Well-designed Query Optimization Based on Pattern Tree

Tenglong Ren[1][0009−0007−4215−4170], Hongshen Yu[1], Xiaowang Zhang[1,2⋆][0000−0002−3931−3886], Guangxi Ji[1], and Zhiyong Feng[1][0000−0001−8158−7453]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, 300354, China
[2] The Center of National Railway Intelligent Transportation System Engineering and Technology
{tenglongren,hs_yu,xiaowangzhang,guangxiji,zyfeng}@tju.edu.cn

Abstract. OPTIONAL, one of the most distinctive features of SPARQL, allows partial answers when complete answers are not available due to lack of information. Optional matching is computationally expensive and the query answering is PSPACE-complete. In this paper, we focus on well-designed queries. On the one hand, well-designed queries account for more than 70% of all queries in real queries. Well-designed queries, on the other hand, achieve much better computational properties since restricting optional matching in query answering makes the problem coNP-complete. The pattern tree represents the OPT nested hierarchical relationships between well-designed query patterns in a tree structure. We optimize well-designed queries based on the pattern tree. In our methods, we propose characteristic sets(CSs) merge-based algorithms to optimize queries within pattern tree nodes. We merge CSs based on predicate correlation while reducing null values in the data table. In addition, we propose to pass parent-to-child constraints on pattern trees to filter invalid data and reduce intermediate results to optimize inter-node queries. The extensive experiments on synthetic and real-world datasets show that our method outperforms the S2RDF and Prost by order of magnitude.

Keywords: Well-designed query · Well-designed pattern tree · Merging characteristic sets · Passing constraint · Query optimization.

1 Introduction

The W3C represents the knowledge graph model using the Resource Description Framework (RDF) and designates SPARQL as the standard query language for RDF. Owing to the flexibility of RDF, RDF data is extensively utilized across various domains, including science, business intelligence, and social networks[1]. Existing research on SPARQL query optimization primarily concentrates on the basic graph patterns(BGPs), which are implemented through graph matching and relational joins. Queries that extend beyond BGPs(Not limited to the AND

⋆ Corresponding author

operator) are often indirectly executed based on relational algebra. A BGP necessitates an exact match for the entire query. Otherwise, no answer will be returned, as illustrated in Figure 1(a). For instance, the query in Figure 1(a) requests personal information of all students attending University0, including their names, emails, teles (phone numbers), and so on. However, BGPs are incapable of capturing more complex queries. Given that RDF semantics adhere to an open-world assumption, RDF databases are inherently incomplete. Consequently, querying RDF with SPARQL typically yields partial answers.

```
SELECT ?people ?name ?email ?tele
WHERE{
   ?people :name ?name .
   ?people :school "University0" .
   ?people :email ?email .
   ?people :tele ?tele .
}
```

(a) BGP Query Q_1

```
SELECT ?people ?name ?email ?tele
WHERE{
   ?people :name ?name .
   ?people :school "University0" .
   OPTIONAL{ ?people :email ?email .}
   OPTIONAL{ ?people :tele ?tele .}
}
```

(b) Well-designed Query Q_2

Fig. 1: Query Example

The distinctive feature of SPARQL is the OPTIONAL(OPT) operator, which enables the retrieval of partial answers when complete answers are unavailable. Due to the unrestricted use of nested optional components, the evaluation of the OPT fragment of SPARQL is PSPACE-complete. Furthermore, the OPT operator is non-monotonic. Consider the following SPARQL graph pattern P: ?X <Wife> ?Y. OPT {?Y <Email> ?Z.} OPT {?X <Phone> ?W.}. Given two RDF datasets, G_1={(<Bob> <Wife> <Alice>)}, G_2={(<Bob> <Wife> <Alice>),(<Bob> <Phone> "7001")}. The answer to P in G_1 is $\{\mu_1\}$, where μ_1 is the mapping {?X $\rightarrow$ <Bob>, ?Y $\rightarrow$ <Alice>}. The answer to P in G_2 is $\{\mu_2\}$, where μ_2 is the mapping {?X $\rightarrow$ <Bob>, ?Y $\rightarrow$ <Alice>, ?W $\rightarrow$ "7001"}. Thus, we observe that $[\![P]\!]_{G_1} \nsubseteq [\![P]\!]_{G_2}$(since $\mu_1 \notin [\![P]\!]_{G_2}$). This demonstrates that P is not monotone as $G_1 \subseteq G_2$. Due to this non-monotonicity, traditional methods for handling BGP, which rely on monotonicity and graph matching, are no longer applicable. Instead, a SPARQL graph pattern P is considered weakly monotonic if for every pair of RDF graphs G_1,G_2 such that $G_1 \subseteq G_2$, it holds that $[\![P]\!]_{G_1} \sqsubseteq [\![P]\!]_{G_2}$. For example, the graph pattern Q_2(Figure 1(b)) is weakly monotonic. [3] demonstrates that weakly monotonicity is a more suitable framework for studying the OPT operator.

Pérez et al.[2] proposed well-designed patterns, which are SPARQL fragments that contain only the operators AND, FILTER, and OPT with restrictions on variables. Every well-designed graph pattern is weakly monotone[3]. Evaluation is coNP-complete for the fragment of SPARQL consisting of well-designed patterns[15]. Well-designed queries often appear in real query workloads. According to the analysis of LSQ[5] query logs by Han et al. [6], more than 70% of all queries are well-designed queries. Therefore, the well-designed query is uni-

versal and representative, and the research on well-designed query optimization is of great significance. Figure 1(b) is a well-designed query example.

Most research in well-designed query optimization has focused on theoretical studies. Letelier et al.[4] proposed the pattern tree to represent well-designed queries, termed Well-designed Pattern Trees(WDPTs). They proved that well-designed patterns can be equivalently converted to WDPTs and that the WDPT problem is coNP-complete. WDPTs intuitively capture the structure of well-designed query optimization. Each node corresponds to a BGP where the parent-child relationship represents the OPT operator relationship. Building on this, we propose CoPPT[3], which optimizes well-designed queries using the pattern tree. Key challenges in this optimization include:

(1) We analyzed the DBpedia query logs and counted all well-designed queries and found that 94.6% of the join variables between OPT operators are subject(similar to star queries[4]). When the data load is complex, joins between data tables during query execution are expensive. (2) Most existing work parses SPARQL queries as relational algebra trees, which do not well represent the structure of SPARQL queries. (3) Well-designed queries based on the pattern tree, the parent node has constraints on the child nodes, and the child nodes will produce more intermediate results due to ineffective constraints, which affects the query performance. To address the above challenges, the contributions of the paper are as follows:

- We propose a novel correlation-based CSs merging algorithm that decomposes BGPs on pattern tree nodes into star-shaped subqueries. This approach ensures predicates within each subquery are clustered in the same data source, effectively optimizing query processing at the pattern tree node level through predicate correlation analysis.
- We develop a cross-node query optimization method based on constraint passing on the pattern tree. The constraint characteristics of the OPT operator are utilized to transfer the constraints of the parent node to the child node on the pattern tree to reduce the intermediate results of query execution.
- Extensive experiments on synthetic and real-world RDF graphs have been conducted to verify the efficiency and scalability of our method. On average, the experimental results show that CoPPT outperforms the state-of-the-art method by an order of magnitude.

2 Related Work

The existing research on SPARQL query processing mainly focuses on optimizing BGPs[28][25][26], and the research on well-designed queries is more at the theo-

[3] https://github.com/long123625/CoPPT/tree/master

[4] Star queries have ternary patterns connected at a single point, i.e., they have the same subject. According to Bonifati et al.'s[8] analysis of real SPARQL query loads, it is concluded that more than 99% of BGPs are star queries.

retical level. This section presents work related to well-designed query processing in terms of theoretical research and system implementation.

(1) **Query Theory.** Pérez et al.[2] proposed the well-designed query for the first time. [7][10] reveal that well-designed queries have a high frequency of occurrence. [11][12][16] have extensively analyzed the semantics of the well-designed pattern and proved that it has ideal computational complexity. Letelier et al.[4] first proposed the WDPT and demonstrated that well-designed queries could be equivalently converted to WDPT. Romero[13] researched query evaluation complexity. It is also shown that well-designed queries that can be evaluated in polynomial time are exactly those with bounded dominant width. On this basis, Mengel et al.[17] extend their research to queries with PROJECTION operators and characterize the processable classes in polynomial time under the assumption of complexity theory. Barceló et al.[18] also optimized well-designed queries based on WDPT. This work proposed semantic optimization and approximate query theory on well-designed queries.

(2) **Query Processing System.** PIWD[14] is a query processing framework that uses the gStore[19] engine at the bottom layer. It proposes reconstructing the query mode into a well-designed query form that includes AND and OPT operators, but it does not specifically optimize the OPT operator. Treat PIWD as a left outer join operation in algebra. Xiao [23] optimize opt matching in the Ontology-Based Data Access (OBDA) setting, where the data is stored in a SQL relational database and exposed as a virtual RDF graph by means of an R2RML mapping. LBR[9] is a SPARQL query system specially optimized for well-designed queries. Its main idea is to use the GoSN to capture the OPT relationship of the query and construct the join variable graph, to capture the variable connection relationship between RDF triples, and then process the OPT operation and join variables based on semi-join reduction to reduce the intermediate results as much as possible. For inner-join optimization, existing research works like TriAD[24], RDF-3X[20], Stylus[21], S2RDF[27] and PRoST[29] mainly focuses on BGP query, and the optimization of OPT operator is not good.

In this paper, we enhance query performance by leveraging WDPT. Within the WDPT nodes, we employ BGP and optimize their query joins using the CS merging method. For the OPT relationships between WDPT nodes, we propagate constraints between parent and child nodes to minimize intermediate results and further refine the queries. In conclusion, our approach improves not only the BGP queries but also the overall efficiency of OPT within the WDPT framework.

3 Preliminaries

In this section, we introduce several basic background definitions of well-designed queries and pattern trees.

Definition 1. ***Well-designed SPARQL Query.*** *Given a SPARQL query Q containing only the AND, OPT, and FILTER. Q is a well-designed query if it meets the following criteria:*

– *For any form of the* $Q = (Q_1\ FILTER\ R)$, $\text{vars}(R) \subseteq \text{vars}(Q_1)$. *The* Q *is safe. All Variables in the constraint* R *are a subset of the variables in* Q_1. *That is, all variables that appear in* R *also appear in* Q_1.

– *For any subquery of the form* $Q_1 = (Q_2\ OPT\ Q_3)$, *all variables that also appear in* Q_3, *outside* Q_1, *must also appear in* Q_2.

Definition 2. ***OPT Normal Form.*** *A pattern* P *is in OPT normal form if it meets the following two conditions:*

– P *only contains AND and FILTER operators;*

– $P = (P_1\ OPT\ P_2)$, *where* P_1 *and* P_2 *are basic graph patterns.*

Definition 3. ***Pattern Tree.*** *A pattern tree is a tuple* $\mathcal{T} = (T, \mathcal{P})$, *where* $T = (V, E, r)$, *is a tree with node set* V, *edge set* E *and root* r, $\mathcal{P} = (P_n)_{n \in V}$ *is then the set of node labels on the tree* T, P_n *is the non-empty set of triple patterns corresponding to the nodes in the tree.*

Definition 4. ***Well-designed Pattern Tree(WDPT).*** *A pattern tree* $\mathcal{T} = ((V, E, r), \mathcal{P})$ *is deemed well-designed if, for every variable ?x present within* T, *the node that encompasses the variable ?x is connected, the pattern tree qualifies as a well-designed pattern tree.*

Figure 2 (a-b) illustrates the pattern tree associated with the two SPARQL queries. Within each tree, the tree represents a BGP constructed by a set of triple patterns, and parent-child relationships between nodes represent OPT in queries. Only the pattern tree in Figure 2(b) is a well-designed pattern tree, because the nodes where any of the same variables are located are connected.

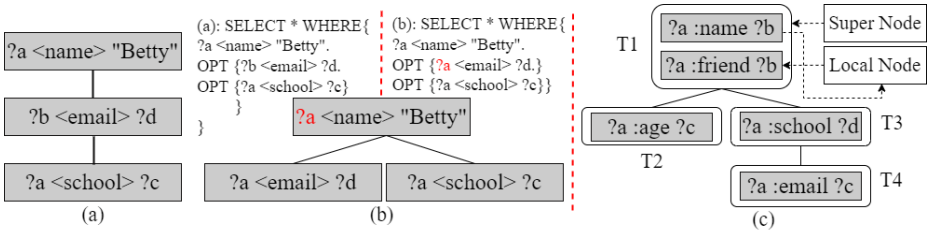


Fig. 2: SPARQL Query Pattern Tree and Pattern Tree

4 CSs Merging Based on Predicate Correlation

Recently works have shown that CSs[30] can capture the implicit schema of RDF data and improve optimization.The CS for a subject is the set of predicates on the outgoing edges from that subject. Formally, given an RDF dataset D, a subject s, the characteristic sets $CS(s)$ of s is defined as follows:

$$cs(s) = \{p \mid \exists o : (s, p, o) \in D\} \tag{1}$$

Subjects with the same CSs are often similar, i.e., belong to the same class of entity nodes. Also, the predicates in the CS are related and often appear in the same query, forming a star query. Star query is the most common query type in SPARQL query[8]. Directly using the CS to build the property table can improve query performance. However, when the data load is complex, and the number of CSs becomes very large, the number of final data tables and the number of CSs will result in a highly partitioned data organization[31][32]. Therefore, we merge CSs to reduce the number of CSs.

4.1 Predicate Correlation for Data Tables

Papastefanatos et al.[32] point out that merging CSs based on the hierarchical structure reduces the number of CSs. However, it only works in subset inclusion relations between property sets. The predicates in the same CS are related. Such predicates often appear together in a query and should be classified into the same data table. However, a predicate may appear in more than one CS. In this paper, the predicates with a higher degree of correlation are grouped together by measuring the degree of correlation between predicates. Intuitively, the higher the frequency of two predicates appearing in the same CS, the stronger the correlation, but at the same time, the respective frequency of occurrence of the two predicates needs to be considered. Formally, given a RDF dataset D, two predicates p_i and p_j in D, the predicate correlation can be calculated:

$$PCD(p_i, p_j) = \frac{PC\left(p_i, p_j\right)}{\min\left(PF\left(p_i\right), PF\left(p_j\right)\right)} \tag{2}$$

$PC\left(p_i, p_j\right) = |\{CS \mid \exists CS \mid p_i, p_j \in CS\}|$, indicates the number of CS that contain both the predicate p_i and the predicate p_j, $PF(p_i) = |\{s \mid \exists s, o : (s, p_i, o) \in D\}|$, indicates the predicate frequency of the predicate p_i, i.e. the number of subjects in triples containing p_i, and $PF(p_j) = |\{s \mid \exists s, o : (s, p_j, o) \in D\}|$, indicates the predicate frequency of the predicate p_j.

Expand to multiple predicates to obtain the correlation of predicates in a CS or data table. Given a data table T, the average correlation between the predicates of the data table is defined as follows:

$$PCD(T) = \frac{\sum_i^n \sum_j^n \frac{PC(p_i, p_j)}{\min(PF(p_i), PF(p_j))}}{n^2} \tag{3}$$

n is the number of predicates in the T, p_i, p_j are the predicates in the T, $PC\left(p_i, p_j\right)$, $PF\left(p_i\right)$ and $PF\left(p_j\right)$ indicate the same meaning as the equation 2 .

4.2 CSs Merging

When there are too many data tables, the join cost increases, and when there are too many null values in the data table, the data storage cost increases. To

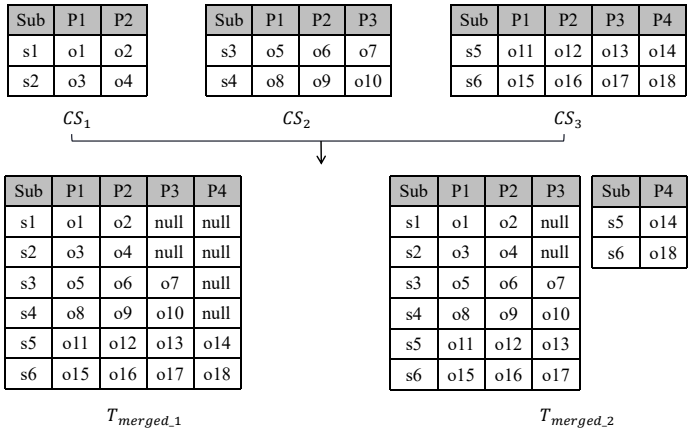


Fig. 3: Example of CSs merging

avoid these two situations, we propose CS merging. Figure 3 is an example of CSs merging, there are many combinations in the process of CSs merging, and different data tables are generated.

Intuitively, the more predicates merged in the table, the more null values exist in the table. We also consider the null values in the table while merging CSs. Given a table T with n predicates, let the number of rows in the table T be AP, and the null ratio of T is as follows:

$$NIL(T) = \frac{NULL(T)}{SUM(T)} = 1 - \frac{\sum_i^n PF\,(p_i)}{AP * n} \tag{4}$$

$NULL(T)$ is the number of null cells in the T, $SUM(T) = AP * n$ is the total number of cells in the T, $PF(p_i)$ is the frequency of the predicate p_i. In order to obtain the exact number of rows AP, it is necessary to obtain the subject union of all predicates first, and then calculate the number of elements in the union. This makes the computation of the null ratio expensive. Therefore, we made a further approximation optimization, assuming that the null of all predicates is brought by the predicate with the highest predicate frequency in the current table, to reduce the cost of null calculation. Thus, the cost function of merging CSs is as follows:

$$Cost(T) = \frac{NIL(T)}{PCD(T)} = \frac{\frac{\sum_i^n (PF(p_{max}) - PC(p_i, p_{max}))}{n * PF(p_{max})}}{\sum_i^n \sum_j^n \frac{PC(p_i, p_j)}{\min(PF(p_i), PF(p_j))} / n^2} \tag{5}$$

p_{max} is the predicate with the highest predicate frequency in the current table T, $PF(p_{max})$ is the predicate frequency of p_{max}, $PC\,(p_i, p_{max})$ is the number of CSs that contain both predicate p_i and predicate p_{max}. Finally, the process of merging CSs is solved based on the greedy algorithm. The algorithm for merging CSs is shown as Algorithm 1.

Algorithm 1: CS Merge

Input: All of the Characteristic sets: CSs, Number of Predicates: k,
Frequency of Predicates: PF, Collection of visited predicates: visit
Output: All data tables: tableSet

```
1  cost ← + ∞, table ← ∅;
2  for each CS ∈ CSs do
3      if Cost(CS) < cost then
4          cost ← Cost(CS); table ← CS; visit ← table;
5  while visit.size < k do
6      PFMax ← 0; PFMaxPre ← null;
7      for p_i ∈ table do
8          if PF(p_i) > PFMax then
9              PFMax ← PF(p_i); PFMaxPre ← p_i;
10     for p_j ∈ PFMaxPre do
11         if Cost(TNew) < cost then
12             cost ← Cost(TNew); visit.add(p_j); table ← TNew;
13             for preSet ∈ preSets do
14                 preSet.remove(p_j);
15     tableSet.add(table);
16     if visit.size < k then
17         for preSet ∈ preSets do
18             Cost(preSet);
19         cost ← Min(Cost(preSet)), table ← preSet;
20 return tableSet;
```

In algorithm 1, it is exponential to enumerate the predicate combinations in all CSs(**Input**). We select the predicate set with the least cost from all CSs as the initial data table for merging. When looking for the predicates related to the current data table, we make an approximate optimization and only select the predicates related to the predicates with the highest predicate frequency as the merged object instead of enumerating related predicates of all predicates in the data table(**line 1–4**). During the merging process, use *visit* to record all predicates that have been used to ensure that all predicates are only partitioned into one data table. Also, predicates that have already been used need to be removed from the remaining predicate set to update the remaining predicate set and their costs(**line 5–19**). This process is repeated until all predicates have been partitioned into one data table, i.e. *visit* contains all predicates. Finally, return all generated data tables *tableSet*.

4.3 Star Decomposition within Pattern Tree

Star decomposition based on the pattern tree makes the query simple while ensuring that the subqueries are in the same data table, which ensures efficient

query execution. Inside the pattern tree node is a BGP query, decompose the query in the node, process the obtained subqueries separately, and then join the results of the subqueries. The storage of CoPPT is the property table constructed from CSs, which has advantages for star queries. All predicates of each subquery are required to be located in one data table during query decomposition. In Figure 4, for instance, the Q is decomposed into three subqueries Q_1, Q_2, and Q_3, and the predicates in each subquery are located in the same data table.

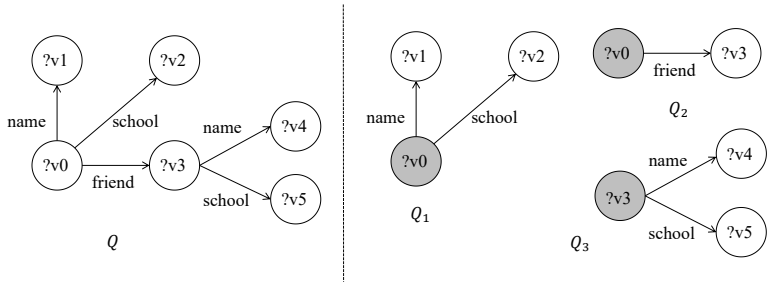


Fig. 4: Star decomposition of query Q

5 Constraint Passing on Pattern Trees

In WDPTs, the parent node has constraints on the child nodes. This optional matching produces many intermediate results. To solve this problem, we propose an optimization method for constraint passing on pattern trees.

5.1 Pattern Tree Structure

Given a query pattern tree, define the corresponding attributes inside the nodes to describe the structure of the pattern tree as follows:

- **Super Node**: for each pattern tree node, triple patterns with the same subject are divided into a node, called a Super Node.
- **Local Node**: for each Super Node, the triple patterns corresponding to the predicates located in the same data table are further divided into the same node, which is called the Local Node.

Figure 2(c) shows an example of pattern tree properties for a well-designed query. There is a Super Node in the T1 node, the subject is $?a$, and there are two Local Nodes in the Super Node, which means that the triple pattern $(?a, :name, ?b)$ and $(?a, :friend, ?c)$ are located in different data tables and belong to different subqueries. The Local Node corresponds to the subquery decomposed inside the pattern tree node, and each Local Node can be indexed and distinguished by the name of the table.

5.2 Constraint Passing

Each Local Node corresponds to a star subquery; no joins between tables are required. Define $sub(\cdot)$ to represent the subject of the node or triple pattern, and

$tab(\cdot)$ to represent the data table corresponding to the node or triple pattern. Given two triple patterns tp1 and tp2, if sub(tp_1) = sub(tp_2), and tab(tp_1) = tab(tp_2), then triplet patterns tp_1 and tp_2 are considered to have the same schema and can be merged into a Local Node.

To ensure the correctness of sub-node query rewriting, the matching results of the variables in the triple pattern copied from the sub-node are deleted. The rules for triple pattern replication between parent and child nodes are as follows: Given two pattern tree nodes n_1 and n_2, Local Nodes $local_1$ and $local_2$ are located in n_1 and n_2, respectively if the following three conditions are met at the same time:
(1) n_1 is the parent node of n_2;
(2) $local_1$ and $local_2$ have the same subject, i.e. sub($local_1$) = sub($local_2$);
(3) $local_1$ and node $local_2$ have the same data table, i.e. tab($local_1$) = tab($local_2$).

Then the node $local_1$ can be copied to the child node n_2 and merged with the node $local_2$, and maintain several attributes in the node $local_2$, including the triple pattern set TP_{up} and the variable set var(TP_{up}) in $local_1$. Finally, to ensure the correctness of query rewriting, delete the variables that only exist in $local_1$, var($local_1$) - var($local_2$).

Algorithm 2: Pattern replication between parent and child nodes

Input: Pattern tree parent node: fa, Pattern tree child node: ch
Output: Triple patterns copy tag: flag

```
1  flag ← false, chVarSet ← ch.varsSet ;
2  for superNode ∈ ch.getAllSuperNode do
3      sub ← superNode.subject, lnList ← superNode.localNodes;
4      for ln ∈ lnList do
5          lnFa ← fa.findLocalNodeBySchema(ln.subject, ln.table);
6          if lnFa ! = null then
7              flag ← true, tripleUp ← ∅ ;
8              for triple ∈ lnFa.getTPs do
9                  tripleUp.add(triple);
10             for each triple ∈ tripleUp do
11                 ln.tps.add(triple), ln.tpsUp.add(triple);
12             ln.varSetUp ← getVarsOfTps(lnfa);
13             ln.varSetUp.removeAll(chVarSet);
14 Return: flag;
```

Algorithm 2 describes the process of replication triple patterns between parent and child nodes on the pattern tree. Traverse each Super Node in $ch(childnode)$ to obtain the subject sub and Local Node list $lnList$ corresponding to the Super Node(**line 2–3**). Traverse each Local Node in the Local Node list $lnList$, find Local Nodes with the same pattern structure from $fa(parentnode)$, and perform triple pattern copy operations between parent and child nodes to trans-

fer constraints(**line 4–5**). Maintain the set of triplet patterns copied from fa and maintain variables that only exist in the copied set of triplet patterns(**line 7–13**).

Figure 5 is an example of the pattern tree after constraint passing based on triple pattern replication between Pattern Tree nodes. The superNodes of node T4 contain two Super Nodes, which are respectively from the Super Node of node T3 itself and the Super Node in its superNodes. The triple pattern $(?a, :name, ?b)$ merged in the T4 node comes from the superNodes of its parent node T3, and originally from the T1 node. Therefore, by maintaining the corresponding attributes, discontinuous triple patterns located in the same data table can be merged into the Local Node at a lower cost, thus implementing the implicit constraint of the OPT operator and reducing intermediate results, thereby optimizing query performance.

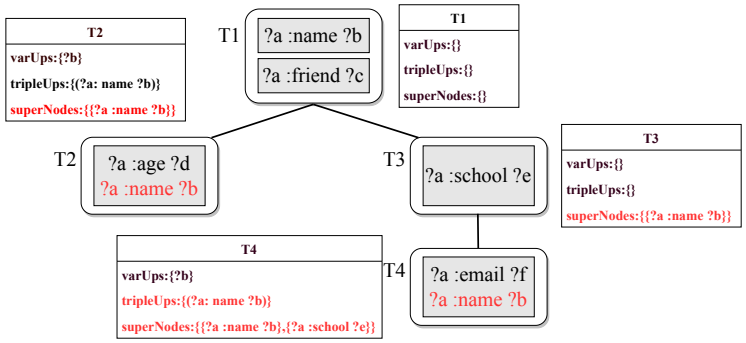


Fig. 5: Pattern tree after constraint passing

6 Experiments

We have carried out extensive experiments on both synthetic and real-world RDF datasets to verify the efficiency and scalability of CoPPT and compare it with distributed query engines S2RDF[27][5] and PRoST[29][6] that support OPT.

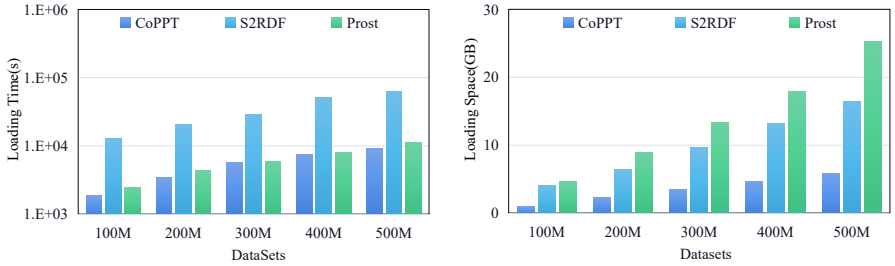
Experimental Environment: CoPPT is implemented in Spark using Scala and deployed on a 4-site cluster connected by Gigabit Ethernet. Each site is equipped with 128G RAM, 2TB disk, and a 4 core Intel(R) Xeon(R) E5-4603 of 2.00 GHz. We used Spark 2.4.0 and Hadoop 3.0.0 on Linux (64-bit CentOS).

Dataset: (1) WatDiv[7] is a benchmark for RDF query, which generates RDF datasets of different sizes, effectively validating the query efficiency and scalability. We generated 5 scale datasets for WatDiv(100M $\sim$ 500M). (2) DBpedia[8] is a dataset composed of extracted structured information from Wikipedia. There

[5] https://github.com/aschaetzle/S2RDF
[6] https://github.com/tf-dbis-uni-freiburg/PRoST
[7] http://dsg.uwaterloo.ca/watdiv/.
[8] http://downloads.dbpedia.org/2016-10/core/.

are many predicates in the DBpedia. Considering that the S2RDF and Prost systems use vertical partitioning as a store, it takes too much time. We filter the triples where the predicates with frequency greater than or equal to 10,000. Finally, the dataset contains 348,325,953 triples and 960 predicates.

Well-designed queries: For the above datasets, evaluations are performed using well-designed queries with varying selectivity and complexity. For the WatDiv, its query evaluation templates do not contain OPT operators, so we construct 8 well-designed queries; for the DBpedia, use the queries proposed in [9], and constructed several query patterns with higher frequency of query workload. All of our queries are in the GitHub link[9]



(a) Watdiv Loading Time (b) Watdiv Loading Space

Fig. 6: Loading time and space of Watdiv dataset

Loading Time Figure 6(a) gives the loading time of three systems over the WatDiv100M $\sim$ WatDiv500M. CoPPT has the smallest loading time. Although CS merging needs to consider the predicate correlation, predicate partitioning ensures that the merging is fast since each predicate will only appear in one table. S2RDF uses extended vertical partitioning(ExtVP), which needs to preprocess the semi-join between all predicates. Therefore, S2RDF has the longest data processing time. Prost uses a combination of vertical partitions(VP) and property tables to store data, which takes less time than S2RDF.

Figure 6(b) gives the HDFS overhead incurred by the three systems over WatDiv100M $\sim$ 500M. The HDFS space used by CoPPT is the smallest. Since each predicate appears in only one table, CoPPT doesn't store data repeatedly. Prost needs to store data twice in order to take advantage of VP and property tables. Therefore, the HDFS space consumed by prost is the largest. The storage time and space overhead of the CoPPT are the smallest for DBpedia, which contains a large number of predicates. In contrast, both Prost and S2RDF require vertical partitioning of the data, significantly prolonging the processing time. Additionally, the redundant storage of data in these methods results in substantial space requirements. Consequently, the results for DBpedia are not included in the comparison.

[9] https://github.com/long123625/CoPPT/tree/master

6.1 Efficiency

We refer to the system using the constraint passing as optCoPPT[10]. We compare the query efficiency of CoPPT, optCoPPT, S2RDF and Prost over the WatDiv and DBpedia datasets. Table 1 to Table 5 show the query performance of four systems on the WatDiv.

Table 1: Query time(ms) on different queries over WatDiv100M

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
optCoPPT	9308	14363	8388	10928	11340	10339	11877	11112
CoPPT	13614	14746	11048	18741	19747	19923	17047	19718
S2RDF	96610	318280	151149	113127	96438	103345	146980	148174
Prost	40402	68222	60220	31328	36823	36325	31157	31515

Table 2: Query time(ms) on different queries over WatDiv200M

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
optCoPPT	8643	13176	8418	11216	14362	12432	13132	13503
CoPPT	14501	14608	8423	21877	15645	22698	22633	24905
S2RDF	111709	413440	192654	133270	113939	128567	196252	200806
Prost	42225	113915	76664	36133	38813	43364	37229	37294

Table 3: Query time(ms) on different queries over WatDiv300M

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
optCoPPT	10836	15627	12246	15995	15757	16316	14694	15357
CoPPT	15021	16091	15672	18986	19042	27381	25098	25414
S2RDF	129852	499502	277585	145479	117976	138308	223238	226426
Prost	43518	146939	90024	37908	40298	45496	44596	47022

Table 4: Query time(ms) on different queries over WatDiv400M

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
optCoPPT	13191	18676	15991	16159	18810	16047	16242	17380
CoPPT	16556	21620	9574	20587	20862	30600	25711	27240
S2RDF	135723	577352	291502	195613	141765	166804	315456	254996
Prost	42781	188689	109221	39640	42951	48344	47201	51081

Table 5: Query time(ms) on different queries over WatDiv500M

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
optCoPPT	13275	21901	16225	18367	17869	18727	18650	18883
CoPPT	18450	27177	16071	21190	22457	32975	32345	30493
S2RDF	138685	700660	397409	229522	149508	170224	348338	316662
Prost	48225	228689	125221	41540	44863	50321	54364	60071

As shown in the tables, optCoPPT has the best query efficiency on most queries and CoPPT is also much better than S2RDF and Prost. Comparing S2RDF and optCoPPT, although S2RDF pre-stores the semi-join table between predicates in the storage stage when the query cannot use the ExtVP table, the query speed of S2RDF is slow. The optCoPPT implements query optimization based on the constraint properties between nodes of the pattern tree transfer for the potential constraint nature of OPT operations. From the system's query results on Watdiv datasets of various scales, it can be seen that the optCoPPT's query speed is much faster than that of S2RDF, which proves the effectiveness of the proposed query optimization method. Compared with the S2RDF, the optCoPPT has a minimum speedup ratio of 7.49 (Q5 over Watdiv300M) and a maximum

[10] The optCoPPT is based on the CoPPT with the addition of constraint passing optimization methods

speedup ratio of 31.99 (Q2 over Watdiv500M). Prost stores data using the vertical partition and property table methods simultaneously, processes the data in two ways and stores them twice so that when the query is executed, the method that is beneficial to the current query execution can be selected according to the query's characteristics. Although both Prost and optCoPPT use attribute tables to optimize some star join queries, Prost does not specifically optimize OPT operations, so its performance is low when processing queries with OPT operators. Over the Watdiv300M dataset, the lowest speedup of optCoPPT over Prost is 2.37 (Q4), and the highest speedup is 9.40 (Q2).

OptCoPPT adds a query optimization method based on constraint passing on the pattern tree to filter redundant data of child nodes in advance and optimize query performance on the basis of CoPPT. The query results of Watdiv show that for most queries, the query performance of the optCoPPT is better than that of CoPPT, which proves the effectiveness of the constraint-passing optimization method.

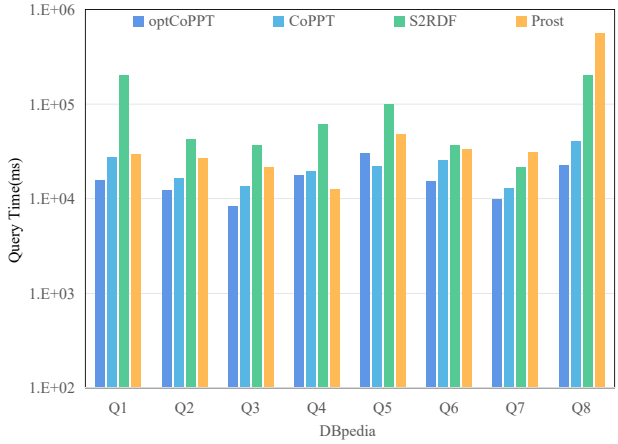


Fig. 7: Query time(ms) on different queries over DBpedia

Figure 7 shows the query time of all systems on DBpedia. Q1 to Q6 are the queries used in LBR. Q7 and Q8 are the queries constructed in this paper according to the query load. On most of the queries, optCoPPT performs the best, followed by CoPPT, which proves that the proposed query optimization method is also effective on real-life datasets. However, when processing query Q4, the optimal one is the Prost. For Q4, Prost can just use the property table to reduce part of the join cost, but the optimization methods of optCoPPT and CoPPT cannot work. However, the query time of the optCoPPT is still smaller than that of CoPPT and S2RDF, indicating that the constraint on the passing pattern tree can optimize the query. For Q5, the semi-join optimization already filters most of the redundant data on the pattern tree nodes, so the optimization of passing inter-node constraints to filter the redundant data is not as effective. Furthermore, the passing of constraints itself is computationally costly, so the query performance is degraded. Moreover, we calculated the average times of

four systems on WatDiv over the 8 queries. Experimental results show that our optimization method has high scalability.

7 Conclusion and Future Work

In this paper, we present CoPPT, a system designed to optimize well-designed queries. (1) We implement a system based on WDPT that proves the validity of theoretical work([4]). (2)We propose an efficient storage approach specifically for well-designed query systems. (3) Our system makes the leap from optimized BGP to OPT. (4) We optimize more than 70% of real queries. We consider SPARQL query pattern characteristics and optimize OPT queries in terms of weakly monotonicity. In the future, we will study systematic work on weakly well-designed(WWD) SPARQL patterns and beyond well-designed(BWD) SPARQL patterns.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Hammoud, M., Rabbou, D.A., Nouri, R., Beheshti, S.M.R., Sakr, S.: DREAM: Distributed RDF engine with adaptive query planner and minimal communication. Proc. VLDB Endow **8**(6), 654–665 (2015)
2. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems **34**(3), 1–45 (2009)
3. Arenas, M., Perez, J.: Querying Semantic Web Data with SPARQL. In: Proceedings of the ACM Symposium on Principles of Database Systems, pp. 305–316. ACM (2011)
4. Letelier, A., Pérez, J., Pichler, R., Skritek, S.: Static analysis and optimization of semantic web queries. ACM Transactions on Database Systems **38**(4), 1-45 (2013)
5. Saleem, M., Ali, M., et al.: LSQ: The linked SPARQL queries dataset. In: Proceedings of the 14th ISWC, pp. 261–269. Springer, Bethlehem (2015)
6. Xingwang, H., Zhiyong, F., Xiaowang Z., et al.: On the statistical analysis of practical SPARQL queries. In: Proceedings of the 19th International Workshop on Web and Databases, pp. 1–6. Association for Computing Machinery (2016)
7. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M. A.: An Empirical Study of Real-World SPARQL Queries. In: Proceedings of the 1st International Workshop on Usage Analysis and the Web of Data, pp. 1–4. USEWOD, Hyderabad (2011)
8. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. The VLDB Journal, **29**(2), 655–679 (2020)
9. Atre, M.: Left bit right: For SPARQL join queries with OPTIONAL patterns (left-outer-joins). In: Proceedings of the 42th ACM SIGMOD International Conference on Management of Data, pp. 1793–1808. Association for Computing Machinery (2015)
10. Picalausa, F., Vansummeren, S.: What are real SPARQL queries like? In: Proceedings of SWIM workshop at SIGMOD, pp. 1–6. ACM (2011)
11. Arenas, M., Pérez, J.: Querying semantic web data with SPARQL. In: Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 12–16. ACM, Athens (2011)
12. Kostylev, E.V., Grau, B.C.: On the Semantics of SPARQL Queries with Optional Matching under Entailment Regimes. In: Proceedings of the 14th International Semantic Web Conference, pp. 374–389. Springer, Bethlehem (2014)

13. Romero, M.: The tractability frontier of Well-designed SPARQL queries. In: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 295–306. Association for Computing Machinery (2018)
14. Xiaowang, Z., Zhenyu, S., Zhiyong, F., et al.: PIWD: A plugin-based framework for Well-designed SPARQL. In: Proceedings of the 6th Joint International Conference, pp. 213–228. Springer (2016)
15. Kaminski, M, Kostylev, EV.: Complexity and Expressive Power of Weakly Well-designed SPARQL. Theory Comput Syst. **62**(4), 772–809 (2018)
16. Schmidt, M., Meier, M., Lausen, Georg.: Foundations of SPARQL Query Optimization. In: Proceedings of the International conference on database theory(ICDT), pp. 4–33. ACM, Lausanne (2010)
17. Mengel, S., Skritek, S.: Characterizing tractability of simple Well-designed pattern trees with projection. Theory of Computing Systems **65**(1), 3–41 (2021)
18. Barceló, P., Pichler, R., Skritek, S.: Efficient evaluation and approximation of Well-designed pattern trees. In: Proceedings of the 34th ACM SIGMOD, pp. 131–144. Association for Computing Machinery (2015)
19. Zou, L., Özsu, M., Chen, Lei., Shen, X.: gStore: A graph-based SPARQL query engine. The VLDB Journal **23**(4), 565–590 (2014)
20. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB Journal **19**(1), 91–113 (2010)
21. Liang, H., Shao, B., Li, Y., Xia, H., et al.: Stylus: A strongly-typed store for serving massive RDF data. Proceedings of the VLDB Endowment **11**(2), 203–216 (2017)
22. Wilkinson, K., Sayers, C., Kuno, H., et al.: Efficient RDF storage and retrieval in Jena2. In: Proceedings of the 1st International Workshop on Semantic Web and Databases, pp. 131–150. Springer (2003)
23. Xiao, G., Kontchakov, R., Cogrel, B., Calvanese, D., Botoeva, E.: Efficient Handling of SPARQL OPTIONAL for OBDA. ISWC(1), pp. 354–373. (2018)
24. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In: Proceedings of the 41st ACM SIGMOD International Conference on Management of Data, pp. 289–300. Association for Computing Machinery, Snowbird (2014)
25. Zhang, W., Sheng, Q., Taylor, k and Qin, Y.: Identifying and Caching Hot Triples for Efficient RDF Query Processing. In DASFAA, pp. 259–274. Springer (2015)
26. Yasin, M., Zhang, X., et al.: A Comprehensive Study for Essentiality of Graph Based Distributed SPARQL Query Processing. International Conference on DASFAA, pp. 156–170. Springer (2018)
27. Schätzle, A., Przyjaciel, M.: S2RDF: RDF querying with SPARQL on Spark. Proceedings of the VLDB Endowment **9**(10), 804–815 (2016)
28. Guo, X., Gao, H and Zou, Z.: Leon: A Distributed RDF Engine for Multi-query Processing. In International Conference on DASFAA, pp. 742–759. Springer (2019)
29. Matteo, C., Michael, F.: PRoST: Distributed execution of SPARQL queries using mixed partitioning strategies. In: Proceedings of the 21st International Conference on Extending Database Technology, pp. 469–472. OpenProceedings.org (2018)
30. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: Preceedings of the 27th International Conference on Data Engineering, pp. 984–994. IEEE, Hannover (2011)
31. Meimaris, M., Papastefanatos, G., Vassiliadis P.: Hierarchical property set merging for SPARQL query optimization. In: Proceedings of the 22nd International Workshop on Data Warehousing and OLAP, pp. 36–45. CEUR-WS.org (2020)
32. Papastefanatos, G., Meimaris, M.:Relational schema optimization for RDF-based knowledge graphs. Information systems Journal **104**, 101754–101772 (2022)