Efficient Execution of SPARQL Queries with OPTIONAL and UNION Expressions

Yue Pang
Peking University
Beijing, China
michelle.py@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@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 planning challenges. In this paper, we propose techniques for optimizing 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.**

***Index Terms*—graph database, graph query, query optimization, OPTIONAL expressions, UNION expressions**

I. 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 focus on basic graph pattern (BGP) execution, 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.

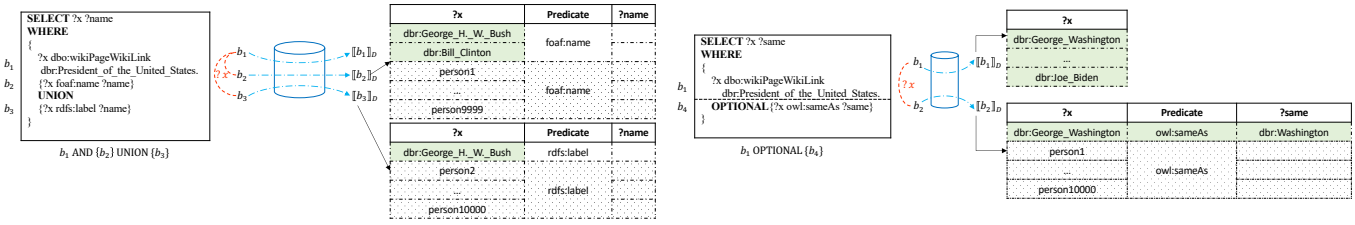
`UNION` and `OPTIONAL` expressions are essential in SPARQL grammar. Firstly, RDF datasets are semi-structured and do not enforce a schema. The `UNION` operator is crucial in this case since it merges diversely expressed information. For example, in DBpedia [1], an open-domain knowledge graph extracted 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 (Fig. 1(a)). Secondly, RDF datasets are commonly incomplete. Specifically, an entity may lack some attributes or relationships that most other entities of the same type have. In this case, the OPTIONAL operator is useful, since it allows attaching some attributes or relations as optional information. For example, the `OPTIONAL` query in Fig. 1(b) fetches all the presidents of the United States, along with other references to them that are not on the same Wikipedia page. Since not every president has multiple references in the database, the triple with the predicate `owl:sameAs` is enclosed in an `OPTIONAL` expression.

`UNION` and `OPTIONAL` expressions are widely used in real-world SPARQL workloads and are part of the SPARQL 1.1 specification. Recent empirical studies [2] 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 query plan representation for the evaluation plan of SPARQL-UO queries, which is a tree structure with BGPs as leaves and graph pattern operators as internal nodes. 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, which is backed by relational algebra (RA). The BE-tree serves as an easily constructible and comprehensible intermediate form between a SPARQL query and its RA-based 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, which are one-to-one mapped from the BE-trees to the RA-based plans for execution. Since there are many different ways to transform an RA-based plan, we devise a cost model that accounts for the cost of evaluating both BGPs and these operators, and choose the transformation that reduces the most cost. Because of the vast space of possible transformations, we propose a greedy strategy to determine

This work was supported by The National Key Research and Development Program of China under grant 2023YFB4502303. Özsu's research was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada. Lei Zou is the corresponding author of this paper.



(a) UNION Clause

(b) OPTIONAL Clause

Fig. 1: An Example Query with a UNION and OPTIONAL Clause.

the transformations step-by-step. The transformed plan is then evaluated by the BGP-based scheme, enhanced by the query-time optimization, *candidate pruning*, which prunes the search space of BGP evaluation on the fly.

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 RA-backed 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.

The rest of the paper is organized as follows. A brief review of related work is given in Sec. II. Sec. III provides the necessary preliminary information. Sec. IV presents the BE-tree plan representation, the transformations, and the query evaluation scheme. The cost-driven planning algorithm is proposed in Sec. V, and the query-time optimization is presented in Sec. VI. We experimentally evaluate our method in Sec. VII and conclude the paper in Sec. VIII.

II. Related Work

Although SPARQL query optimization has been extensively studied, most of the focus has been on evaluating BGPs [3], including graph-based approaches and relational approaches. Graph-based approaches include works on effective index strategies (e.g., gStore [4]) and join order optimization (e.g.,WCOJ [5]). In contrast, relational approaches rely on a relational DBMS and consider RDF graphs as three-column tables or other complex table organizations [6], [7]. Processing SPARQL queries is then mapped to its relational counterparts, as done in Apache Jena [8] and Virtuoso [9]. Relational BGP optimization approaches focus primarily on efficient data organization (e.g., property table [8], vertical partitioning [10] and single table exhaustive indexing [11]). However, these BGP optimization techniques cannot optimize UNION and OPTIONAL, since the semantics of these operators are fundamentally different from joins. 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.

Works on optimizing SPARQL queries with UNION and OPTIONAL operators are quite scarce. LBR [3] 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 [12], and proposes a semijoin strategy to prune the candidate results. However, it follows an execution strategy of two-pass semi-join scans on the join graph, which introduces additional overhead during query execution. In this paper, we propose a more comprehensive approach that handles both UNION and OPTIONAL. Experiments also demonstrate that our techniques significantly outperform LBR on OPTIONAL queries. Works on ontology-based query reformulation [13], [14], which target SPARQL queries with UNION, bear a conceptual similarity to ours in that they consider a space of possible rewritten queries and select the one with the lowest cost for execution. However, the rewriting rules and cost model that they use are different from ours, and they do not consider OPTIONAL.

There are a number of theoretical works on optimizing queries with OPTIONAL [15], [16], which propose semantics-preserving rewriting rules for such queries. Our method, which operates on two particular rules, is parallel to these works and may be extended to support them in a cost-based manner in the future.

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 subselects in SQL, so our techniques cannot be applied without major adaptations [17], [18]. We are aware of a recent demonstration [19] 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. However, no description of the employed cost model is provided, which renders experimental comparison impossible. Some works on relational algebra have focused on outerjoin optimization [20], which can be applied to our setting. However, to our knowledge, there has not been similar works that treat rewritings in a systematic

cost-based manner and outperform well-established baseline RDF systems experimentally.

III. Preliminaries

RDF Dataset.

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 $t = \langle subject, property, object \rangle \in (I \cup B) \times I \times (I \cup B \cup L)$. Tab. I is an example RDF dataset with seven triples.

SPARQL Query—Syntax. In the following, we present the query dialect that we target in this paper, which is a proper subset of SPARQL 1.1. Assume 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. 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 (Fig. 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 needs to be matched over the RDF dataset, which may consists of many other types of graph patterns, 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*.

Basic graph patterns (BGPs) are sets of triple patterns. In this paper, we focus on connected BGPs, i.e., BGPs that represent connected query subgraphs. For brevity, all mentions of BGPs refer to connected BGPs henceforth. To give a formal definition of BGPs, we 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. We denote the coalesce operation as $+$, e.g., the BGP formed from coalescing b_1 and b_2 is denoted as $b_1 + b_2$.

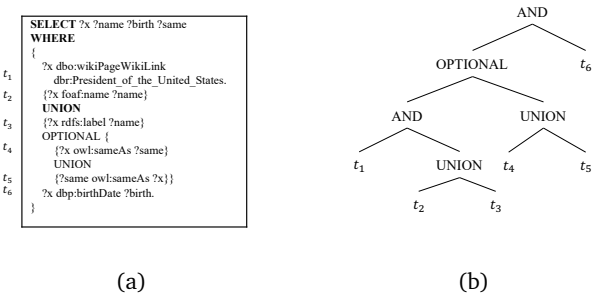
Definition 5 (Basic Graph Pattern (BGP)): A BGP is recursively defined as follows:

1) A triple pattern t is a BGP;
2) if P_1 and P_2 are coalescable BGPs, $P_1 + P_2$ 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;



(a) (b)

Fig. 2: An example SPARQL query and its tree expression

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.

Fig. 2 is an example SPARQL query with six triple patterns ($t_{1..6}$) and `UNION` and `OPTIONAL` operators.

SPARQL Query—Semantics. A graph pattern is essentially an expression that contains triple patterns and the left-associative operators `AND`, `UNION` and `OPTIONAL`, which accept graph patterns as their operands. `AND` and `OPTIONAL` are binary operators, while `UNION` can have two or more operands. Such an expression can be equivalently represented by a tree, where each leaf node represents a triple pattern and each internal node represents an operator. The positioning of each node in the tree is determined by the priority of the operators: `AND` = `OPTIONAL` $\prec$ `UNION`. Curly braces override the standard order of operations and define the scope of the operands. Fig. 2b shows such a tree expression of the query in Fig. 2a.

A graph pattern P is matched on an RDF dataset D (denoted by $[\![P]\!]_D$) to produce a bag (i.e., multi-set) of mappings $\{\mu_1, \mu_2, \ ... \ , \ \mu_n\}$, which 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:

TABLE I: 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-

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) $\Omega_1 \cup_{bag} \Omega_2 = \{\mu_1 \mid \mu_1 \in \Omega_1\} \bigcup_{bag} \{\mu_2 \mid \mu_2 \in \Omega_2\}$.
3) $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 \ : \ \mu_1 \nsim \mu_2\}$.
4) $\Omega_1 \leftmodels \Omega_2 = (\Omega_1 \bowtie \Omega_1) \bigcup_{bag} (\Omega_1 \setminus \Omega_1)$

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

We define how to evaluate graph patterns according to W3C's standard semantics for SPARQL as follows.

Definition 7 (Evaluation of Graph Patterns on an RDF dataset): The evaluation of graph patterns P on an RDF dataset D, denoted as $[\![P]\!]_D$, is recursively defined as follows, where the order of the operators follow that in the tree expression:

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, $[\![P]\!]_D = [\![P_1]\!]_D \bigcup_{bag} [\![P_2]\!]_D$.
5) If $P = P_1$ OPTIONAL P_2, $[\![P]\!]_D = [\![P_1]\!]_D \leftmodels [\![P_2]\!]_D$.

IV. Plan Representation: BGP-based Evaluation Tree

The most straightforward approach for evaluating a graph pattern P is to employ a bottom-up strategy on the tree representation. In each step, we either evaluate a triple pattern, or perform an operator (AND, UNION, or OPTIONAL). This tree-based evaluation strategy strictly follows the SPARQL semantics discussed in Sec. III, but has inherent performance limitations due to the large number of intermediate results generated for each triple pattern at the leaf nodes. To illustrate this, consider the simple SPARQL query in Fig. 3. Note that the outermost group graph pattern of this query only contains a BGP. Following the tree-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 have their birth dates as an attribute.

It is 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 [11], SW-store [6], gStore [4] and Jena [8]. Thus, in our approach, we design a BGP-based Evaluation Tree (BE-tree) to represent the intermediate form between SPARQL queries and their RA-based query plans.

A. BE-Tree & RA-Based Plan Structure

The BE-tree is a conceptually simple syntax parse tree of the SPARQL dialect considered in this paper, which is straightforward to construct and intuitive to understand. We use the BE-tree as an intermediate form between a SPARQL query and its RA-based plan.

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 (Def. 6) representing the query Q;
- An internal node of $T(Q)$ can be one of {UNION, OPTIONAL, group graph pattern} nodes:
  - 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 OPTIONAL-left and OPTIONAL-right graph patterns. It has exactly one child node: the OPTIONAL-right graph pattern, which is a group graph pattern node;
- Each leaf node of $T(Q)$ is a BGP node (Def. 5).

After constructing a BE-tree from parsing a SPARQL query, all coalescable sibling triple patterns and BGPs whose parent is a group graph pattern node are coalesced until no further coalescing can be performed, forming maximal BGPs. When the triple patterns of siblings or BGPs are on different sides of an OPTIONAL node, they can only be coalesced when the OPTIONAL is well-designed [21].

As a concrete example, the BE-tree of the query in Fig. 2a is given in Fig. 4. 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.

The RA-based plan, which dictates how the query is executed, is defined as follows.

Definition 9 (RA-Based Plan): Given a group graph pattern Q, its corresponding RA-based plan Plan(Q) is defined as follows:

- Each internal node of Plan(Q), including the root node, is an AND, OPTIONAL, or UNION node;
- Each leaf node of Plan(Q) is a BGP node (Def. 5).

Compared with the tree expression in Fig. 2b, the RA-based plan has the same types of internal nodes, but has BGPs as leaf nodes instead of triple patterns. For convenience, we denote the internal nodes as the graph pattern

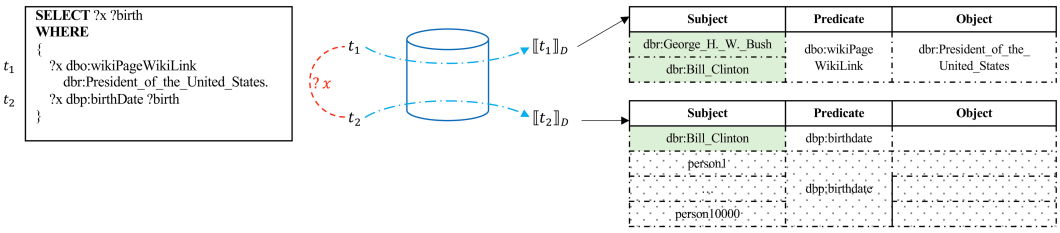


Fig. 3: Inefficiency of binary-tree-based query evaluation

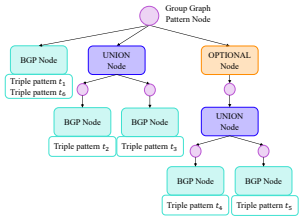


Fig. 4: Example BE-tree.

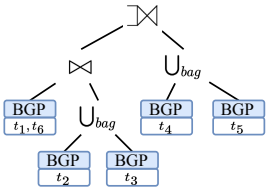


Fig. 5: Example RA-based plan.

operators' respective RA operators, i.e., AND as $\bowtie$, UNION as $\bigcup_{bag}$, and OPTIONAL as $⟕$. For example, Fig. 5 shows the RA-based plan converted from the BE-tree Fig. 4. To evaluate a SPARQL query, its RA-based plan is traversed in a depth-first manner, during which the BGPs are evaluated and their results are combined according to the operators' semantics, as shown in Alg. 1.

Algorithm 1: BGP-based query evaluation

Input: RDF dataset D, RA-based plan Plan(Q)
Output: $[\![Q]\!]_D$

```
1  BGPBasedEvaluation(D, Plan(Q).root);
2  Function BGPBasedEvaluation(D, v):
3      if v is of type ⋈ then
4          r ← BGPBasedEvaluation(D, v.child[0]);
5          r ← r ⋈ BGPBasedEvaluation(D, v.child[1]);
6          return r;
7      else if v is of type ⋃_bag then
8          r ← BGPBasedEvaluation(D, v.child[0]);
9          for i in 1, 2, ··· , num(child) do
10             r ← r ⋃_bag BGPBasedEvaluation(D, v.child[i]);
11         return r;
12     else if v is of type ⟕ then
13         r ← BGPBasedEvaluation(D, v.child[0]);
14         r ← r ⟕ BGPBasedEvaluation(D, v.child[1]);
15         return r;
16     else if v is a BGP then
17         return EvaluateBGP(D, e_i)
```

It is straightforward to convert from a BE-tree to an RA-based plan. Basically, we perform a depth-first search on the BE-tree, prioritizing each node's rightmost children. The RA-based plan is grown from right to left as the BE-tree nodes are visited. A group graph pattern node in a BE-tree will be mapped to one or more $\bowtie$ nodes in an RA-based plan.

B. BE-Tree & RA-Based Plan Transformations

In the previous subsection, we invoke the BGP-based evaluation procedure (Alg. 1) on the RA-based plan directly constructed from BE-tree of 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*.

1) Goals: Our aim is to transform the original RA-based plan so that the resulting RA-based plan has the following properties:

- *Validity:* the resulting RA-based plan should maintain the previously defined tree structure and have the same node types.
- *Efficiency:* the evaluation of the resulting RA-based plan should be more efficient than the original RA-based plan.

2) Semantics-Preserving Transformations: In order to optimize for query execution efficiency while maintaining correctness, we need to leverage the inherent semantic equivalences regarding the UNION and OPTIONAL, according to the following theorems backed by RA. We first introduce these transformations on the BE-tree, then map them to RA-based plans.

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

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

Proof 1:

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

Note that Thm. 1 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

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

Proof 2:

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

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, defined as follows.

Definition 10 (Merge transformation): A merge transformation is performed on a graph pattern P_1's node 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

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

The merge transformation consists of the following steps:

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 11 (Inject transformation): An inject transformation is performed on a graph pattern P_1's node and one of its sibling OPTIONAL nodes to its right, the child node of which represents the group graph pattern P_2, when

1) P_1 is a BGP node;
2) P_2 has a BGP child node that is coalescable with P_1.

The inject transformation 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.

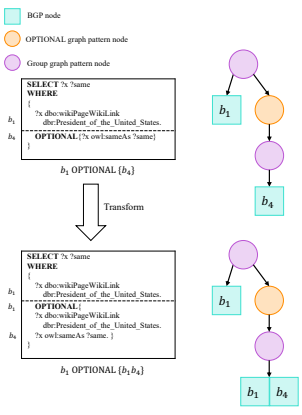


Fig. 6: Example *Inject* Transformation.

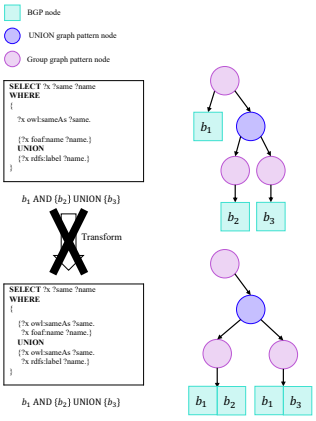


Fig. 7: Example *Merge* Transformation.

Example. In Fig. 6 and 7, we give examples of these transformations' effects on SPARQL queries targeting DBpedia.

In Fig. 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, which 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. After 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 explains why the transformations are only defined on coalescable BGPs. If no coalescing happens, the repeated evaluation of the merged or injected BGP will incur overhead.

However, not all available transformations can help improve efficiency, which necessitates enumeration and cost-based selection. Fig. 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.

When mapped onto the RA-based plan, the scope of a transformation is not as obvious as sibling nodes in the BE-tree. To search for UNIONs and OPTIONALs viable for transforming with a BGP, we need to search the BGP's ancestors until finding the $\bowtie$ operator closest to the root without passing through any $\bigcup_{bag}$ node. All the $\bigcup_{bag}$'s and $\bowtie$'s reachable from this $\bowtie$ through only $\bowtie$'s are exactly this BGP's siblings in the BE-tree and can be considered for transformation. We denote the set of $\bigcup_{bag}$'s and $\bowtie$'s that can be considered by a BGP b as scope(b).

V. Cost-Driven Plan Selection

In this section, we introduce the cost-driven approach selecting the expectedly most efficient transformations to be performed on an RA-based plan.

Our cost-driven approach operates on a higher level than BGP evaluation, but still relies on estimations of the evaluation costs and result sizes of BGPs, which are obtainable from the underlying BGP evaluation engine [22].

A. Cost Models

1) Cost Model for SPARQL-UO: SPARQL-UO query execution cost is made up of two main components: the cost of evaluating BGPs and of computing graph pattern operators: $\bowtie$, $\bigcup_{bag}$, and $\bowtie$. 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. In the following, we discuss how to estimate the Δ-cost.

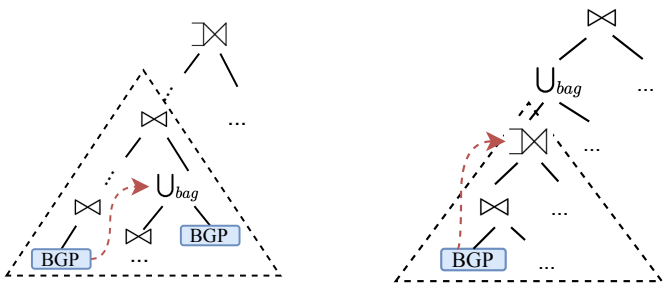


Fig. 8: Merge & Inject Transformations on an RA-Based Plan.

a) Merge transformation: Consider the RA-based plan in Fig. 8a. The merge transformation shown as the red arrow will cause the following structural changes in the RA-based plan: the outer BGP will be coalesced with the $\bigcup_{bag}$'s child BGPs, and if some children of the $\bigcup_{bag}$ are not BGPs or are not coalescable with the outer BGP, new $\bowtie$'s will join them with the outer BGP. The transformation will also affect the cost of the $\bigcup_{bag}$, since the operands' result sizes will change. The cost of all the ancestor operators of the $\bigcup_{bag}$ will also change if the result size of the $\bigcup_{bag}$ changes, but we do not account for them for efficiency. When estimating the Δ-cost of a merge operation, we only need to account for the cost of the affected parts in the RA-based plan before and after the operation.

Suppose a merge transformation concerns the outer BGP b and the $\bigcup_{bag}$ node u; the set of child nodes of u in the RA-based plan is denoted as $child(u)$. We can express the aforementioned affected parts' cost before and after the merge by the following formulas:

$$cost_{\text{BGP}}(b,u) = cost(b) + \sum_{b_i \in \text{child}(u)} cost(b_i) \tag{1}$$

$$cost_{\text{algebra}}(b,u) = cost(u) \tag{2}$$

$$cost_{\text{BGP}}(\text{merge}(b,u)) = \sum_{b_i \in \text{child}(u)} cost(b + b_i) \tag{3}$$

$$cost_{\text{algebra}}(\text{merge}(b,u)) = cost(u') + \sum_{p_i \in \text{child}(u)} cost(\bowtie(b,p_i)) \tag{4}$$

$$\begin{aligned}\Delta cost(b,u) = {} & cost_{\text{BGP}}(b,u) + cost_{\text{algebra}}(b,u) \\ & - cost_{\text{BGP}}(\text{merge}(b,u)) \\ & - cost_{\text{algebra}}(\text{merge}(b,u))\end{aligned} \tag{5}$$

where Eq. 1 and 3 represent the cost of the affected BGPs, i.e., the outer BGP b and the child BGPs of u that are coalescable with b, denoted as b_i, before and after the merge operation, since b is coalesced with each b_i during the merge; while Eq. 2 and 4 represent the cost of the affected algebraic operators, including u and the newly introduced $\bowtie$'s. Thus, the Δ-cost of this merge operation is the difference between the summed cost of the affected BGPs and algebraic operators before and after the operation.

The cost of the BGPs can be directly obtained from the BGP evaluation engine, while the cost of the algebraic operators are functions on the result sizes of their operands. These functions may differ with physical implementations. In our experiments, to fit the system we choose for implementation, the cost of $\bowtie$ is set as the product of its arguments' result sizes, and the cost of the $\bigcup_{bag}$ is set as the sum of its arguments' results sizes. A BGP node's result size estimate can be obtained from the 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 estimate the result size of any $\bowtie$ or $\rtimes$ as the product of their operands' result sizes, and the result size of $\bigcup_{bag}$ to be the sum of its operands' result sizes.

b) Inject transformation: An inject transformation such as the red arrow in Fig. 8b will cause a BGP in the subtree rooted at the right child of the $\rtimes$ node to coalesce with the outer BGP. In practice, we choose the first coalescable BGP as the target for injection for convenience. The transformation will also affect the cost of the $\rtimes$ as well as all the $\bowtie$'s on the path from the $\rtimes$ to the coalesced BGP.

Suppose an inject transformation concerns the outer BGP b and the $\rtimes$ node o due to the change in result sizes. The following formulas show the affected parts' cost before and after injecting:

$$cost_{\text{BGP}}(b,o) = cost(b_i) \tag{6}$$

$$cost_{\text{algebra}}(b,o) = cost(o) + \sum_{a_j \in p(o,b_i)} cost(a_j) \tag{7}$$

$$cost_{\text{BGP}}(\text{inject}(b,o)) = cost(b + b_i) \tag{8}$$

$$cost_{\text{algebra}}(\text{inject}(b,o)) = cost(o') + \sum_{a_j' \in p(o,b_i)} cost(a_j\prime) \tag{9}$$

$$\begin{aligned}\Delta cost(b,o) = {} & cost_{\text{BGP}}(b,o) + cost_{\text{algebra}}(b,o) \\ & - cost_{\text{BGP}}(\text{inject}(b,o)) \\ & - cost_{\text{algebra}}(\text{inject}(b,o))\end{aligned} \tag{10}$$

where b_i in Eq. 6 denotes the BGP in o's right child's subtree; after injecting, it coalesces with b as shown in Eq. 8. a_j denotes the $\bowtie$'s on the path $p(o,b_i)$ from the $\rtimes$ to the coalesced BGP. o' and $a_j\prime$ denotes the $\rtimes$ and $\bowtie$'s whose cost changes due to the change in cardinality.

The cost and cardinality of the BGPs and the algebraic operators involved are estimated in the same way as for merge transformations.

2) Cost Model for BGPs: We briefly introduce the BGP cost models employed by gStore, one of the systems we select for implementation, for completeness. The evaluation of BGPs consists of joins. Thus the cost of a BGP plan T is the sum of each join j's cost:

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

BGP evaluation in gStore uses the worst-case optimal (WCO) join. 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:

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

where $card(\{v_1, \cdots, v_{k-1}\})$ indicates the estimated cardinality, 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.

The cardinality estimation of query vertex sets starts from single triple patterns, whose exact cardinality can be obtained from the RDF store. Each time 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 the previous estimate:

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

B. Cost-Driven Transformation

In this subsection, we discuss RA-based plan transformation selection algorithms that leverage the cost model discussed above to decide on transformations for obtaining the most efficient query plan for execution.

1) Transforming a Single BGP: We first concentrate on the simpler case where only transformations concerning a BGP are considered, in which case the algorithm for transformation selection is shown in Alg. 2. The algorithm first finds the sets of $\bigcup_{bag}$'s and $\ltimes$'s in the RA-based plan that can perform a transformation with the given BGP (Line 2), the details of which will be introduced below. According to Thm. 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 the $\bigcup_{bag}$'s, but can be injected into multiple $\ltimes$'s. Therefore, in order to decide on a merge transformation, we need to look at all the viable $\bigcup_{bag}$'s and choose the transformation that incurs the lowest Δ-cost (Lines 3-11 and 14-17). On the other hand, *inject* transformations are mutually independent, so we scan over each $\ltimes$ and decide individually which ones should be transformed based on the Δ-cost (Lines 12-13 and 18-22).

We further explain how to find the sets of $\bigcup_{bag}$'s and $\ltimes$'s viable for transformation with a given BGP, i.e., the `FindRelevantUO` subroutine in Alg. 2. Assuming well-designed OPTIONALs [21] throughout the query, the set of viable $\ltimes$'s consists exactly of those joined with the given BGP via any number of $\bowtie$'s or $\ltimes$'s, while the set of viable $\bigcup_{bag}$'s consists exactly of those joined with the BGP via any number of $\bowtie$'s. Therefore, we search upwards for a root of a subtree in the RA-based plan via only $\bowtie$'s or only $\bowtie$'s and $\ltimes$'s (Lines 26-32) and put all the $\bigcup_{bag}$'s or $\ltimes$'s reachable via these operators from the root into the viable sets (Lines 33-34 and 36-43).

2) Handling an RA-Based Plan: Handling the entire RA-based plan, which often consists of multiple BGPs, is challenging because of the possible interdependence between transformations across BGPs. 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 RA-based plan.

In order to balance the time complexity and the efficiency of the transformed plan, we propose a greedy strategy to decide on the transformations on the entire RA-based plan (Alg. 3). Specifically, the algorithm first conducts a topological sort over the RA-based plan (Line 1). The BGPs are then considered in a reverse topological order for transformations (Lines 3-5). In this way, we ensure that the BGPs that are lower in the plan have been appropriately transformed before considering transforming the BGPs higher up in the plan, preventing expensive backtracking.

Algorithm 2: Single-BGP Transformation Selection

```
   Input: RDF dataset D, RA-based plan P, a BGP b
1  Function SingleBGPTransform(D,P,b):
2      U,O ← FindRelevantUO(b, P)
3      minUnionCost ← 0
4      targetUNION ← empty node
5      foreach u ∈ U do
6          minUnionCostCur ← DecideMerge (b, u)
7          if minUnionCostCur < minUnionCost then
8              minUnionCost ← minUnionCostCur
9              targetUNION ← u
10     if minUnionCost < 0 then
11         Perform merge on subBGPglobal and targetUNION
12     foreach o ∈ O do
13         DecideInject (b, o)
14 Function DecideMerge(b,u):
15     if constraints are violated then
16         return 0
17     return Δcost(b,u) (Eq. 5)
18 Function DecideInject(b,o):
19     if constraints are violated then
20         return
21     if Δcost(b,o) < 0 (Eq. 10) then
22         Perform inject on b and o
23 Function FindRelevantUO(b, P):
24     U ← ∅,O ← ∅, curNode ← b
25     localRootU ← empty node, localRootO ← empty node
26     while curNode is not P's root do
27         if curNode.parent is ⋃bag or ⋉ and localRootU is an empty node
             then
28             localRootU ← curNode
29         if curNode.parent is ⋃bag then
30             localRootO ← curNode
31             break
32         curNode ← curNode.parent
33     SearchSubtree(localRootU, P, U)
34     SearchSubtree(localRootO, P, O)
35     return U,O
36 Function SearchSubtree(v, P, S):
37     if v is an empty node then
38         v ← Proot
39     if The elements in S are of the same type as v then
40         S ← S ∪ {v}
41     else if v is ⋈ or those in S are ⋉ and v is ⋉ then
42         SearchSubtree(v.leftChild, P, S)
43         SearchSubtree(v.rightChild, P, S)
```

Algorithm 3: RA-Based Plan Transformation Selection

```
   Input: RDF dataset D, RA-based plan P
1  Function RAPlanTransform(D, P):
2      Do a topological sort on P's nodes
3      Let B be the BGPs in P sorted in descending topological order
4      foreach b ∈ B do
5          SingleBGPTransform(D, P, b)
```

VI. Query-Time Optimization: Candidate Pruning

In the previous section, we introduced how to select transformations on the RA-based plan based on cost estimations prior to execution. In this section, we present *candidate pruning*, a query-time optimization incorporated into Alg. 1 to enhance efficiency.

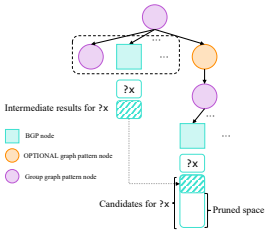


Fig. 9: Candidate pruning for `OPTIONAL`.

The basic idea of candidate pruning is also drawn from Thm. 1 and 2. The equivalence between the evaluation results implies that the results of the $\bigcup_{bag}$'s and $\bowtie$'s are constrained by those of the outer graph pattern in term of the common variables. Therefore, when a $\bigcup_{bag}$ or $\bowtie$ 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. For ease of explication, we show the mechanism of candidate pruning for an `OPTIONAL` query in Fig. 9, where 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 group graph pattern below the `OPTIONAL`, pruning redundant matchings of `?x` that will be materialized if the BGP is evaluated independently.

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. Therefore, we adopt an adaptive threshold on the candidate result size. The cost model for BGPs (Sec. V-A2) invoked as part of the transformation selection procedure (Alg. 2) 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 Sec. VII for the threshold setting in our experiments.)

To implement candidate pruning, we modify Alg. 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 $\bigcup_{bag}$ or $\bowtie$ (Lines 10 and 14);
- Pass *cand* as the third argument to `EvaluateBGP` (Line 17). Only when the size of *cand* is smaller than the threshold is it set as the candidate results of the BGP.

Tree transformations, which happens before query execution, and candidate pruning, which take effect during execution, 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. Meanwhile, while tree transformations are constrained to be performed one BGP at a time 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 viable for transforming with a `UNION` or `OPTIONAL`, performing transformations on them is equivalent to candidate pruning. In this case, tree transformation is skipped to evade the additional overhead.

VII. Experiments

To evaluate the effectiveness of our approach, we employ the BGP query engines of Jena[1], Blazegraph[2], and gStore to implement our BGP-based cost-aware SPARQL-UO evaluation strategy. 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 [23]) and real (DBpedia[4] [24] and YAGO [25]) RDF datasets, the statistics of which are listed in Table II. Our implementation and all the queries used in our experiments can be found in our GitHub repository[5]. We conduct experiments on a Linux server with an Intel Xeon Gold 6126 CPU @ 2.60GHz CPU and 256GB memory.

A. Verification of Optimizations

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

[1]https://github.com/apache/jena.
[2]https://github.com/blazegraph/database.
[3]Our implementation is available at https://github.com/SoftlySpoken/gStore-UO-opt.
[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://github.com/SoftlySpoken/gStore-UO-opt.

TABLE II: 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
YAGO	230,677,128	99,313,763	108	153,988,292

TABLE III: Query Statistics on LUBM

	Query	Type	$Count_{BGP}$	$Depth$	$\lvert[\![Q]\!]_D\rvert$
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 IV: Query Statistics on DBpedia

	Query	Type	$Count_{BGP}$	$Depth$	$\lvert[\![Q]\!]_D\rvert$
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

1) The baseline (abbreviated as `base`),i.e., the base versions of the systems, which invoke the BGP-based query evaluation method (Alg. 1);
2) Tree transformation (abbreviated as `TT`), which transforms the original RA-based plan by Alg. 3 and then invokes Alg. 1 on it;
3) Candidate pruning (abbreviated as `CP`), which invokes Alg. 1 augmented with candidate pruning (Sec. VI) on the original RA-based plan, 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`), 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, DBpedia, and YAGO, respectively, denoted as q1.1-1.6 in the following and given in Appendix A of [26]. 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 III and IV summarizes the statistics and the result sizes of the queries used in this subsection.

We report the query execution time and the time spent carrying out the tree transformations for `TT` and `full`. The performance of our approaches on LUBM, DBpedia, and YAGO is shown in Fig. 10. The absence of a bar indicates an out-of-memory or timeout error on the query. A query is considered timed-out if the execution time exceeds 2×10^6 milliseconds.

The trends of the results across gStore, Jena, and Blazegraph are 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, and on q1.3 and q1.6 on Blazegraph, where our optimizations' performance gains just do not compensate for their extra overhead), beating the baseline by 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.

When `TT` is effective. q1.1 on DBpedia (Listing 15, Appendix A of [26]) is a query on which `TT` is effective, but `CP` is not. In this query, two `UNION` clauses are given first (Lines 2-3), 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 5-8 with the `UNION` clause in Line 3 to accelerate query processing and reduce memory overhead, as evidenced in Fig. 10. q1.2 on LUBM and q1.2 on DBpedia

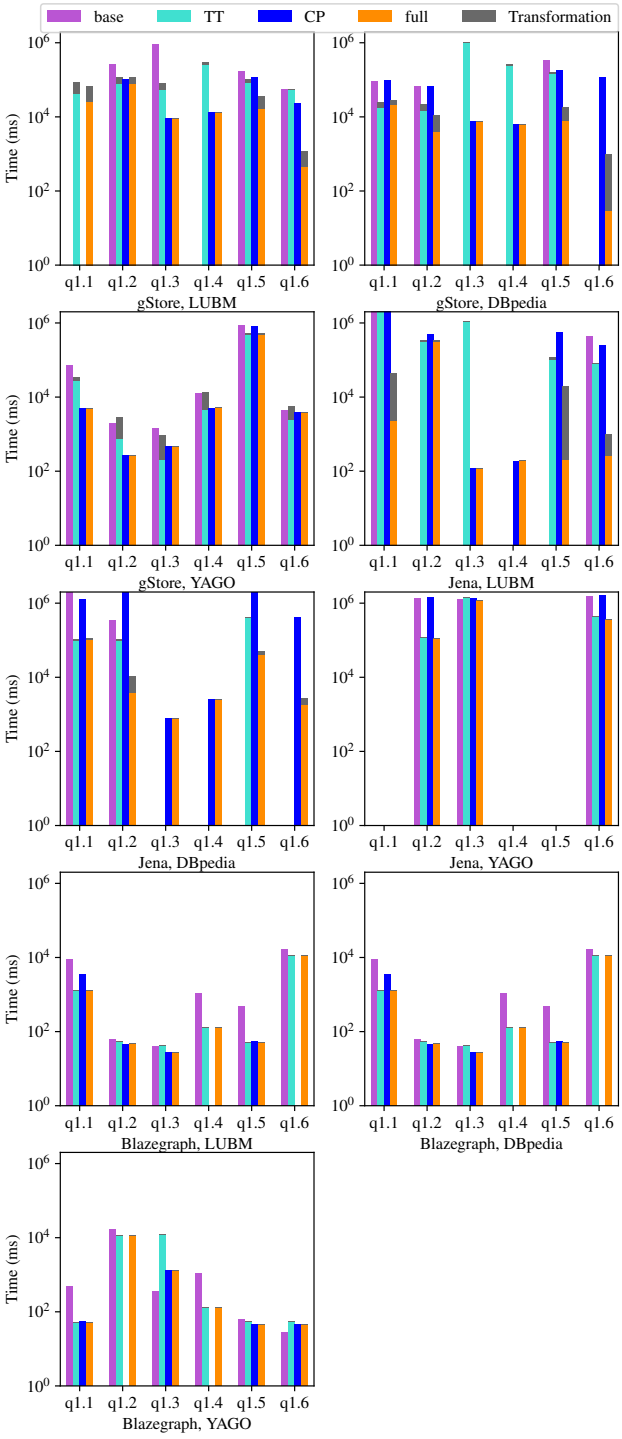


Fig. 10: Verification of optimizations.

also belong to this category. (Note that q1.2 on LUBM corresponds to the special case mentioned in Section VI, where there is only a BGP before an `OPTIONAL` clause, thus `TT` and `CP` have a similar effect.)

When `CP` is effective. q1.3 on LUBM (Listing 4, Appendix A of [26]) is a query on which `CP` is effective, but `TT` is not. In this query, the BGP in Line 2 has high selectivity, followed by nested `OPTIONAL`s with low-selectivity child BGPs. `TT` can inject the BGP into the outermost `OPTIONAL` but cannot reach the inner `OPTIONAL`s, 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, Appendix A of [26]) 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 2-3 has high selectivity, while the BGP in Line 4 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 5. Having executed the graph patterns up to Line 6, `CP` can accelerate the processing of upcoming `OPTIONAL`s. 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 Fig. 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.

B. Comparison with State-of-the-Art

The only work that considers SPARQL with `OPTIONAL` query optimization is LBR [3], with which we compare our `full` approach. We implement LBR in C++ and experiment on the queries provided in LBR [3] on LUBM and DBpedia, listed as q2.1-2.6 (Appendix A, [26]). The statistics of these queries are given in the second group in Tables III and IV. 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 has a high-selectivity BGP followed by an `OPTIONAL`.

The total response time of `full` and LBR are shown in Fig. 13. `full` is significantly faster than LBR on all queries,

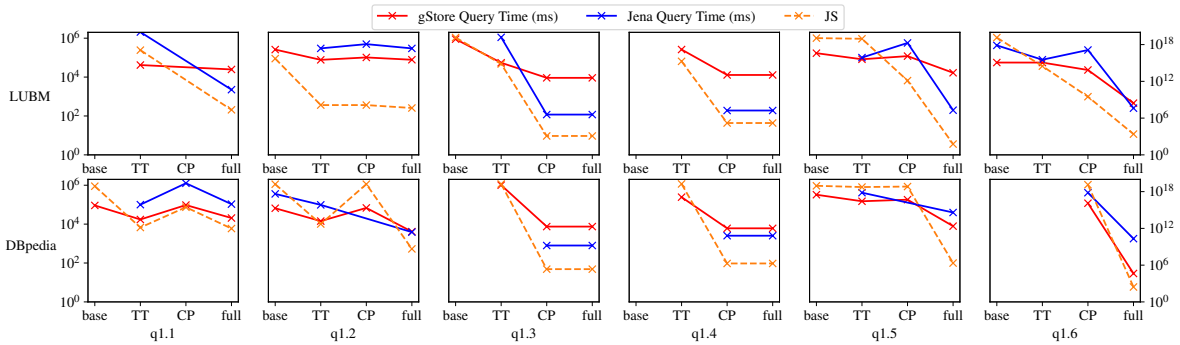


Fig. 11: The execution time and join space of queries.

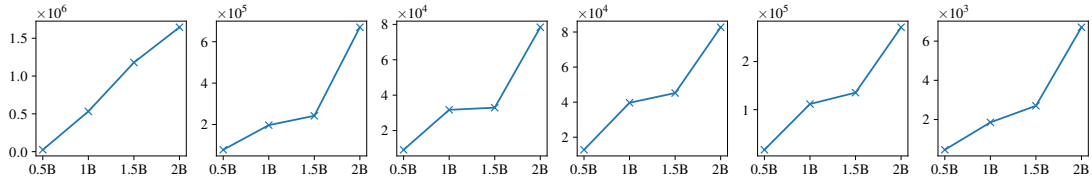


Fig. 12: Query execution time (ms) of `full` on LUBM datasets of different sizes ("B" is short for billion).

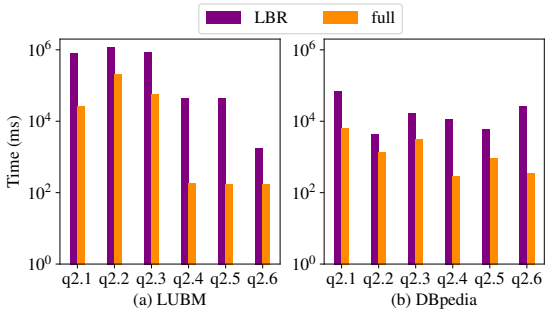


Fig. 13: Comparison with state-of-the-art on LUBM.

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 Sec. VI where tree transformation and candidate pruning are equivalent, hence only candidate pruning is 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 specially optimized for `OPTIONAL`.

C. 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 Fig. 12.

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 Tab. III, while those of q1.1-1.2 grow linearly.)

VIII. 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 billions of triples. These experiments confirm that our SPARQL-UO query evaluation method is orders of magnitude more efficient than existing work.

References

[1] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[2] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large sparql query logs," *The VLDB Journal*, vol. 29, no. 2, pp. 655–679, 2020.

[3] M. Atre, "*Left Bit Right*: For SPARQL join queries with OPTIONAL patterns (left-outer-joins)," in *SIGMOD*. ACM, 2015, pp. 1793–1808.

[4] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: Answering SPARQL queries via subgraph matching," *Proc. VLDB Endow.*, vol. 4, no. 8, pp. 482–493, 2011.

[5] A. Hogan, C. Riveros, C. Rojas, and A. Soto, "A worst-case optimal join algorithm for sparql," in *The Semantic Web – ISWC 2019*, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, Eds. Cham: Springer International Publishing, 2019, pp. 258–275.

[6] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*. ACM, 2007, pp. 411–422.

[7] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient RDF store over a relational database," in *SIGMOD*. ACM, 2013, pp. 121–132.

[8] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF storage and retrieval in jena2," in *The first International Workshop on Semantic Web and Databases*, 2003, pp. 131–150.

[9] "Virtuoso." [Online]. Available: https://virtuoso.openlinksw.com/

[10] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "Sw-store: a vertically partitioned DBMS for semantic web data management," *VLDB J.*, vol. 18, no. 2, pp. 385–406, 2009.

[11] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB Journal*, vol. 19, no. 1, pp. 91–113, Sep. 2009.

[12] J. Rao, H. Pirahesh, and C. Zuzarte, "Canonical abstraction for outerjoin optimization," in *SIGMOD*. ACM, 2004, pp. 671–682.

[13] D. Bursztyn, F. Goasdoué, and I. Manolescu, "Optimizing reformulation-based query answering in rdf," in *EDBT: 18th International Conference on Extending Database Technology*, 2015.

[14] ——, "Teaching an rdbms about ontological constraints," in *Very Large Data Bases*, 2016.

[15] M. Schmidt, M. Meier, and G. Lausen, "Foundations of sparql query optimization," in *Proceedings of the 13th international conference on database theory*, 2010, pp. 4–33.

[16] A. Letelier, J. Pérez, R. Pichler, and S. Skritek, "Static analysis and optimization of semantic web queries," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, pp. 1–45, 2013.

[17] E. Prud'hommeaux and A. Bertails, "A mapping of sparql onto conventional sql," Jul 2008. [Online]. Available: https://www.w3.org/2008/07/MappingRules/StemMapping#sqlOpt

[18] A. Chebotko, S. Lu, and F. Fotouhi, "Semantics preserving sparql-to-sql translation," *Data & Knowledge Engineering*, vol. 68, no. 10, pp. 973–1000, 2009.

[19] M. Al-Kateb, P. Sinclair, A. Crolotte, L. Ma, G. Au, and S. Nair, "Optimizing union all join queries in teradata," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 1209–1212.

[20] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen, "Using eels, a practical approach to outerjoin and antijoin reordering," in *Proceedings 17th International Conference on Data Engineering*, 2001, pp. 585–594.

[21] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Trans. Database Syst.*, vol. 34, no. 3, Sep. 2009. [Online]. Available: https://doi.org/10.1145/1567274.1567278

[22] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1692–1704, 2019.

[23] "Lubm." [Online]. Available: http://swat.cse.lehigh.edu/projects/lubm/

[24] "Dbpedia." [Online]. Available: https://wiki.dbpedia.org/

[25] F. M. Suchanek, M. Alam, T. Bonald, L. Chen, P.-H. Paris, and J. Soria, "Yago 4.5: A large and clean knowledge base with a rich taxonomy," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 131–140. [Online]. Available: https://doi.org/10.1145/3626772.3657876

[26] L. Zou, Y. Pang, M. T. Özsu, and J. Chen, "Efficient execution of sparql queries with optional and union expressions," 2023. [Online]. Available: https://arxiv.org/abs/2303.13844