1 \sim \mu_2$) if and only if for all variables $v \in dom(\mu_1) \cap dom(\mu_2)$ satisfying $\mu_1(v) = \mu_2(v)$. Intuitively, this means that the common variables of μ_1 and μ_2 are mapped to the same values. In the case where μ_1 and μ_2 are compatible, $\mu_1 \cup \mu_2$ is also a mapping. If the two mappings μ_1 and μ_2 are *incompatible*, we denote the case as $\mu_1 \nsim \mu_2$.

We denote two bags of mappings by Ω_1 and Ω_2, and define several operators on bags as follows:

(1) $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$.
(2) $\Omega_1 \cup_{bag} \Omega_2 = \{\mu_1 \mid \mu_1 \in \Omega_1\} \cup_{bag} \{\mu_2 \mid \mu_2 \in \Omega_2\}$.
(3) $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 : \mu_1 \nsim \mu_2\}$.
(4) $\Omega_1 \leftthreetimes \Omega_2 = (\Omega_1 \bowtie \Omega_1) \cup_{bag} (\Omega_1 \setminus \Omega_1)$

Note that the operators above all preserve duplicate elements, as they follow the bag semantics.

Definition 7 (Evaluation of Graph Patterns on an RDF dataset). *The evaluation of graph patterns P on an RDF dataset D (denoted by* $[[P]]_D$*) is recursively defined as follows:*

(1) *If P is a triple pattern t,* $[[P]]_D = \{\mu \mid var(t) = dom(\mu) \wedge \mu(t) \in D\}$ *(*$var(t)$ *represents all variables occurring in t, and* $\mu(t)$ *mean that all variables appearing in t are replaced by* μ*).*
(2) *If* $P = \{P_1\}$, $[[P]]_D = [[P_1]]_D$.
(3) *If* $P = (P_1$ AND $P_2)$, $[[P]]_D = [[P_1]]_D \bowtie [[P_2]]_D$.
(4) *If* $P = (P_1$ UNION $P_2)$, $[[P]]_D = [[P_1]]_D \cup_{bag} [[P_2]]_D$.
(5) *If* $P = (P_1$ OPTIONAL $P_2)$, $[[P]]_D = [[P_1]]_D \leftthreetimes [[P_2]]_D$. *We say that* P_1 *is a* OPTIONAL *left graph pattern, and* P_2 *is a* OPTIONAL *right graph pattern.*

4 PLAN REPRESENTATION: BGP-BASED EVALUATION TREE

The most straightforward approach for evaluating a graph pattern P is to employ a bottom-up strategy. In each step, we either evaluate a triple pattern, or perform a binary operator (AND, UNION, or OPTIONAL). This binary-tree-based evaluation strictly follows SPARQL semantics discussed in Section 3, but it has a number of inherent performance limitations due to the large number of intermediate results generated for each triple pattern at the leaf nodes of the binary tree expression. To illustrate this, consider the simple SPARQL query in Figure 3. Note that the outermost group graph pattern of this query only contains a BGP. Following the binary tree expression-based method, we first need to obtain $[[t_1]]_D$ and $[[t_2]]_D$. Obviously, the triple pattern t_2 will generate a large number of intermediate results, since most persons in the database has their birth dates as an attribute.

It is evidently more desirable to use BGP evaluation as the basic building block for executing SPARQL queries, employing an optimized BGP query evaluation method such as those used in RDF-3x [18], SW-store [5] and gStore [25]. Therefore, in our approach, we design a BGP-based Evaluation Tree (BE-tree) to represent a SPARQL query evaluation plan.

4.1 BE-Tree Structure

Definition 8 (BGP-based Evaluation Tree (BE-tree)). *Given a group graph pattern Q, its corresponding BE-tree $T(Q)$ is recursively defined as follows:*

- *The root of $T(Q)$ is a group graph pattern node (Definition 6) representing the query Q;*
- *An internal node of $T(Q)$ can be one of {UNION, OPTIONAL, group graph pattern} node:*
  - *A UNION node represents the UNION expression that links two or more group graph patterns, called UNION'ed group graph patterns. It has two or more child nodes, which are all group graph pattern nodes;*
  - *An OPTIONAL node represents the OPTIONAL expression that links the graph patterns to its left and the adjacent group graph pattern to its right, called the OPTIONAL-right group graph pattern. It has exactly one child node, which is a group graph pattern node representing the adjacent group graph pattern to its right;*
- *A leaf node of $T(Q)$ is a BGP node, which represents a BGP (Definition 5).*

According to the above definition, each leaf node in a BE-tree corresponds to a BGP, and each internal node corresponds to a group graph pattern, a UNION expression, or an OPTIONAL expression. Figure 4 shows the general structure of a BE-tree. The edge labels indicate how many child nodes of this type is permitted to occur: k indicates that exactly k such child nodes must occur, and $k..*$ indicates that k or more such child nodes can occur. For convenience, we call a group of sibling nodes, that is the group of nodes that have the same parent node, as a *level* of nodes.

It is straightforward to construct a BE-tree from a SPARQL query. Joins between graph patterns are implicitly expressed in the BE-tree as the sibling relation between nodes. Therefore, we first initiate a group graph pattern node as the root, denoting the outermost group graph pattern in the query. Then we put each joined graph pattern within the outermost group graph pattern as the root's children in the original order. For each nested group graph pattern, we consider them in turn as the root of a sub-tree, and recursively execute the aforementioned process.

Note that such a construction procedure generates triple pattern nodes, which is not identified in Definition 8. In order to eliminate them, we coalesce sibling triple pattern nodes into *maximal* BGP nodes, in that no further coalescing can be performed (For the definition of coalescability, please refer to Definitions 3 and 4). We place BGP nodes where its constituent leftmost triple pattern originally resides. It is evident that there is a one-to-one mapping between SPARQL queries and BE-trees by this construction process.

As a concrete example, the BE-tree of the query in Figure 2(a) is given below (Figure 5). Note that the triple patterns t_1 and t_6 are coalesced to form a BGP node; no other triple patterns cannot be coalesced, and thus form individual BGP nodes on their own.

Algorithm 1: BGP-based query evaluation

Input: RDF dataset D, BE-tree $T(Q)$
Output: $[[Q]]_D$

```
1  Function BGPBasedEvaluation(D, T(Q)):
2      r ← ∅;
3      Let root be the root of T(Q) ;
4      foreach child node e_i of root do
5          if e_i is a group graph pattern node then
6              if r = ∅ then
7                  r ← BGPBasedEvaluation(D, T(e_i));
8              else
9                  r ← r ⋈ BGPBasedEvaluation(D, T(e_i));
10         else if e_i is a BGP node then
11             r ← r ⋈ EvaluateBGP(D, e_i);
12         else if e_i is a UNION node then
13             u ← ∅;
14             foreach child group graph pattern node P of e_i do
15                 u ← u ∪_bag BGPBasedEvaluation(D, T(P));
16             r ← r ⋈ u;
17         else if e_i is an OPTIONAL node then
18             Get the child group graph pattern node P of e_i;
19             o ← BGPBasedEvaluation(D, T(P));
20             r ← r ⟕ o;
21     return r
```

Algorithm 1 shows the pseudocode of the BGP-based solution for answering SPARQL query Q based on BE-tree $T(Q)$. The basic idea is to rely on the underlying BGP evaluation engine to evaluate each BGP separately, and combine the results afterwards based on the BE-tree. The return variable r, which indicates the result set, is first initialized to be empty (Line 2). Then the child nodes of the BE-tree's root are iterated and processed (Lines 4-20).

During the iteration across child nodes of the root, the following cases are considered:

- If the current child node is a group graph pattern node, it is recursively evaluated by calling the function on the subtree rooted at it, and the retrieved results are joined with r. (Lines 5-9)

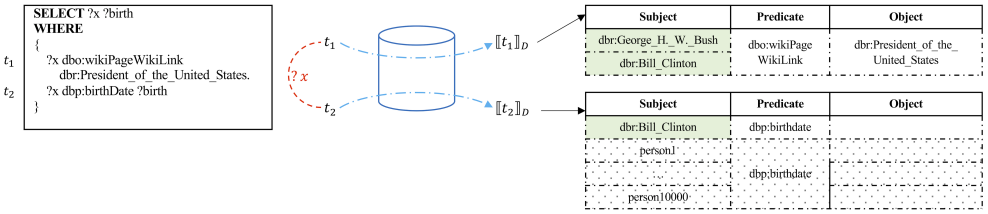


Figure 3: Inefficiency of binary-tree-based query evaluation

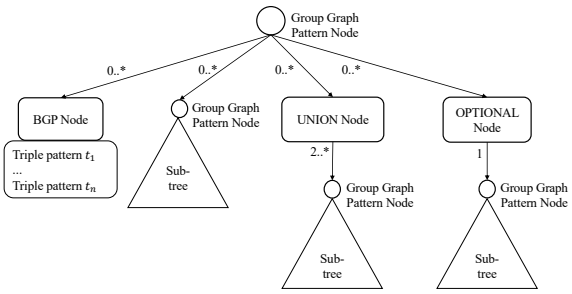


Figure 4: Hierarchical structure of the BE-tree

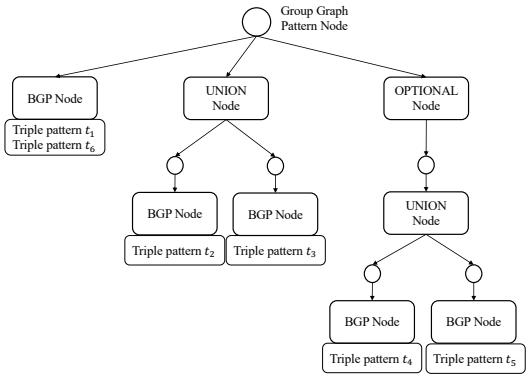


Figure 5: An example BE-tree

- If the current child node is a BGP node, it is evaluated by some existing BGP query evaluation technique, and the retrieved results are joined with r. (Lines 10-11)
- If the current child node is a UNION node, each of its child group graph pattern nodes is recursively evaluated, the results of which are merged by the $\cup_{bag}$ operation. The merged result is finally joined with r. (Lines 12-16)
- If the current child node is an OPTIONAL node, its child group graph pattern node is recursively evaluated, and the retrieved results are left-outer-joined with r. (Lines 17-20)

4.2 BE-Tree Transformations

In the previous subsection, we invoke the BGP-based evaluation procedure (Algorithm 1) on the BE-tree directly constructed from the query. However, it is possible to improve the efficiency of query evaluation by altering the plan. We achieve this by making certain semantics-preserving *transformations* to the BE-tree.

4.2.1 Goals. Our aim is to transform the original BE-tree so that the resulting BE-tree has the following properties:

- *Validity:* the resulting BE-tree maintains the previously defined tree structure, and has the same node types. It can be one-to-one mapped to a syntactically valid SPARQL query by the direct construction process introduced in Section 4.
- *Efficiency:* the evaluation of the resulting BE-tree is expected to be more efficient than that of the original BE-tree. In other words, the expected cost of evaluating the resulting BE-tree is lower.

4.2.2 Semantics-Preserving Transformations. With these two goals in mind, we set out to transform the BE-tree. In order to optimize for query execution efficiency while maintaining correctness, we need to leverage the inherent semantic equivalences regarding the UNION and OPTIONAL operators, formally expressed through the following two theorems[1].

Theorem 1. *For any graph pattern P_1, P_2, P_3 and any RDF dataset D, we have*

$$[[P_1 \texttt{ AND } (P_2 \texttt{ UNION } P_3)]]_D = [[(P_1 \texttt{ AND } P_2) \texttt{ UNION } (P_1 \texttt{ AND } P_3)]]_D.$$

Note that this is also trivially extendable to UNION nodes with more than two child nodes.

Theorem 2. *For any graph pattern P_1, P_2 and any RDF dataset D, we have*

$$[[P_1 \texttt{ OPTIONAL } P_2]]_D = [[P_1 \texttt{ OPTIONAL } (P_1 \texttt{ AND } P_2)]]_D.$$

These two equivalences correspond to two semantics-preserving transformations on the BE-tree: that of *merging* a node with the child nodes of its sibling UNION node, and that of *injecting* a node into the child node of its sibling OPTIONAL node. We define these transformations as follows.

Definition 9 (Merge transformation). *A merge transformation is the action performed on a node, which represents the graph pattern P_1, and one of its sibling UNION nodes, the child nodes of which represents the group graph patterns $P_2, P_3, \cdots, P_n$, when both of the following conditions are met:*

(1) *P_1 is a BGP node;*
(2) *At least one of the group graph patterns in $P_2, P_3, \cdots, P_n$ is the parent node of a BGP node that is coalescable with P_1.*

The action consists of the following steps:

[1]Due to space limit, we put all proofs in Appendix B.

(1) *Insert P_1 as the leftmost child node of $P_2, P_3, \cdots, P_n$;*
(2) *Coalesce P_1 with the other BGP child nodes if possible, until all the BGP nodes are maximal;*
(3) *Remove P_1 from its original position.*

Definition 10 (Inject transformation). *An inject transformation is the action performed on a node, which represents the graph pattern P_1, and one of its sibling* OPTIONAL *nodes to its right, the child node of which represents the group graph pattern P_2, when both of the following conditions are met:*

(1) *P_1 is a BGP node;*
(2) *P_2 is the parent node of a BGP node that is coalescable with P_1.*

The action consists of the following steps:

(1) *Insert P_1 as the leftmost child node of P_2;*
(2) *Coalesce P_1 with the other BGP child nodes if possible, until all the BGP child nodes are maximal.*

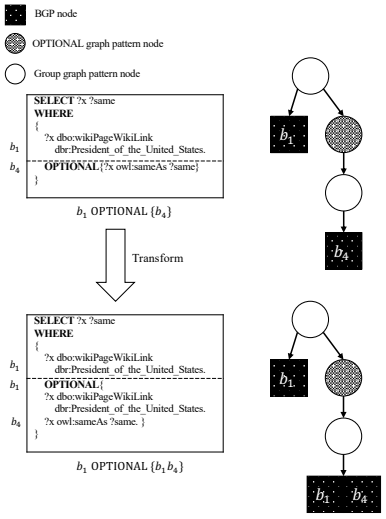


Figure 6: Favorable *Inject* Transformation

Figures 6 and 7 are examples of these two types of transformations in action. The graph database targeted by the queries in these figures is DBpedia, which is an encyclopedic open-domain knowledge graph containing information about a vast number of real-world entities. Via these examples, we give a qualitative overview of the effects of these transformations on the plan's efficiency.

In Figure 6, b_4 is a grandchild BGP node of the OPTIONAL node to the right of b_1, and b_1 and b_4 are coalescable. Therefore, the available *inject* transformation will coalesce b_4 with b_1. This transformation can help improve efficiency. According to the original BE-tree, b_4 is directly evaluated, and the results are left-outer-joined with those of b_1. Since a large number of entities have the ?sameAs relation, which denotes the equivalence between references to the same real-world object, b_4 has many matches, causing both its evaluation and the left-outer-join to be costly. However, presidents of the United States is a minority of the entities, making b_1 highly selective. The *inject* transformation takes advantage of this. After

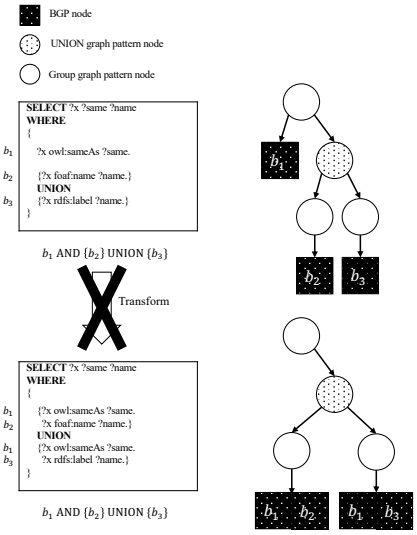


Figure 7: Unfavorable *Merge* Transformation

the *inject*, we can rely on the underlying evaluation engine to efficiently evaluate the coalesced $b_1 b_4$ by choosing a join order that evaluates the much more selective b_1 first. The left-outer-join is also rendered less expensive due to the decrease in the number of results of $b_1 b_4$ compared with b_4.

This example also helps explain the reason why certain conditions need to be met in Definitions 9 and 10. It is observable that only by coalescing is it possible to accelerate BGP evaluation. If no coalescing happens, the repetitive evaluation of the merged or injected BGP will instead incur extra overhead.

However, not all available transformations can help improve efficiency. Figure 7 shows an available *merge* transformation on an example UNION query, which merges the BGP b_1 with its sibling UNION node. Since b_1 has low selectivity, merging it does not accelerate BGP evaluation or reduce the number of intermediate results, and even incurs extra overhead because it now has to be evaluated twice.

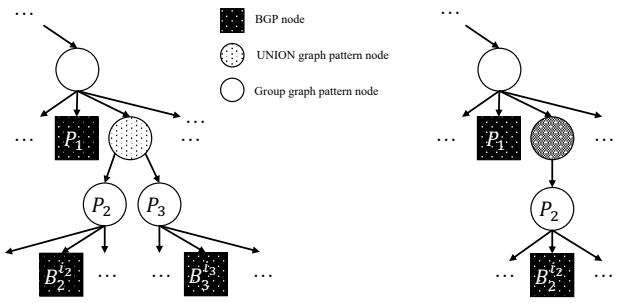
5 COST-DRIVEN PLAN SELECTION

In the previous section, we have established that there are differences in terms of efficiency among different semantics-preserving BE-tree transformations. This is the classical cost-based query plan selection problem. In this section, we introduce the cost model for evaluating a *merge* or *inject* transformation, and the algorithm based on it that decides the transformations to be performed given an original BE-tree.

Our cost model handles the BE-tree, and thus operates on a higher level than BGP evaluation. Nevertheless, our cost model still relies on estimations of the evaluation costs and result sizes of BGPs, which are obtainable as long as the workings of the underlying BGP evaluation engine are transparent. In addition, such estimations can often be directly obtained from the plan generation module of the underlying BGP evaluation engines [17]. For completeness, we briefly introduce the BGP cost model of gStore [25] and Jena [23], the two systems on which we implement our approach for experimentation in Section 5.1.2.

5.1 Cost Models

5.1.1 Cost Model for SPARQL-UO. The basic idea of our cost model is the insight drawn from the previous examples (Figures 6 and 7): SPARQL-UO query execution cost is made up of two main components: the cost of evaluating BGPs and the cost of combining partial results through UNION, OPTIONAL, or implicit AND operations. We are primarily concerned with the cost difference caused by a transformation, which we call Δ-*cost*. A transformation is expected to improve efficiency only when its Δ-cost is negative, indicating a decrease in cost; naturally we are looking for the transformation with the most negative Δ-cost.



(a) Merge (b) Inject

Figure 8: Estimating the Δcost for the *merge* transformation

In the following, we discuss how to estimate the Δ-*cost*. Consider a part of BE-tree shown in Figure 8 that contains a UNION node with two children group graph pattern nodes, P_2 and P_3 that have BGP child nodes, $B_2^{i_2}$ and $B_3^{i_3}$, respectively. Assume that $B_2^{i_2}$ and $B_3^{i_3}$ are coalescable with P_1. According to condition (2) in Definition 9, at most one of P_2 and P_3 lacks a coalescable BGP child node, which is represented by $B_2^{i_2}$ or $B_3^{i_3}$ as an empty node.

A *merge* transformation only affects a BGP node (P_1) and the BGP child nodes of its sibling UNION node ($B_2^{i_2}$ and $B_3^{i_3}$). After the transformation, the constituent triple patterns of these BGP nodes may change, but the occurrence of these nodes are maintained (for empty BGP nodes resulting from transformations are retained). Also, since the transformation preserves the query semantics, the evaluation results of P_1's parent node will not change. Therefore, the cost difference caused by *merge* is local to these nodes and their siblings, as shown in Figure 8(a). The local cost of the BE-tree before the *merge* transformation t_m can then be estimated as follows, where $l(\cdot)$ and $r(\cdot)$ denote all the sibling nodes to the left and right of the node $\cdot$, respectively:

$$cost(t_m) = cost(t_m, BGP) + cost(t_m, algebra) \tag{1}$$

$$cost(t_m, BGP) = cost(P_1) + cost(B_2^{i_2}) + cost(B_3^{i_3}) \tag{2}$$

$$\begin{aligned} cost(t_m, algebra) = {} & f_{AND}(|res(P_1)|, |res(l(P_1))|, |res(r(P_1))|) \\ & + f_{AND}(|res(B_2^{i_2})|, |res(l(B_2^{i_2}))|, |res(r(B_2^{i_2}))|) \\ & + f_{AND}(|res(B_3^{i_3})|, |res(l(B_3^{i_3}))|, |res(r(B_3^{i_3}))|) \\ & + f_{UNION}(|res(P_2)|, |res(P_3)|) \end{aligned} \tag{3}$$

$cost(t_m, BGP)$ can be directly obtained according to the BGP evaluation engine. $cost(t_m, algebra)$ is due to the possible change in the result sizes of the affected BGP nodes. In the case of *merge*, $cost(t_m, algebra)$ consists of the cost of performing implicit AND between the affected BGP nodes and their left and right siblings, and of performing UNION on P_2 and P_3. The costs of algebraic operations are functions on the result sizes of their operands (f_{AND} and f_{UNION} in Equation 3). These functions may differ based on different implementations of these algebraic operations. In our experiments, to fit the system we choose to build our implementation upon, f_{AND} is set to be the product of its arguments, and f_{UNION} is set to be the sum of its arguments.

Note that we need to also estimate the result sizes of some nodes for Δ-cost estimation. A BGP node's result size can be estimated by invoking or simulating an estimation module of the underlying BGP evaluation engine. The result sizes of other types of nodes need to be estimated based on an assumed distribution of data. In our experiments, we simply estimate the result size of any join (including AND and OPTIONAL) to be the product of the result sizes of the joined graph patterns, and the result size of UNION to be the sum of the result sizes of the UNION'ed graph patterns.

Suppose after the *merge* transformation, the affected nodes are turned into $P_1{}'$, $B_2^{i_2}{}'$ and $B_3^{i_3}{}'$. To estimate the local cost after t_m (denoted as $cost(t'_m)$), we simply replace P_1, $B_2^{i_2}$ and $B_3^{i_3}$ in Equations 2 and 3 by $P_1{}'$, $B_2^{i_2}{}'$ and $B_3^{i_3}{}'$. Consequently, the Δ-cost of *merge* can be estimated as follows:

$$\Delta cost(t_m) = cost(t'_m) - cost(t_m) \tag{4}$$

The case is similar for the *inject* operation (Figure 8(b)). The local cost of the BE-tree before the *inject* transformation t_i can then be estimated as follows:

$$cost(t_i) = cost(t_i, BGP) + cost(t_i, algebra) \tag{5}$$

$$cost(t_i, BGP) = cost(P_1) + cost(B_2^{i_2}) \tag{6}$$

$$\begin{aligned} cost(t_i, algebra) = {} & f_{AND}(|res(P_1)|, |res(l(P_1))|, |res(r(P_1))|) \\ & + f_{AND}(|res(B_2^{i_2})|, |res(l(B_2^{i_2}))|, |res(r(B_2^{i_2}))|) \\ & + f_{OPTIONAL}(|res(P_1)|, |res(P_2)|) \end{aligned} \tag{7}$$

Suppose after the *inject* transformation, the affected nodes are turned into $P_1{}'$ and $B_2^{i_2}{}'$. To estimate the local cost after t_i, we simply replace P_1 and $B_2^{i_2}$ in Equations 6 and 7 by $P_1{}'$ and $B_2^{i_2}{}'$. The Δ-cost of *inject* is then computed as follows:

$$\Delta cost(t_i) = cost(t'_i) - cost(t_i) \tag{8}$$

5.1.2 Cost Model for BGP. Although the underline BGP cost model is transparent to our SPARQL-UO cost model (see Equations 2 and 6), for the completeness of exposition, we briefly introduce the BGP cost models employed by gStore and Jena. The evaluation of BGPs consists of joins. Thus the cost of a BGP plan T is the sum of the costs of each executed join operation j:

$$cost(T) = \sum_{j \in T} cost(j)$$

BGP evaluation in gStore uses the worst-case optimal (WCO) join, which is concerned with all the edges labeled with the required predicate that links existing query vertices and the newly extended

vertex. For each result tuple on the existing vertices, all such edges need to be scanned at least once to check whether this tuple can be extended to match the newly extended vertex. Suppose the set of existing vertices is $\{v_1, \cdots, v_{k-1}\}$, and the newly extended vertex is v_k. The cost of a WCO join can then be estimated as follows:

$$\begin{aligned} &cost(WCOJoin(\{v_1, \cdots, v_{k-1}\}, v_k)) \\ =&card(\{v_1, \cdots, v_{k-1}\}) \times \min_{i \in [1,k-1]} average\_size(v_i, p) \end{aligned}$$

where $card(\{v_1, \cdots, v_{k-1}\})$ indicates the estimated *cardinality* – the estimated number of result tuples on the query vertex set $\{v_1, \cdots, v_{k-1}\}$; and $average\_size(v_i, p)$ indicates the average number of edges (*i.e.,* triples) with p as predicate and v_i as subject or object, depending on the direction of the edge between v_i and v_k in the query.

On the other hand, BGP evaluation in Jena uses the binary join, which is conceptually akin to a hash-join in relational databases. It first hashes the result tuples of the BGP with a smaller result size on the common vertices. Then, for each result tuple of the other BGP, the hash index is probed to find compatible matches that can be combined. Suppose the two BGPs to be combined have query vertex sets V_1 and V_2, respectively. The cost of a binary join can then be estimated as follows:

$$\begin{aligned} &cost(BinaryJoin(V_1, V_2)) \\ =&2 \times \min(card(V_1), card(V_2)) + \max(card(V_1), card(V_2)) \end{aligned} \tag{9}$$

where the first part of the sum indicates the cost of building the hash index, and the second part indicates the cost of probing it.

The above cost estimation formulas rely on the cardinality estimation of query vertex sets. Cardinality estimation starts from single triple patterns, whose query vertex set's exact cardinality can be obtained reading the pre-built indexes of the RDF store using the constants as key. Each time that a new query vertex is added to the set, we sample the candidate result set, and collate how many result tuples can be generated from the sample by extending to the new query vertex. The estimated cardinality is updated by scaling up based on the previous estimation in proportion to the ratio between the number of extended result tuples and the sample size:

$$card(V_k) = \max(\frac{\#extend}{\#sample} \times card(V_{k-1}), 1)$$

Note that the more sophisticated cardinality estimation approaches and BGP cost models (such as [22]) are orthogonal to our contribution in this paper. Experimental results show that our approach optimize SPARQL-UO query processing significantly by considering the simple but effective BGP cost models and cardinality estimation methods shown above.

5.2 Cost-Driven Transformation

We discuss BE-tree transformation algorithms that leverage the cost model discussed above to decide which transformations to perform to obtain the most efficient query plan for execution by Algorithm 1.

5.2.1 Transforming a Single Tree Level. We first concentrate on the simpler case where only transformations at a single level are considered.

When a BGP node only has a sibling `UNION` or `OPTIONAL` node, deciding the transformations is already covered by the cost model introduced in Section 5.1.1. However, in reality, multiple sibling `UNION` or `OPTIONAL` nodes may be viable for transformation. Note that according to Theorems 1 and 2, a merged BGP is removed from its original position, while an injected BGP maintains its original occurrence. This means that a BGP can only be merged with one of its sibling `UNION` nodes, but can be injected into multiple sibling `OPTIONAL` nodes. Therefore, in order to decide on a *merge* transformation, we need to look holistically at all the `UNION` nodes at that level, and choose the transformation that incurs the lowest Δcost. On the other hand, *inject* transformations are mutually independent, so we scan over each `OPTIONAL` node to the right of the BGP node, and decide individually which ones are worthy of a transformation based on Δcost.

The transformation decision at a single level of the BE-tree is given in Algorithm 2. The transformation happens at the level of the children of the input group graph pattern node P. Note that the *merge* transformation of a BGP node can only be determined and performed after iterating over all its sibling UNION nodes (Line 14), while the *inject* operation is decided individually on each sibling `OPTIONAL` node (Line 16). The subroutines that compute the Δ-cost of each possible transformation is presented in Algorithm 3.

Algorithm 2: Single-level BE-tree transformation

```
   Input: RDF dataset D, BE-tree T(Q), a group graph pattern node P
1  Function SingleLevelTransform(D, T(Q), P):
2      foreach P_1 in the child nodes of P do
3          if P_1 is a BGP node then
4              minUnionCost ← 0;
5              targetUNION ← empty node;
6              foreach UNION node u in the child nodes of P do
7                  minUnionCostCur ← DecideMerge (P_1, u);
8                  if minUnionCostCur < minUnionCost then
9                      minUnionCost ← minUnionCostCur;
10                     targetUNION ← U;
11             if minUnionCost < 0 then
12                 Perform merge on subBGPglobal and targetUNION
13             foreach OPTIONAL node o to the right of P_1 in the child
                nodes of P do
14                 DecideInject (P_1, O);
```

5.2.2 Handling Multiple Levels. Handling the entire BE-tree, which often consists of multiple levels is particularly challenging because of the possible interdependence between transformations across different levels. For example, if we consider transforming the group graph pattern $\{P_1$ `OPTIONAL` $\{P_2$ `OPTIONAL` $P_3\}\}$ (P_1, P_2 and P_3 are all coalescable BGPs), there are 2^3 possible transformations involving whether P_1 is injected into P_2, whether P_2 is injected into P_3, and whether P_1 is injected into P_3. This results in a plan space that is exponential in terms of the depth of the BE-tree. In fact, we conjecture that finding the optimal transformation on the entire BE-tree is an NP-hard combinatorial optimization problem.

Algorithm 3: Subroutines for BE-tree transformation

```
1  Function DecideMerge(P1, U):
2      if constraints are violated then
3          return 0;
4      originalCost ← local cost (Equations 1, 2 and 3);
5      minUnionCostCur ← 0;
6      foreach child group graph pattern node Pj of U do
7          BSetj ← {Bji | Bji is a BGP child node of Pj coalescable
             with P1};
8          if BSetj = ∅ then
9              Add an empty BGP node to BSetj;
10     foreach tuple (B2i2, B3i3, ···) drawn from BSet2, BSet3, ··· do
11         Perform merge on P1 and U;
12         transformedCost ← local cost (Equations 1, 2 and 3);
13         Δcost ← transformedCost - originalCost;
14         if Δcost < minUnionCostCur then
15             minUnionCostCur ← Δcost;
16         Undo merge;
17     return minUnionCostCur;
18 Function DecideInject(P1, O):
19     if constraints are violated then
20         return;
21     originalCost ← local cost (Equations 1, 6 and 7);
22     foreach BGP child node B2i of O's child group graph pattern
         node P2 coalescable with P1 do
23         Perform inject on P1 and U (coalescing sub with B2i);
24         transformedCost ← local cost (Equations 1, 6 and 7);
25         Δcost ← transformedCost - originalCost;
26         if Δcost ≥ 0 then
27             Undo inject;
```

Algorithm 4: Multi-level BE-tree transformation

```
   Input: RDF dataset D, BE-tree T(Q)
1  Function MultiLevelTransform(D, T(Q)):
2      PostOrderTraverse (D, T(Q), Q);
3  Function PostOrderTraverse(D, T(Q), P):
4      foreach P1 in the child nodes of P do
5          if P1 is a group graph pattern node then
6              PostOrderTraverse (D, T(Q), P1);
7          else if P1 is a UNION node then
8              foreach child group graph pattern node Pi of P1 do
9                  PostOrderTraverse (D, T(Q), Pi);
10         else if P1 is an OPTIONAL node then
11             Get the child group graph pattern node P2 of P1;
12             PostOrderTraverse (D, T(Q), P2);
13     SingleLevelTransform (D, T(Q), P);
```

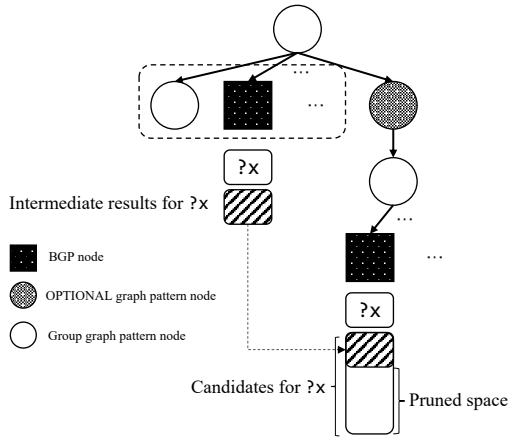


Figure 9: Candidate Pruning for `OPTIONAL`

In order to balance the time complexity and the efficiency of the transformed tree, we propose a greedy strategy to decide on the transformations on the entire BE-tree (Algorithm 4). Specifically, we traverse the BE-tree in a post-order depth-first fashion. Only when all the child nodes of a group graph pattern node have been traversed (Lines 4-12), we consider the possible transformations on the level of its children (Line 13, which invokes Algorithm 2). In this way, we ensure that all the lower levels have been appropriately transformed before considering transforming the current level, and the entire transformed tree is guaranteed to be more efficient than the original without expensive backtracking.

After applying the transformations, the BE-tree still maintains the tree structure and has the same node types. The semantic correctness of the transformed BE-tree is guaranteed by the aforementioned equivalences. Therefore, the evaluation algorithm (Algorithm 1) can still be invoked to evaluate the transformed BE-tree.

6 QUERY-TIME OPTIMIZATION: CANDIDATE PRUNING

In the previous section, we introduced how to generate different SPARQL-UO query plans by BE-tree transformations and select an effective plan based on the cost estimation prior to execution. In this section, we present *candidate pruning*, a query-time optimization incorporated into Algorithm 1 to enhance efficiency.

The basic idea of candidate pruning is also drawn from Theorems 1 and 2. The equivalence between the evaluation results implies that the results of the `UNION`'ed or `OPTIONAL`-right group graph patterns are constrained by those of the outer graph pattern regarding the common variables. Therefore, when a `UNION`, `OPTIONAL` or group graph pattern node is encountered during evaluation, we can set the current results on the common variables as *candidate results* when executing the child BGPs of that node. Figure 9 shows the mechanism of candidate pruning for an `OPTIONAL` query: the results of the variable `?x` from the already evaluated graph patterns serve as the candidate results of `?x` for the child BGP of the `OPTIONAL`-right group graph pattern, pruning redundant matchings of `?x` that will be materialized if the BGP is evaluated independently.

The discussion above establishes that candidate pruning preserves semantic correctness. However, to achieve a pruning effect, we need to ensure that the size of the candidate results is smaller than the size of the actual results of the BGP. A smaller candidate result size also reduces the overhead incurred by scanning them and setting them as candidates. We adopt an adaptive threshold on the candidate result size. The cost model for BGP (Section 5.1.2) invoked as part of tree transformation provides an estimate of the actual BGP result size, which we employ as the threshold on candidate result size whenever possible. When no such estimate is

available, we set the threshold based on the dataset size. (Please refer to Section 7 for the threshold setting in our experiments.)

To implement candidate pruning, we modify Algorithm 1 as follows (Note that the results can be passed as arguments in the form of pointers to prevent expensive copying):

- Add a third argument *cand*, which denotes the candidate results, to the `BGPBasedEvaluation` function;
- Pass the current results r as the third argument to `BGPBasedEvaluation` when processing a `UNION`, `OPTIONAL` or group graph pattern node (Lines 7, 9, 15 and 19);
- Pass *cand* as the third argument to `EvaluateBGP` (Line 11). Only when the size of *cand* is smaller than the threshold is it set as the candidate results of the BGP.

Tree transformation and candidate pruning, which take effect prior to and during query execution, respectively, are complementary to each other. Prior to execution, high-selectivity BGPs are targeted by *merge* or *inject* transformations, which breaks up graph patterns with large overall results that originally cannot be handled by candidate pruning. Tree transformation also supplies candidate pruning with estimates of the BGP result sizes. On the other hand, while tree transformations are constrained to be performed level-by-level due to the vast plan space, candidate pruning can transmit the pruning effect of small results across levels during execution. For example, when processing a query with the group graph pattern $\{P_1$ `OPTIONAL` $\{P_2$ `OPTIONAL` $P_3\}\}$, P_1 cannot be injected into P_3 by the greedy transformation strategy even if it is selective, but its results can serve as candidates for P_3 via P_2. In the special case where there is only a BGP node to the left of the `UNION` or `OPTIONAL` node, performing transformations on the BGP is equivalent to candidate pruning. In this case, tree transformation is skipped to evade the additional overhead.

Table 2: Datasets Statistics

Datasets	triples	entities	predicates	literals
LUBM	534,355,247	86,990,882	18	44,658,530
DBpedia	830,030,460	96,375,582	57,471	59,825,935

Table 3: Query Statistics on LUBM

	query	type	$Count_{BGP}(Q)$	$Depth(Q)$	$\lvert[\![Q]\!]_D\rvert$
Group 1	q1.1	U	9	2	645,666
	q1.2	O	3	2	44,653,510
	q1.3	O	4	4	76
	q1.4	O	4	4	5,583
	q1.5	UO	6	3	4,348
	q1.6	UO	9	3	37
Group 2	q2.1	O	3	1	4,176,432
	q2.2	O	4	3	8,698
	q2.3	O	4	3	13,124,940
	q2.4	O	2	3	10
	q2.5	O	2	2	10
	q2.6	O	2	2	7

Table 4: Query Statistics on DBpedia

	query	type	$Count_{BGP}(Q)$	$Depth(Q)$	$\lvert[\![Q]\!]_D\rvert$
Group 1	q1.1	U	6	2	153,325
	q1.2	UO	4	3	610,434
	q1.3	O	5	5	1,192
	q1.4	UO	7	5	92,041
	q1.5	UO	6	3	3,699,995
	q1.6	UO	10	4	176
Group 2	q2.1	O	5	3	490,876
	q2.2	O	2	2	55,054
	q2.3	O	2	2	61,318
	q2.4	O	3	2	4,757
	q2.5	O	2	2	5,330
	q2.6	O	9	2	36

7 EXPERIMENTS

To evaluate the effectiveness, we employ the BGP query engines of Jena and gStore to implement our BGP-based cost-aware SPARQL-UO evaluation strategy. We pull the latest version of Jena as of 27 June, 2022 from their GitHub repository[2]. All the experiments run on Jena have enabled the statistics-based optimizations. We forked a branch from the main branch of gStore and implement our proposed SPARQL-UO optimizer based on it[3]. Experiments are conducted on both synthetic (LUBM [2]) and real (DBpedia[4][1]) RDF datasets, the statistics of which are listed in Table 2. Our implementation and all the queries used in our experiments can be found in our anonymous GitHub repository[5]. We conduct experiments on a Linux server with an Intel Xeon Gold 6126 CPU @ 2.60GHz CPU and 256GB memory.

7.1 Verification of Optimizations

In this section, we verify the effectiveness of the proposed optimizations in Section 4.2 and evaluate the following four approaches:

(1) The baseline (abbreviated as `base`), which invokes the BGP-based query evaluation method (Algorithm 1) on the original BE-tree;
(2) Tree transformation (abbreviated as `TT`), which transforms the original BE-tree by Algorithm 4 and then invokes Algorithm 1 on it;
(3) Candidate pruning (abbreviated as `CP`), which invokes Algorithm 1 augmented with candidate pruning (Section 6) on the original BE-tree, using a fixed threshold of 1% of the total number of triples in the database;
(4) The full version that coordinates tree transformation and candidate pruning (abbreviated as `full`), which transforms the original BE-tree by Algorithm 4, and then invokes Algorithm 1 augmented by candidate pruning, using an adaptive threshold on the candidate result size.

Since there is no benchmark tailored for SPARQL-UO queries to our knowledge, we construct a mini-benchmark with realistic semantics and varying complexities, containing six queries on LUBM

[2]https://github.com/apache/jena.
[3]Our implementation is available at https://anonymous.4open.science/r/gStore-UO/.
[4]The DBpedia data dump that we use is V3.9, which is downloadable at http://downloads.dbpedia.org/3.9/en/. We use the concatenation of all the N-Triples files.
[5]https://anonymous.4open.science/r/gStore-UO-opt/.

and DBpedia, respectively, denoted as q1.1-1.6 in the following, given in Appendix A. Let Q be the outermost group graph pattern in the query. To measure the complexity of a query, we define two metrics: (1) the BGP count ($Count_{BGP}(Q)$), and (2) the maximum depth of nested group graph patterns ($Depth(Q)$).

$Count_{BGP}(P)$ of a graph pattern P is recursively defined:

(1) If P is a BGP, $Count_{BGP}(P) = 1$.
(2) If $P = \{P_1\}$, $Count_{BGP}(P) = Count_{BGP}(P_1)$.
(3) If $P = P_1$ AND P_2 or P_1 UNION P_2 or P_1 OPTIONAL P_2, $Count_{BGP}(P) = Count_{BGP}(P_1) + Count_{BGP}(P_2)$.

$Depth(P)$ of a graph pattern P is recursively defined as follows:

(1) If P is a BGP, $Depth(P) = 0$.
(2) If $P = \{P_1\}$, $Depth(P) = Depth(P_1) + 1$.
(3) If $P = P_1$ AND P_2 or P_1 UNION P_2 or P_1 OPTIONAL P_2, $Depth(P) = \max(Depth(P_1), Depth(P_2))$.

Suppose P is the outermost group graph pattern of query Q, we have $Count_{BGP}(Q) = Count_{BGP}(P), Depth(Q) = Depth(P)$. Group 1 in Tables 3 and 4 summarizes the statistics and the result sizes of the queries used in this subsection.

We measure the performance by the query execution time. We also report the time spent carrying out the tree transformations for TT and full. The performance of our approaches on LUBM and DBpedia is shown in Figure 10. The absence of a bar indicates an out-of-memory error on the query. A query is considered timed-out if the execution time exceeds 2×10^6 microseconds.

The trend of the results across gStore and Jena is similar, showing the adaptability of our approach regardless of the underlying BGP execution engine. Both of our proposed optimizations are shown to be effective since TT, CP and full perform better than base on all queries. TT and CP can be more advantageous on different queries and datasets than the other. Their optimization effects are cumulative when combined: full performs best all queries and datasets (except on q1.2 on gStore, where CP beats full by a small margin), beating the baseline by at least 2x and up to over an order of magnitude. Our optimized approaches also consume less memory. While base runs out of memory on 13 out of 24 queries, full successfully runs all the queries.

In the following, we try to draw some conclusions about the applicability of our optimizations to different SPARQL-UO queries by analyzing the benchmark queries and the behavior of the optimized approaches on them.

When TT is effective. q1.1 on DBpedia (given in Appendix A) is a query on which TT is effective, but CP is not. In this query, two UNION clauses are given first (Lines 11-12), whose child BGPs all have low selectivity. There is no high-selectivity graph pattern before them to enable CP. However, TT can merge the high-selectivity BGP in Lines 14-17 with the UNION clause in Line 12 to accelerate query processing and reduce memory overhead, as evidenced in Figure 10. q1.2 on LUBM and q1.2 on DBpedia also belong to this category. (Note that q1.2 on LUBM corresponds to the special case mentioned in Section 6, where there is only a BGP before an OPTIONAL clause, and thus TT and CP have a similar effect.)

When CP is effective. q1.3 on LUBM (given in Appendix A) is a query on which CP is effective, but TT is not. In this query, the BGP in Line 5 has high selectivity, followed by nested OPTIONALs with low-selectivity child BGPs. TT can inject the BGP into the outermost OPTIONAL but cannot reach the inner OPTIONALs, thus having limited effect. However, CP can carry the small number of results into the innermost OPTIONAL and set them as candidates to accelerate query processing. q1.3-4 on LUBM and q1.3-4 on DBpedia also belong to this category.

When TT and CP are jointly effective. q1.6 on LUBM (Listing 7 in Appendix B) is a query on which TT and CP work complementarily, causing full to perform much better than TT and CP. In this query, the BGP in Lines 4-5 has high selectivity, while the BGP in Line 6 has relatively low selectivity. Upon obtaining their considerably large results, CP has limited effect on the following UNION clauses. TT, however, can pick the high-selectivity BGP to merge with the UNION in Line 7. Having executed the graph patterns up to Line 8, CP can accelerate the processing of upcoming OPTIONALs. q1.1 and q1.5 on LUBM and q1.5 and q1.6 DBpedia also belong to this category.

For a quantitative perspective on the optimization effects, we define the join space of a graph pattern $JS(P)$ as follows:

(1) If P is a BGP, $JS(P) = |[\![P]\!]_D|$.
(2) If $P = \{P_1\}$, $JS(P) = JS(P_1)$.
(3) If $P = P_1$ AND P_2 or P_1 OPTIONAL P_2, $JS(P) = JS(P_1) \times JS(P_2)$.
(4) If $P = P_1$ UNION P_2, $JS(P) = JS(P_1) + JS(P_2)$.

The join space of a query estimates the largest intermediate result size that is materialized during the execution of this query. Therefore, it is indicative of both the query's execution time and memory overhead. We plot the execution time of all the queries on gStore and Jena (the y-axis on the left) with their respective join spaces (the y-axis on the right) in Figure 11. Across the tested approaches, these three metrics show a similar trend. On all the queries, the join spaces of TT and CP are smaller than those of base, and full has the smallest join space overall, which corroborates the qualitative analysis above.

7.2 Comparison with State-of-the-Art

The only work that considers SPARQL with OPTIONAL query optimization is LBR [7]. Thus, we compare our full approach with LBR. We also implement LBR in C++. We experiment on the queries provided in LBR [7] on LUBM and DBpedia, listed as q2.1-2.6, given in Appendix A. The statistics of these queries are given in the second group in Tables 3 and 4. q2.1-2.3 are complex with multiple nested group graph patterns, each containing a low-selectivity BGP followed by an OPTIONAL with a single low-selectivity child BGP. Meanwhile, q2.4-2.6 are simple without nested group graph patterns, and their outermost group graph pattern contains a high-selectivity BGP followed by an OPTIONAL.

The total response time of full and LBR are shown in Figures 13. full is significantly faster than LBR on all queries, and the improvement on q2.4-2.6 is more significant than on q2.1-2.3. This is because candidate pruning can take advantage of the high-selectivity BGPs in q2.4-2.6, while q2.1-2.3 does not contain high-selectivity BGPs. (Note that since all the group graph patterns in q2.1-2.6 contain a BGP followed by an OPTIONAL clause, they correspond to the special case mentioned in Section 6 where tree transformation and candidate pruning are equivalent, hence only candidate pruning is

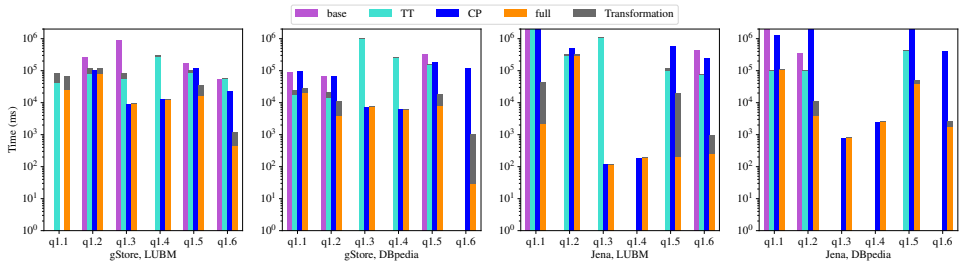


Figure 10: Verification of optimizations.

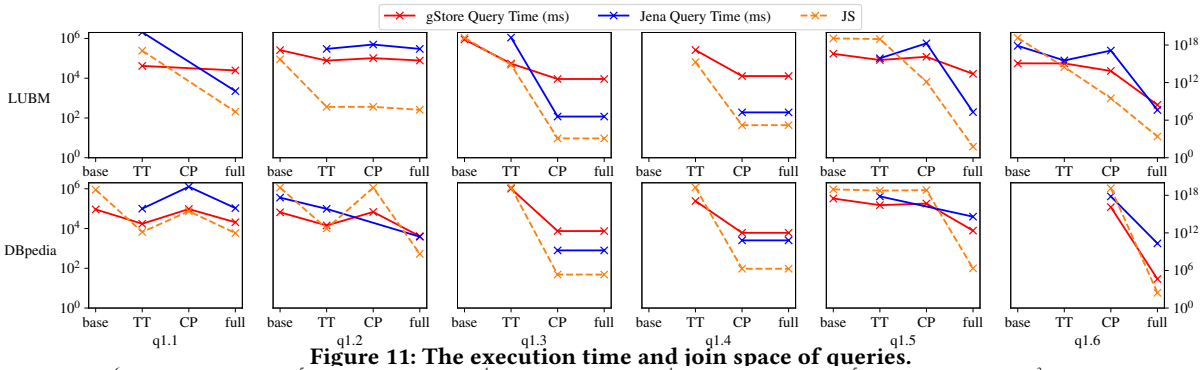


Figure 11: The execution time and join space of queries.

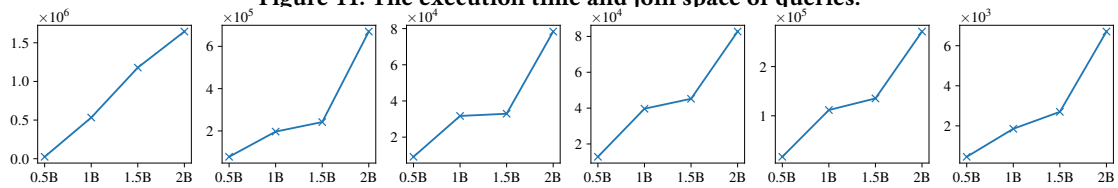


Figure 12: Query execution time (ms) of `full` on LUBM datasets of different sizes ("B" is short for billion).

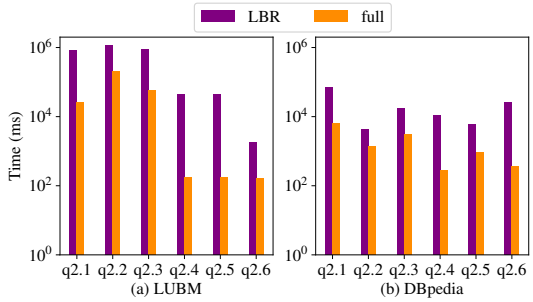


Figure 13: Comparison with state-of-the-art on LUBM.

performed.) The results show that when candidate pruning takes effect, it is more efficient than LBR's heavy-weight pruning strategies. On q2.1-2.3, `full` is still faster than LBR since its BGP-based evaluation scheme is more efficient than LBR's separate treatment of triple patterns.

In summary, our approach outperforms LBR on `OPTIONAL` queries, despite LBR being optimized for `OPTIONAL`.

7.3 Scalability Study

Lastly, we evaluate how well our approach scales to larger datasets. By setting the scaling factor of LUBM, i.e., the number of universities, we generate three more LUBM datasets with 1, 1.5 and 2 billion triples, respectively. We run the `full` approach on q1.1-q1.6 on these datasets and plot how the execution time changes with the dataset size on each query in Figure 12.

These plots are empirical complexity curves of our approach. Our approach scales almost linearly to the number of triples in the datasets. The growth rate of the query execution time correlates with each query's result sizes: the execution time of queries with larger result sizes grows faster with the dataset size. (The result sizes of q1.3-1.6 on larger LUBM datasets are equal to those shown in Table 3, while those of q1.1-1.2 grow linearly.)

8 CONCLUSION

The proliferation of knowledge graph applications has generated increasing RDF data management problems. In this paper, we focus on how to optimize SPARQL queries with `UNION` and `OPTIONAL` clauses (SPARQL-UO for short). Making use of existing BGP query evaluation modules in SPARQL engines, we propose a series of cost-driven transformations on the BGP-based evaluation tree (BE-tree). These optimizations can significantly reduce the search space and intermediate result sizes, and thus improve both the time and space efficiency of SPARQL-UO query evaluation. We experimentally validate the effectiveness of our optimizations, and compare the performance of the optimized method with the state-of-the-art on large-scale synthetic and real RDF datasets containing millions of triples. These experiments confirm that our SPARQL-UO query evaluation method is orders of magnitude more efficient than existing work.

REFERENCES

[1] [n.d.]. DBPedia. https://wiki.dbpedia.org/

[2] [n.d.]. LUBM. http://swat.cse.lehigh.edu/projects/lubm/

[3] [n.d.]. Virtuoso. https://virtuoso.openlinksw.com/

[4] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. 2009. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.* 18, 2 (2009), 385–406.

[5] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*. ACM, 411–422.

[6] Mohammed Al-Kateb, Paul Sinclair, Alain Crolotte, Lu Ma, Grace Au, and Sanjay Nair. 2017. Optimizing UNION ALL Join Queries in Teradata. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1209–1212.

[7] Medha Atre. 2015. *Left Bit Right*: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *SIGMOD*. ACM, 1793–1808.

[8] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *WWW*. ACM, 41–50.

[9] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40.

[10] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.

[11] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *SIGMOD*. ACM, 121–132.

[12] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. 2009. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering* 68, 10 (2009), 973–1000.

[13] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *The Semantic Web – ISWC 2019*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer International Publishing, Cham, 258–275.

[14] Vijay Ingalalli, Dino Ienco, Pascal Poncelet, and Serena Villata. 2016. Querying RDF Data Using A Multigraph-based Approach. In *EDBT*. OpenProceedings.org.

[15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.

[16] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. 2013. Static Analysis and Optimization of Semantic Web Queries. *ACM Trans. Database Syst.* 38, 4, Article 25 (dec 2013), 45 pages. https://doi.org/10.1145/2500130

[17] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704.

[18] Thomas Neumann and Gerhard Weikum. 2009. The RDF-3X engine for scalable management of RDF data. *VLDB Journal* 19, 1 (Sept. 2009), 91–113.

[19] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[20] Eric Prud'hommeaux and Alexandre Bertails. 2008. A Mapping of SPARQL Onto Conventional SQL. https://www.w3.org/2008/07/MappingRules/StemMapping#sqlOpt

[21] Jun Rao, Hamid Pirahesh, and Calisto Zuzarte. 2004. Canonical Abstraction for Outerjoin Optimization. In *SIGMOD*. ACM, 671–682.

[22] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1043–1052. https://doi.org/10.1145/3178876.3186003

[23] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. 2003. Efficient RDF Storage and Retrieval in Jena2. In *The first International Workshop on Semantic Web and Databases*. 131–150.

[24] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a Fast and Compact System for Large Scale RDF Data. *Proc. VLDB Endow.* 6, 7 (2013), 517–528.

[25] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. 2011. gStore: Answering SPARQL Queries via Subgraph Matching. *Proc. VLDB Endow.* 4, 8 (2011), 482–493.

APPENDIX

A QUERIES USED IN EXPERIMENTS

A.1 Queries on LUBM

Listing 1: Prefixes of LUBM Queries

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Listing 2: q1.1 on LUBM

```
SELECT * WHERE {
{ ?v2 ub:headOf ?v1. } UNION { ?v2 ub:worksFor ?v1. }
?v2 ub:undergraduateDegreeFrom ?v3.
?v4 ub:doctoralDegreeFrom ?v3.
?v5 ub:publicationAuthor ?v2.
{ ?v6 ub:headOf ?v1. } UNION { ?v6 ub:worksFor ?v1. }
{ ?v2 ub:headOf ?v7. } UNION { ?v2 ub:worksFor ?v7. }
<http://www.Department0.University0.edu/UndergraduateStudent91> ub
      :memberOf ?v1.
?v7 ub:name ?v8. }
```

Listing 3: q1.2 on LUBM

```
SELECT * WHERE {
?v3 ub:emailAddress "UndergraduateStudent91@Department0.
      University0.edu" .
?v2 ub:emailAddress ?v1 .
OPTIONAL { ?v2 ub:teacherOf ?v4. ?v3 ub:takesCourse ?v4 . } }
```

Listing 4: q1.3 on LUBM

```
SELECT * WHERE {
<http://www.Department1.University0.edu/UndergraduateStudent363>
      ub:takesCourse ?v1.
OPTIONAL { ?v2 ub:teachingAssistantOf ?v1.
OPTIONAL { ?v2 ub:memberOf ?v3.
?v4 ub:subOrganizationOf ?v3.
?v4 ub:subOrganizationOf ?v5.
?v4 rdf:type ?v6.
OPTIONAL { ?v5 ub:subOrganizationOf ?v7. } } } }
```

Listing 5: q1.4 on LUBM

```
SELECT * WHERE {
?v1      ub:emailAddress "UndergraduateStudent309@Department12.
      University0.edu".
OPTIONAL{ ?v1 ub:memberOf ?v2. ?v2 ub:name ?v3.
OPTIONAL{?v5 ub:publicationAuthor ?v4. ?v4 ub:worksFor ?v2.
OPTIONAL{ ?v6 ub:publicationAuthor ?v4. } } } }
```

Listing 6: q1.5 on LUBM

```
SELECT * WHERE {
{ ?v2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v3. }
UNION
{ ?v2 ub:name ?v4. }
<http://www.Department0.University0.edu/UndergraduateStudent356>
      ub:memberOf     ?v1.
?v2      ub:worksFor      ?v1.
OPTIONAL{ ?v5 ub:advisor ?v2.
OPTIONAL{ ?v5 ub:teachingAssistantOf ?v6.} }
OPTIONAL{ ?v7 ub:advisor ?v2. } }
```

Listing 7: q1.6 on LUBM

```
SELECT * WHERE {
?v4 ub:headOf ?v1.
<http://www.Department1.University0.edu/UndergraduateStudent256>
      ub:memberOf ?v1.
?v3 ub:subOrganizationOf ?v5.
{ ?v2 ub:worksFor ?v1. } UNION { ?v2 ub:headOf ?v1. }
{ ?v2 ub:worksFor ?v3. } UNION { ?v2 ub:headOf ?v3. }
OPTIONAL { ?v6 ub:publicationAuthor ?v2. }
OPTIONAL { { ?v7 ub:headOf ?v1. } UNION { ?v7 ub:worksFor ?v1. } }
       }
```

A.2 Queries on DBpedia

Listing 8: Prefixes of DBpedia Queries

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
PREFIX purl:   <http://purl.org/dc/terms/>
PREFIX skos:   <http://www.w3.org/2004/02/skos/core#>
PREFIX nsprov:   <http://www.w3.org/ns/prov#>
PREFIX owl:   <http://www.w3.org/2002/07/owl#>
PREFIX dbo:   <http://dbpedia.org/ontology/>
PREFIX dbr:   <http://dbpedia.org/resource/>
PREFIX dbp:    <http://dbpedia.org/property/>
```

Listing 9: q1.1 on DBpedia

```
SELECT * WHERE {
{ ?v3 rdfs:label ?v7. } UNION { ?v3 foaf:name ?v7. }
{ ?v1 purl:subject ?v3. } UNION { ?v3 skos:subject ?v1. }
?v3 rdfs:label ?v4.
?v5 nsprov:wasDerivedFrom ?v2.
?v1 owl:sameAs ?v6.
?v1 dbo:wikiPageWikiLink dbr:Economic_system.
?v1 nsprov:wasDerivedFrom ?v2. }
```

Listing 10: q1.2 on DBpedia

```
SELECT * WHERE {
{ ?v3 purl:subject ?v5. OPTIONAL{ ?v5 rdfs:label ?v6 } }
UNION
{ ?v5 skos:subject ?v3. OPTIONAL{ ?v5 foaf:name ?v6 } }
?v1 dbo:wikiPageWikiLink dbr:Economic_system .
?v1 nsprov:wasDerivedFrom ?v2 .
?v3 dbo:wikiPageWikiLink ?v4 .
?v3 nsprov:wasDerivedFrom ?v2 . }
```

Listing 11: q1.3 on DBpedia

```
SELECT * WHERE {
dbr:Air_masses foaf:isPrimaryTopicOf ?v1.
?v2 foaf:isPrimaryTopicOf ?v1.
OPTIONAL {
?v2 dbo:wikiPageRedirects ?v3. ?v4 foaf:primaryTopic ?v2.
OPTIONAL{
?v5 dbo:wikiPageWikiLink ?v3.
OPTIONAL{ ?v6 dbo:wikiPageRedirects ?v5.
OPTIONAL{ ?v6 dbo:wikiPageWikiLink ?v7. } } } } }
```

Listing 12: q1.4 on DBpedia

```
SELECT * WHERE {
dbr:Functional_neuroimaging purl:subject ?v1.
OPTIONAL{
?v1 owl:sameAs ?v2. ?v1 rdf:type ?v3. ?v4 owl:sameAs ?v2. ?v5 skos
      :related ?v4.
OPTIONAL{ ?v6 skos:related ?v4. }
OPTIONAL{
{ ?v7 purl:subject ?v1. } UNION { ?v1 skos:subject ?v7. }
OPTIONAL{
{ ?v7 purl:subject ?v8. } UNION { ?v8 skos:subject ?v7. } } } } }
```

Listing 13: q1.5 on DBpedia

```
SELECT * WHERE {
{ ?v2 purl:subject ?v3. } UNION { ?v2 dbo:wikiPageWikiLink ?v4. }
?v1 dbo:wikiPageWikiLink dbr:Abdul_Rahim_Wardak.
?v2 dbo:wikiPageWikiLink ?v1.
OPTIONAL{ ?v5 owl:sameAs ?v2.
OPTIONAL{ ?v5 dbo:wikiPageLength ?v6. } }
OPTIONAL{ ?v2 skos:prefLabel ?v7 . } }
```

Listing 14: q1.6 on DBpedia

```
SELECT * WHERE {
{ ?v2 foaf:primaryTopic ?v1. } UNION { ?v1 foaf:isPrimaryTopicOf ?
      v2. }
{ ?v2 foaf:primaryTopic ?v3. } UNION { ?v3 foaf:isPrimaryTopicOf ?
      v2. }
?v1 dbo:wikiPageWikiLink dbr:Category:Cell_biology.
?v3 dbo:wikiPageWikiLink ?v1.
```

```
OPTIONAL{
{ ?v2 foaf:primaryTopic ?v4. } UNION { ?v4 foaf:isPrimaryTopicOf ?
    v2. } }
OPTIONAL{ ?v5 dbo:phylum ?v3. ?v6 dbo:phylum ?v3.
OPTIONAL{
{ ?v7 foaf:primaryTopic ?v5. } UNION { ?v5 foaf:isPrimaryTopicOf ?
    v7. } } } }
```

B PROOFS OF THEOREMS

B.1 Proof of Theorem 1

Proof. By Definition 7 and the definitions of the operators on bags, we have

$$
\begin{aligned}
&[[P_1 \text{ AND } (P_2 \text{ UNION } P_3)]]_D \\
&= [[P_1]]_D \bowtie [[P_2 \text{ UNION } P_3]]_D \\
&= [[P_1]]_D \bowtie ([[P_2]]_D \cup_{bag} [[P_3]]_D) \\
&= ([[P_1]]_D \bowtie [[P_2]]_D) \cup_{bag} ([[P_1]]_D \bowtie [[P_3]]_D) \\
&= [[P_1 \text{ AND } P_2]]_D \cup_{bag} [[P_1 \text{ AND } P_3]]_D \\
&= [[(P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ AND } P_3)]]_D.
\end{aligned}
$$

□

B.2 Proof of Theorem 2

Proof. Similarly, we have

$$
\begin{aligned}
&[[P_1 \text{ OPTIONAL } (P_1 \text{ AND } P_2)]]_D \\
&= ([[P_1]]_D \bowtie [[P_1 \text{ AND } P_2]]_D) \cup_{bag} ([[P_1]]_D \setminus [[P_1 \text{ AND } P_2]]_D) \\
&= ([[P_1]]_D \bowtie ([[P_1]]_D \bowtie [[P_2]]_D)) \\
&\quad \cup_{bag} ([[P_1]]_D \setminus ([[P_1]]_D \bowtie [[P_2]]_D)) \\
&= ([[P_1]]_D \bowtie [[P_2]]_D) \cup_{bag} ([[P_1]]_D \setminus [[P_2]]_D) \\
&= [[P_1 \text{ OPTIONAL } P_2]]_D.
\end{aligned}
$$

□