Top-k queries on RDF graphs

Dong Wang, Lei Zou *, Dongyan Zhao

Peking University, Beijing, China

Abstract

Recent years have witnessed unprecedented volumes of structured data published in RDF format. Taking full advantage of such data has attracted a growing amount of research interest from both academia and industry. However, efficient processing of top-k queries in RDF data is still a new topic. Most existing approaches ignore top-k queries, or only provide a limited number of ranking functions. In this paper, we provide an effective and efficient processing algorithm for top-k queries that consists of a novel tree-style index MS-tree and MS-tree-based filtering and pattern-matching functions. Firstly, candidate entities in RDF data were efficiently ranked and filtered through an MS-tree-based top-down method. Then, query structure patterns are matched in RDF data through graphic exploration. In order to handle more complex scoring functions, a dynamic variable selecting optimization is employed to accelerate the threshold decrease. We evaluate our solutions with both synthetic and real-world datasets. The experimental results show that our model significantly outperforms state-of-the-art approaches.

1. Introduction

As a core concept in the “Web of Data”, RDF data (Resource Description Framework) have attracted increasingly more attention in both academia and industry. Recently, a growing amount of data have been published in RDF format, such as government data (Linked Open Government Data [24,13]), map data (OpenStreetMap [11]), and knowledge-based data (FreeBase [5], DBpedia [3] and YAGO [25]), etc. Generally speaking, an RDF dataset is a collection of triples, each of which is denoted as an SPO (subject, predicate, object). Fig. 1 shows a flattened representation of an RDF dataset that we used as a running example. In order to query RDF repositories, the query language SPARQL was proposed by W3C.

The amount, and volume, of RDF datasets has increased significantly in recent years. For example, in the data.gov project, many open government datasets have been released in RDF format. There are also a number of available biological RDF datasets, such as Bio2RDF (bio2rdf.org) and Uniprot RDF (dev.isb-sib.ch/projects/uniProt-rdf). In order to answer SPARQL queries efficiently, a number of query engines (e.g., Jena [34], RDF-3X [20,21] and gStore [38]) and processing algorithms (e.g., [7,27,13,15]) have been developed.

Along with the growing RDF repository scales, the “information overload” problem creates additional difficulties for users when they query RDF repositories. To alleviate the information overload problem, an alternative approach is used in top-k queries, i.e., queries return k answers with the highest rank order by using a specific ranking function. In practice, a scoring function can be employed, where each candidate is assigned a score by the scoring function, and the k candidates with the highest scores are returned. In this case, users only need to focus on the k answers with the greatest significance regardless of

* Corresponding author.

E-mail addresses: wangd@pku.edu.cn (D. Wang), zoulei@pku.edu.cn (L. Zou), zhaodongyan@pku.edu.cn (D. Zhao).

http://dx.doi.org/10.1016/j.ins.2015.04.032
0020-0255/© 2015 Elsevier Inc. All rights reserved.
the scale of the dataset. Indeed, according to SPARQL specification 1.1, a top-$k$ query can be expressed using the “ORDER BY” and “LIMIT” clauses. Clearly, top-$k$ SPARQL queries are useful in large-scale data analyses. Example 1 demonstrates two top-$k$ SPARQL queries.

Example 1. Two example queries, $Q_1$ and $Q_2$, are listed below.

$Q_1$: Find two European countries which have the fastest economic growth.

```
SELECT x WHERE {
  ?x economicGrowth ?y.
  ?x locatedIn ?z.
  ?z hasName "Europe"
} ORDER BY DESC(?y) LIMIT 2.
```

$Q_2$: Which country has the most area difference when compared to its neighboring countries?

```
SELECT x WHERE {
  ?x area ?y.
  ?x neighbors ?x1.
  ?x area ?z.
} ORDER BY DESC(?y - ?z) LIMIT 1.
```

However, very few SPARQL query optimization techniques consider top-$k$ SPARQL queries. A possible solution is to answer SPARQL queries in a post-processing way, meaning that it should first answer an original SPARQL query without an “ORDER BY” clause, and then reorder the candidates. Obviously, if the original SPARQL query leads to a lot of intermediate results, the post-processing solution is not efficient. A “smart” algorithm should stop as early as it can guarantee that it has found the top-$k$ results.

To the best of our knowledge, there is only one solution in [18] which processes top-$k$ queries in a non-post-processing way. In [18], two new algebraic operators are introduced, a ranking operator and a rank-join operator, and query plans are generated based on the operators. However, this solution suffers from two main disadvantages. First, in order to compute the upper bound of the possible candidates during the joining processes, unprocessed variables are replaced with their maximum values. In this case, since most of the variables are unprocessed in the first joining steps, the upper bound is not well constrained. Second, based on the algorithm description in [18], too many random accesses are involved in the rank-join operation. If the dataset is too large to be cached in memory, SPARQL-RANK suffers from frequent I/O accesses.

In contrast, we propose a graph-exploration-based method to deal with top-$k$ SPARQL queries on RDF graphs. Once an entity with a potentially high score is found, the graph-exploration method is employed to find the candidate’s corresponding sub-graph matches. To improve the query’s performance, we propose an optimized upper-bound computation method to obtain a tight upper bound. As indicated in [37], the graph-exploration method can achieve high query performance. We experimentally study our method and compare it with SPARQL-RANK with both real and synthetic datasets. Experimental results show that our method outperforms SPARQL-RANK by orders of magnitude when used on large datasets.

The main contributions of our work are summarized as follows:

- Based on numerical values, we define the RDF data model with numerical properties and query semantics of top-$k$ queries. Moreover, the TopKRDF problem is introduced to model the user’s query requirements.
• We introduce an effective index MS-tree to evaluate top-k queries in RDF data efficiently. Based on an MS-tree, we can maintain a tight upper bound of the candidates, and stop the process as early as possible. We also implement an MS-tree-based top-k SPARQL query evaluation algorithm to find the results of TopK RDF problems.

• We carried out extensive experiments on both synthetic and real datasets to evaluate the effectiveness and the efficiency of our solution.

The rest of this paper is organized as follows. Section 2 reviews our method in the context of related works in the literature, and the problem is formally defined in Section 3. In Section 4, the framework of our solution is briefly introduced. The tree index employed to obtain the candidates is introduced in Section 5, and the algorithm used to obtain the final result is stated in Section 6. The experimental results are found in Section 7. Finally, we conclude the paper in Section 8.

2. Related work

2.1. Traditional RDF data management

Most of the existing SPARQL query engines can be separated into two categories: solutions based on the relational data model [19,34,7,12,1,20,21] and graph-based solutions [38,37]. In addition, most of them utilize the relational data model, and employ join-based algorithms to evaluate SPARQL queries.

Based on different data storages, the solutions based on the relational data model can be classified into three types. The earliest solution employs a single three-column table to store the subject, the predicate and the object of each statement. Accordingly, this method suffers from computationally expensive self-joins when the data scale is large. To alleviate the scalability problem, the multiple indices optimization method is employed, as in YARS2 [12] and RDF-3x [20,21]. In RDF-3x, the RDF data have six copies, and each copy is organized in a B+-tree with one of the six orders: spo, sop, pos, osp and ops, where s, p and o denote the subject, the predicate and the object in the RDF statement, respectively. Furthermore, an encoding technique is used to decrease the index size by encoding each statement into a bit string.

The second kind of management system employs property tables as the RDF data storage architecture, such as Jena [19,34] and Sesame [7]. In these solutions, the correlated properties (or subjects with similar properties) are clustered together, and each cluster generates a specified property table. Based on the clustered items, there are two kinds of property tables. The first one is called a clustered-property table, which groups together properties that tend to co-occur with same subjects. The second type2hopwww clusters subjects with the same type of properties together, which is called a property-class table.

The third category of RDF data-management system employs the vertically partitioned table method, where SW-store [1] and Hexastore [33] are representative approaches. For each element in an RDF statement (subject, predicate and object), this approach divides the RDF data into multiple two-column (the other two elements) tables based on the different values of a specific element. All the join processes can be executed as fast merge-joins. However, this approach does not scale well as the number of different values (entities, predicates and objects) becomes huge.

Different to approaches based on the relational data model, Zou et al. [38] and Zeng et al. [37] considered the SPARQL query evaluation as a graph-isomorphism problem. They modeled the RDF data as a graph, and employed the graph-exploration method to evaluate SPARQL queries. In some cases [37], the graph-exploration algorithm performs better (fewer intermediate results) than the join-based method.

Among these existing RDF data-management systems, all of the indices are based on their lexicographical order. In this case, these approaches cannot determine the order of the intermediate results before the query evaluation stops. Therefore, based on existing approaches, only the post-process method, which finds all candidates first and then ranks the candidates, can be employed to deal with the top-k problem.

2.2. Top-k queries in relational databases

The top-k problem in relational databases has been well studied in recent years. A comprehensive survey [17] concluded that the main techniques in top-k processing includes the query model, the data and query certainty method, the data-access method, the implementation method, and the ranking function method. Most approaches have the same features, such as: (1) dealing with selection problems, (2) finding the exact results in certain data, (3) data can be both randomly and orderly accessed, (4) the ranking function is monotone, and (5) the algorithm is designed based on database query engines. Since RDF data can be managed by utilizing the relational data model, the top-k query models and related algorithms can also be employed in RDF data. When the property-table technique is adopted to preserve the RDF data, it is advantageous to then adopt the techniques in relational databases to deal with the top-k SPARQL queries.

However, when the schema of RDF data is not regular enough, the property-table technique is ineffective and inefficient for RDF data management. If other storage models (e.g., the triple table) are adopted, it would not be effective to employ the unchanged relational-database technique in top-k queries of RDF data. First, in each statement there is only one property value. In this case, most of the top-k queries are computationally expensive self-join queries. Second, no mature query engine exists for SPARQL queries. The researchers should define their own indices, and implement basic operations. Different implementations and data-access methods may greatly affect the query’s performance.
2.2. Top-k queries in RDF data

As Valle et al. indicated in [28], ranking the query results may significantly boost effectiveness and scalability in many concrete application problems. However, the top-k problem in RDF data is still a new problem and has not been well studied yet. Until now, most of the “top-k” research such as [30,26,8,29,23,32,16,31] aim to find a high-weight RDF structure instead of traditional top-k semantics, as in relational databases. The ranking functions respectively consider the relevance between the query keywords and text in the data [26,29], the relevance between the matches of partial basic graph patterns [32,31], the property frequency and the link strength in the matched structure [16], and the weight of each triple in RDF data [8].

In contrast, some researchers consider the numerical values of RDF data in the ranking function, which are similar to the top-k semantics in relational databases. Battre et al. proposed a solution in [2] to retrieve the best k result ordered by a single property value in distributed RDF data. Since mathematical operations were not employed, the scoring function was limited. In [10], Garcia et al. considered multiple user preferences, including numerical values. The different ranking mechanisms were seamlessly and transparently integrated into the framework. In [22], Ovelgönne et al. proposed a numerical-term-based ranking function to rank query results. However, only additive and multiplicative operations were considered. In contrast, our query model employs all four arithmetic operations.

To the best of our knowledge, there is only one solution, proposed by Bozzon et al. [6,18], that takes the full top-k query semantics of SPARQL into consideration, which is similar to our query model. In [6,18], they extended the SPARQL algebra and proposed an algorithm, SPARQL-RANK, to efficiently evaluate top-k queries in an RDF graph. The RDF statements were accessed by either a random sequence or an ordered sequence, and a join-based-threshold algorithm was employed to stop the process when the top-k result was generated. Based on the DBpedia SPARQL benchmark, they built a benchmark for top-k SPARQL queries [36]. However, their work is not yet complete. In contrast, our solution utilizes the graph-exploration method instead of the join method. The pros and cons of the two methods are clearly analyzed in [37]. Moreover, as presented in Section 1, the SPARQL-RANK solution is not effective enough when applied to large datasets.

3. Problem definition

3.1. Data model

An RDF dataset is a list of triples ((subject, predicate, object)), as shown in Fig. 1. However, we can also model an RDF dataset as a graph, where the subjects and objects are vertices, and each triple is represented as a directed edge from the subject to the object.

Definition 1 (RDF graph). An RDF graph is a four-tuple \( G = (V, L_V, E, L_E) \), where

1. \( V = V_c \cup V_e \cup V_l \) is a collection of vertices that correspond to all subjects and objects in an RDF dataset, where \( V_c, V_e, \) and \( V_l \) are collections of class vertices, entity vertices, and literal vertices, respectively.
2. \( L_V \) is a collection of vertex labels. The label of a vertex \( u \in V_i \) is its literal value, and the label of a vertex \( u \in V_c \cup V_e \) is its corresponding URI.
3. \( E = \{ (u_1, u_2) \} \) is a collection of directed edges that connect the corresponding subjects and objects.
4. \( L_E \) is a collection of edge labels. Given an edge \( e \in E \), its edge label is its corresponding predicate.

An edge \( (u_1, u_2) \) is a property edge if \( u_2 \in V_l \); otherwise, it is a link edge. The label of a property edge is also called a property.

In an RDF graph, some entities have numerical values, such as the area and the population of a country. The scoring function is then based on the numerical values. The numerical values often co-exist with various specific properties (predicates), and these properties are called numerical properties.

Definition 2 (Numerical property). In an RDF graph, given an edge \( (\overline{u_1}, \overline{u_2}) \), if \( u_2 \) is a numerical value, the corresponding property, i.e., the label of \( e \), is called a numerical property.

Fig. 2 shows a running example of the graphic view of an RDF dataset that contains the areas and the economic growth ratios of some European countries. Nodes surrounded by rectangles are entities or classes with their own labels, and the other nodes are literal values. The text on the arrows are the edge labels. There are two numerical properties “area” and “economicGrowth”.

3.2. Query semantics

A SPARQL query is a list of patterns (also called triple patterns), and the result of a SPARQL query is a list of statements that satisfy the patterns. A SPARQL query can also be modeled as a graphic structure, \( Q \). Answering a SPARQL query is equivalent to finding sub-graph isomorphic matches of query graph \( Q \) in RDF graph \( G \).
Sometimes, the number of results is excessively large, and users may only require the most significant answers. In this case, besides the structured SPARQL query, it is useful to allow users to indicate their preferences (a scoring function \( F(\cdot) \)) and the answer-set size (a specific number \( k \)). Only the candidates with a rank no larger than \( k \) should be returned.

**Definition 3 (Top-k SPARQL query).** A top-\( k \) SPARQL query is a triple \( Q = (P_Q, F(\cdot), k) \), where \( P_Q \) is a structure pattern, \( F(\cdot) \) is a scoring function and \( k \) is the maximum number of results. The result of a top-\( k \) SPARQL query is a no-more-than-\( k \)-sized list of matches of \( P_Q \), with the highest scores based on \( F(\cdot) \). Note that \( F(\cdot) \) is monotonic to each variable.

Given \( P_Q, F(\cdot) \) and the variables \( \{x_1, x_2, \ldots, x_n\} \subseteq P_Q \), let \( \sigma(x) \) be a mapping \( \% \rightarrow \mathbb{R} \) where \( x \in \% \), \( \% \) is the variable space and \( \mathbb{R} \) is the real-number space, \( F(\cdot) \) is a combination of \( \sigma(x_i) \), i.e., \( F(P_{C}) = F(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n)) \). \( F(\cdot) \) is required to be monotonic, i.e., \( F(G) \geq F(G') \) if \( \forall i, \sigma(x_i) \geq \sigma(x'_i) \), where \( G \) and \( G' \) are isomorphic structures of \( P_Q \) in a data graph. Therefore, given an entity set \( S_e \), the score of a potential candidate, which includes all of the entities in \( S_e \), has an upper bound as shown below, where the over-line on the \( F(\cdot) \) represents the upper bound of the score.

\[
F(\overline{S_e}) = F \left( \begin{cases} 
\sigma(x_i), & \text{if } x_i \text{ has a mapping in } S_e, \\
\text{maximum possible value,} & \text{Otherwise}
\end{cases} \right), 
\]

Fig. 3 shows the graphic views of the two sample queries Q1 and Q2. Here, the scoring functions are \(?y\) and \(?y - ?z\) in Q1 and Q2, respectively, and the \( k \) parameters are set to be 2 and 1, respectively, as shown in Example 1. In the RDF graph in Fig. 2, all five countries are candidates of Q1 and Q2. The result of Q1 is Poland and France with scores of 1.8 and –1.2, respectively, and the result of Q2 is Russia with a score of 1.71E13.

4. Framework

The basic idea when evaluating a top-\( k \) SPARQL query is (1) to find all the matches of the structure pattern, (2) to calculate the scores of the matches, (3) to order the matches by their scores, and (4) to return the best \( k \) matches. Obviously, it is unnecessary to retrieve all of the matches since not all of the matches need to be returned. To improve the query performance, the retrieval process should stop as soon as we can guarantee that the best \( k \) candidates have been generated.

A heuristic clue to optimizing the process is to utilize the monotonicity of the scoring function \( F(\cdot) \). Consider two entity combinations \( E = (e_1, e_2, \ldots, e_n) \) and \( E' = (e'_1, e'_2, \ldots, e'_n) \). Let \( \nu(e) \) denote the value of the corresponding numerical property of \( e \). Suppose that we can find two candidates \( c \) and \( c' \) based on \( E \) and \( E' \), respectively. If \( \forall e_i, \nu(e_i) \leq \nu(e'_i) \), we have \( F(c) \leq F(c') \). As a result, if we could access the entities in descending order by their numerical values, we can progressively lower the upper bound of the unseen candidates. When the score’s upper bound of the unseen candidates is lower than the
Algorithm 1. The top-k candidate retrieval framework

**Require:** RDF Graph G, RDF Top-k Query Q.

**Ensure:** k results of Q.

1: Set $lb = \text{MIN}$, $ub = \text{MAX}$. //Set the lower and upper bounds.
2: Reset candidate k-size heap H.
3: while $lb < ub$ do
4: Obtain the next entity combination that corresponds to the variables.
5: Find candidates based on the combination.
6: Calculate the candidates’ scores and add the candidates to H.
7: if $H:\text{size} = k$ then
8: Set $lb = H_{\text{min}}$. //The minimum score in the heap.
9: Update $ub$ with the rest entities.
10: Output the k candidates.

According to the framework description, we have three main problems: (1) how to access the entities that are ordered by their numerical values, (2) how to find the candidates given an entity combination, and (3) how to compute the upper bound of the unseen candidates.

We employed an index called the MS-tree to deal with the first problem. In an MS-tree, each node maintains two extreme values of each numerical property: the minimum and the maximum among its children’s values. Accordingly, given a numerical property, if the minimum value of a node $n_1$ is not less than the maximum value of another node $n_2$, each corresponding property value of $n_1$’s descendants is no less than the corresponding values of $n_2$’s descendants. Given a numerical property $p$ and a priority queue $Q$, the tree nodes are traversed by utilizing $Q$. If the front one of $Q$ is an entity $e$, $e$’s corresponding numerical value of $p$ is confirmed to be the largest. In addition, the MS-tree maintains the structured feature of the RDF data nodes. If an entity $e$ is accessed, the neighboring structure of $e$ is guaranteed to be similar to a variable in the query, i.e., it is likely that we can find candidates near $e$.

Since the SPARQL-structure-pattern-matching problem can be modeled as a graph-isomorphism problem, given an arbitrary entity combination, the chance of finding a match in the structure pattern containing all of the combination’s entities in the RDF graph is very small. In practice, instead of selecting an entity combination, we selected an entity, and explored the RDF graph around the entity to find candidate matches. In our solution, for a specific variable, we maintained a priority queue to access the eligible entities based on the MS-tree. Once an entity reached the top of the queue, the graph-exploration process was executed to find candidates.

The upper-bound calculation is based on the previous analysis and the graph-exploration process. In the exploration process, given a starting entity $e$, every candidate containing $e$ is generated. In this case, we can confirm the possibility that unseen candidates contain $e$ or not. As a result, $e$ can be removed from the entity list after the exploration process. Based on the analysis, given an entity combination $\langle e_1, e_2, \ldots, e_n \rangle$, if we can guarantee that the corresponding numerical values of every $e_i, i = 1, 2, \ldots$ are the best candidate ones, then $F(\cdot)$ is the highest, i.e., the score $F(\cdot)$ is the upper bound of the unseen candidates. Suppose that the exploration process from $e_i$ is executed, $e_i$ is removed from the combination and replaced with the second best one. The new score after the replacement is the new upper bound, and the upper bound is updated.

5. MS-tree

In this section, we will introduce the MS-tree index, and describe the construction and the usage of an MS-tree.

5.1. Index structure

An MS-tree is a tree structure with the following features:

1. An MS-tree is a height-balanced tree.
2. Each node in an MS-tree has three features: a bit string, the max–min values of the numerical properties, and its child nodes. Note that leaves have no child nodes.
3. Each MS-tree has a child-number constraint $N$, where the child number’s scale of each node (except for the root and the leaves) is from $N/2$ to $N$.

---

Which is a prior, i.e., the bigger or the smaller one, which depends on the monotonicity of the scoring function.
4. The bit string of a leaf node is generated based on the neighbors of the corresponding entity in the RDF graph. The maximum and minimum values of a leaf node are set as the maximum and minimum values of the corresponding numerical properties of the entity, respectively.

5. The bit string of a non-leaf node equals $bs_1 | bs_2 | \ldots | bs_n$, where $bs_i$ is the bit string of its $i$-th child, and $|$ denotes the “OR” bit operation.

6. Given a specific numerical property, the maximum and minimum values of a non-leaf node is equal to the maximal and minimal values of its children’s corresponding maximum and minimum values.

The MS-tree is an extension of the S-tree [9], and a variation of the version of the VS-tree [38]. In an MS-tree, each entity in the RDF graph is encoded into a bit string (called a signature), and an S-tree is constructed based on the signatures. Different to the S-tree, the tree nodes of an MS-tree contain the aggregate values of its numerical properties.

When the S-tree is completely built, the max–min features are appended to the nodes of the S-tree. First, based on the data model, each entity has its own numerical value. Note that if an entity does not have a specific numerical property, the corresponding max–min features of the entity are set to be null. Then, given a node $n$ in an MS-tree and the child nodes $\{v_1, v_2, \ldots, v_m\}$ of $n$, the max–min features $f_{\max n}$ and $f_{\min n}$ of $n$ are computed according to the following formulations:

$$f_{\max n} = \max(f_{\max v_1}, f_{\max v_2}, \ldots, f_{\max v_m}), f_{\min n} = \min(f_{\min v_1}, f_{\min v_2}, \ldots, f_{\min v_m})$$

Note that only the non-null values are considered in the $\max(\cdot)$ and the $\min(\cdot)$ functions. If all the parameters of the $\max(\cdot)$ or $\min(\cdot)$ function are null, then the output value of the function is null. Fig. 4 shows a running example of an MS-tree.

5.2. Tree construction

The MS-tree construction method adopts a three-stage algorithm.

- **Encoding Stage**: Encoding the entities into signatures (bit strings).
- **Construction Stage**: Building an S-tree based on the signatures.
- **Assignment Stage**: Adding the max–min features to the tree nodes.

In the first stage, the encoding algorithm in [38] is adopted. Suppose that adjacent edges starting from $s$ are denoted as $\{e_1, e_2, \ldots, e_n\}$. For each $e_i$, several hash functions are employed to generate an $N + M$ length signature $\text{sig}_e$, where the front $N$ bits and the following $M$ bits are encoded based on the predicate and the object, respectively. For instance, if the hash codes of the predicate that are based on different hash functions are three and four, the third bit and the fourth bit are set to be 1, and the other of the front $N$ bits are set to be 0. The signature, $\text{sig}_s$ of $s$ is computed via combining the signature of the edges based on $\text{sig} = \text{sig}_e_i \| \text{sig}_e_2 \| \ldots \| \text{sig}_e_n$.

For example, in Fig. 2, there are four edges starting at Russia. Suppose that we set the first five bits for the predicate and the following five bits for the object, we can get four signatures that correspond to each of the four edges. The signature of Russia is 10111 01111. Fig. 5 shows the encoding processing for Russia.

When a query occurs, the query is also encoded into signatures. For example, the signature representations of Q1 and Q2 (in Fig. 3) are shown in Fig. 6.

![Fig. 4. An MS-tree.](image-url)
In the second stage, we employed the “insert” operation to build the MS-tree. Given a list of signatures, we inserted the signatures one by one into the MS-tree. A similar “insert and split” strategy was employed as a balanced n-ary tree. When a signature $n$ is created, the strategy works as follows.

1. Iteratively choose the node from top down with the lowest cost when inserting $n$ into it. If the chosen node $v$’s children are leaf nodes, generate a node with signature $n$ and insert it into $v$.
2. If $v$ is full, split $v$ to two separated node $v_1$ and $v_2$, where the respective costs of $v_1$ and $v_2$ are minimized. If a splitting operation makes the father of the split node full, split the father node recursively.
3. If the root is split into $r_1$ and $r_2$, generate a new root $r$ and let $r$ be father of $r_1$ and $r_2$.

Algorithm 2. Query pattern matching

| Require RDF Graph $G$, RDF Top-k Query $Q$, MS-tree $T$, Starting variable $v$. |
| Ensure Query pattern matchings. |
| 1: Encoding $v$ into a signature $s_v$. |
| 2: Reset the priority queue $PQ$ of the tree nodes. //Preserve the intermediate results. |
| 3: Reset the result queue $RQ$. |
| 4: Put the root node $r$ into $PQ$. |
| 5: while $PQ \neq \emptyset$ do |
| 6: Node $n = PQ.pop()$. |
| 7: if $n.sig \land s_v == 0$ then |
| 8: continue; |
| 9: if $n$ is an entity then |
| 10: Call function $GetResultFromEntity(n)$ in $G$. //Using the BFS method. |
| 11: if Obtained a valid result $r$ then |
| 12: Put $r$ into $RQ$. //Stop here if necessary. |
| 13: else |
| 14: Do nothing. //An entity candidate may fail to get a match. |
| 15: else |
| 16: for each child node $n_c$ of $n$ do |
| 17: Put $n_c$ into $PQ$. |
| 18: Output $RQ$. |

In the third stage, we implemented a recursive procedure to complete the MS-tree. First, the max–min features of each entity were appended to the signature node of the entity. Next, the features of the upper nodes were appended according to the child nodes of the upper nodes, i.e., the maximum/minimum feature of an upper node $n$ is the maximal/minimal one among the maximum/minimum features of $n$’s child nodes.

Here, the cost calculation adopted the cost model used in [38], i.e., the cost $c_{12}$ of merging $s_1$ and $s_2$ is $c_{12} = \text{bit count}(s_1 \text{xor } s_2)$. That is, the more similar that the two signatures are, the smaller the cost becomes.
5.3. Candidate generation

Given a specific variable that is based on the MS-tree, the candidate-generation procedure was divided into two stages. First, the entities (called entity candidates), which were probably contained in a match, were generated incrementally. Second, the BFS method (the GetResultFromN() function in Algorithm 2) was adopted to obtain matches containing the entity candidate.

In Algorithm 2, the root node of the MS-tree was visited first (line 4). Next, for each node \( n \) in the queue, if the neighbors of \( n \) did not meet the conditions, the sub-tree that was rooted on \( n \) could be safely pruned (line 6–8). Otherwise, the children of \( n \) were pushed into the queue, as ranked by the scoring function (line 15–18). If \( n \) did not have a child, i.e., \( n \) denotes an entity, the GetResultFromEntity() method (Algorithm 3) was adopted to find candidates of \( n \) (line 10). In the GetResultFromEntity() procedure, the edges in the SPARQL query were incrementally evaluated to find matches in the data graph. All valid candidates found by the BFS procedure were added to the result queue (line 11–14).

**Algorithm 3. GetResultFromEntity(\( n \))**

\[
\begin{align*}
\textbf{Require:} & \quad \text{RDF Graph } G, \text{ SPARQL Query } Q, \text{ Starting entity } n, \text{ Starting variable } \nu. \\
\textbf{Ensure:} & \quad \text{The candidates containing } n. \\
1: & \quad \text{Reset the intermediate result list } IL, \text{ and add } n \text{ to } IL. \\
2: & \quad \text{Reset variable list } VL, \text{ and add } \nu \text{ to } VL. \\
3: & \quad \text{Reset result list } RL. \\
4: & \quad \text{while } VL \neq \emptyset \text{ do} \\
5: & \quad \quad \text{Let } \nu = VL.\text{top}. \\
6: & \quad \quad \text{for each } R_i \in IL \text{ do} \\
7: & \quad \quad \quad \text{for each adjacent edge } e_j \text{ of } \nu \text{ do} \\
8: & \quad \quad \quad \quad \text{Let } \nu_j \text{ is the neighbor of } \nu \text{ in } e_j. \\
9: & \quad \quad \quad \text{VL.\text{push}(\nu_j)} \\
10: & \quad \quad \text{Find } e_j' \text{'s corresponding edge set } \{l_j\} \in G, \text{ and } \nu_j' \text{'s corresponding node set } \{n_j\} \in G \text{ from } R_i. \\
11: & \quad \quad \text{for each } l_k \in \{l_j\} \text{ and the corresponding node } n_k \text{ do} \\
12: & \quad \quad \quad \text{Add } n_k \text{ to } R_i \text{ and insert } R_i \text{ into intermediate result list } IL'. \\
13: & \quad \quad \text{IL} = IL'. \\
14: & \quad \text{Set } RL = IL \text{ and return } RL.
\end{align*}
\]

Suppose that the starting variable is \(?x\) in Q1 of Figs. 3 and 6. At the beginning, the root \( d_1 \) of the MS-tree in Fig. 4 was pushed into the queue. Since the maximum possible value of \( d_1 \) on economicGrowth is 1.8, and the scoring function is \( F(\cdot) = \text{DESC}(\cdot') \), the priority of \( d_1 \) was set to be 1.8. Then, children \( d_2 \) and \( d_2' \) were pushed into the queue. When \( d_2' \) was accessed, we found that the signatures \( s_{d_2} \) of \( d_2' \) and \( s_4 \) were 11000 00100 and 10011 01010, respectively, and \( s_{d_2} \& s_4 \neq s_4 \). Thus, the sub-tree rooted at \( d_2 \) could be safely pruned. After several iterations, the entity node Poland, with a priority of 1.8, reached the top of the queue. Then, a breadth-first-search algorithm was employed to find which candidates started from Poland. Based on Fig. 2, a candidate containing Poland was generated, which was \{Poland economicGrowth 1.8, Poland locatedIn Europe, Europe hasName “Europe”\}.

5.4. Correctness guarantee

**Lemma 1.** Algorithm 2 has no false negative, i.e., if an entity can be mapped to the variable \( \nu \), it would not be filtered out in Algorithm 2.

**Proof.** Given a variable \( \nu \) in a SPARQL query \( Q \), for each edge \( e \) starting from \( \nu \), the encoding procedure generated a corresponding signature of \( e \). Each edge \( e \) has a corresponding data edge \( e_d \) that satisfies \( e_d.\text{sk} \& e.s = e \), where \( x.s \) denotes the signature of \( x \). As a result, if a data node \( n_d \) can be mapped onto \( \nu, n_d \) satisfies \( n_d.\text{sk} \& \nu.s = \nu.s \).

According to the construction method of an MS-tree, given a tree node \( n \) and its descendant \( n' \), we have \( n'.\text{sk} \& n.s = n'.s \). Therefore, if data node \( n_d' \) can be mapped onto \( \nu \), and \( n_d \) is an ancestor of \( n_d' \), we have \( n_d.\text{sk} \& \nu.s = n_d.\text{sk} \& (n_d'.\text{sk} \& \nu.s) = (n_d.\text{sk} \& n_d'.s) \& \nu.s = n_d'.\text{sk} \& \nu.s = \nu.s \). According to Algorithm 2, all possible entities would not be filtered out.

**Lemma 2.** Given an entity \( e \), the function GetResultFromN(\( e \)) generates all candidates \( C = \{c_i\} \) iff. \( \forall c_i \in C, e \in c_i \) and \( \forall c_j \in \overline{C}, e \notin c_j \), where \( \overline{C} \) denotes the candidates which are not in \( C \).

**Proof.** (1) Since each \( c_i \in C \) is generated by employing the BFS method from \( e \), \( \forall c_i \in C, e \in c_i \) is self-evident.
Algorithm 4 can then fit the multiple-variable case. Based on Algorithm 2, the candidates were generated in descending property-value order. The ranking is credible.

6. Top-k SPARQL query evaluation

Based on the MS-tree index, we implemented a top-k results generation algorithm to obtain the highest score matches as ranked by the scoring function $F(\cdot)$. The simplest situation is that there is only one variable involved in the scoring function. For convenience, we first analyzed the single-variable case and displayed the solution, and then extended the solution to the multiple-variable case.

6.1. Single-variable case

If the scoring function only contains one variable $v$, the score of a candidate only depends on $v$. Therefore, it is easy to employ a maximum/minimum heap and the MS-tree-based query-pattern-matching algorithm to build the top-k results generation algorithm in the single-variable case. In the query-pattern-matching procedure, the returned candidates were ranked by the scoring function. Therefore, when $k$ candidates were generated by the query-pattern-matching procedure, the top-k results were obtained.

Algorithm 4. Single-variable case

| Require: RDF Graph $G$, RDF Top-k Query $Q$, MS-tree $T$, Starting variable $v$. |
| Ensure: The top-k query pattern matches. |

1: Reset the $k$-size maximum heap $H$ of the candidates. //Order by $F(\cdot)$.  
2: $\text{while true do}$  
3: Get the next candidate $c$ by calling Algorithm 2.  
4: $\text{if no more candidates then}$  
5: return $H$.  
6: $\text{Put c into H based on } F(c)$.  
7: $\text{if H has k candidates then}$  
8: return $H$.  

Take $Q1$ in Fig. 3 as an example. The starting variable was set to be $?x$ since there is only the $?x$’s economicGrowth property value in the scoring function. Based on Algorithm 2, the candidates were generated in descending property-value order. The first and second generated candidates were Poland and France, and the scores of these were 1.8 and $-1.2$, respectively. Since the scores of the remaining candidates were no larger than $-1.2$, the process stopped.

6.2. Multiple-variable case

6.2.1. Basic method

The multiple-variable case is more complicated since the $F(\cdot)$ score cannot be calculated when we only know the numerical value of just a single variable. Based on Definition 3, $F(\cdot)$ is monotonic. In this case, given a variable $v$, the possible $F(\cdot|v)$ is maximized when each $\sigma(x_i | x_i \neq v)$ is maximized, where each $x_i$ represents a variable. Algorithm 4 can then fit the multiple-variable case with only a small modification.

Suppose that variables $X = \{x_1, x_2, \ldots, x_n\}$ are involved in the scoring function $F(\cdot)$. When the process starts, a random variable $v \in X$ is selected, and the other variables $x_i$ in the scoring function are set to be constants with the maximum possible mapping value $\sigma(x_i)$. Given any $\sigma(v)$, the calculated score is the upper bound of the real score. Since the candidate entities of $v$ are decreasingly ordered by $\sigma(v)$, the calculated score is the upper bound of the remaining candidates. As soon as a candidate is generated, the real score is calculated. When at least $k$ candidates are generated, and the upper bound of the remaining candidates is no bigger than the $k$-th biggest score of the generated candidates, the algorithm stops.

Theorem 1. Given a SPARQL query $Q$. Algorithm 2 generates all candidates of $Q$.

Proof. According to Lemmas 1 and 2, all candidates are generated in Algorithm 2. \qed
In Algorithm 4, the modifications are listed below. First, the starting variable was fixed before the algorithm’s execution. Second, the stop condition (Line 7) was modified to a threshold-based condition, i.e., the lowest score in the heap was not smaller than the score’s upper bound of the next candidate.

6.2.2. Optimized method

**Algorithm 5.** Multiple-variable case

```plaintext
Require: RDF Graph G, RDF Top-k Query Q, MS-tree T, Scoring function F(·).
Ensure: The top-k query pattern matchings.
1: Reset the k-size maximum heap H of the candidates. ]Order by F(·).
2: Set upper bound \( ub = MAX \) and lower bound \( lb = MIN \).
3: Get all variables \( \{ v_1, v_2, \ldots, v_n \} \) involved with \( F(\cdot) \).
4: Reset an n-sized array, \( C_1[n] \), for the best candidates.
5: Reset an n-sized array, \( C_2[n] \), for the second-best candidates.
6: for \( i = 1 \) to \( n \) do
7:   if \( C_1[i] == null \) then
8:     Get the next entity candidate \( c_i \) that corresponds to \( v_i \), and set \( C_1[i] = c_i \).
9:     Compute \( ub \) using each \( c_i \in C_1[i] \).
10: while true do
11:   for \( i = 1 \) to \( n \) do
12:     if \( C_2[i] == null \) then
13:       Get the next entity candidate \( c_i \) that corresponds to \( v_i \).
14:       if no more entity candidates then
15:         Get all candidates from \( C_1[i] \), then add them to \( H \) and return \( H \).
16:       Set \( C_2[i] = c_i \).
17:       Compute the lowest \( ub' \) and \( j \) by replacing each \( C_1[j] \) in \( C_2[j] \).
18:       Let \( ub = ub' \) and \( c = C_2[j] \), and get all candidates from \( c \).
19:     Let \( C_1[j] = C_2[j] \) and \( C_2[j] = null \).
20:   for each candidate \( c \) do
21:     if \( c \) is not a redundant candidate then
22:       Insert \( c \) into \( H \), and set the priority to \( F(c) \).
23:     if \( H \) has been updated, and size(\( H \)) \geq k \) then
24:       Let \( lb = \) the minimal score in \( H \).
25:     if \( ub < lb \) then
26:       return \( H \).
```

The descending speed of the upper bound in the basic method was not guaranteed. If variable \( v \), which possesses low selectivity and low descending speed, was selected, the time cost may be high. To speed up the calculation of the descending value of the upper bound, we maintained multiple priority queues for the variables involved in \( F(\cdot) \), and computed the upper bound of the following matches by utilizing a greedy strategy.

In Algorithm 5, we maintained \( n \) priority queues according to the scoring function, instead of only one priority queue in the single-variable case. Since the scoring function involves multiple variables, we cannot generate the candidates one by one as ordered by their scores. Instead, the threshold strategy was employed. The lower bound of the current top-\( k \) candidates was the \( k \)-th maximum score of the candidates, and the upper bound of the potential candidates can be calculated using the top elements of the priority queues. When the lower bound outstrips the upper bound, the algorithm stops.

To reduce the upper bound as soon as possible, we employed a cost model. After the upper bound was calculated, we replaced the top element in each priority queue with the second element, and the \( n \) new bounds were called the cost for the replacement. The priority queue with the lowest cost pops the top element, and the candidates which contained the element were considered to be the next candidates to be added to the heap. For example, suppose the scoring function is \( (\exists x1 + 10) / \exists x2 \). In this case, we have two queues. Suppose that the priority of the front ones are 10 and 2, the priority of the second ones are 6 and 4. The current bound is \( (10 + 10)/2 = 10 \). If \( \exists x1 \) (10) is replaced with 6, the new bound is \( (6 + 10)/2 = 8 \). If \( \exists x2 \) (2) is replaced with 4, the bound is \( (10 + 10)/4 = 5 \). As a result, the costs of the first and the second replacements are 8 and 5, respectively. Thus, the first entity candidate of \( \exists x2 \) is popped, and the front of the queues are 10 and 4. Note that in the worst case, \( n-1 \) of the generated candidates are redundant candidates since every candidates can be generated by each priority queue.

First, the \( n \) best and \( n \) second-best entity candidates were generated by calling Algorithm 2 (line 6–16). Next, we chose the priority queue with the lowest cost (line 17), and the top element of the queue was considered to be the starting entity. For
each candidate \( c \) that contained the entity, if \( c \) was generated by other queues in prior iterations, this candidate was omitted (line 21). Otherwise, the candidate was added to the result heap, and the upper and lower bounds were updated (line 22–24). If the upper bound was not greater than the lower bound, the algorithm stopped (line 25–26).

The scoring function of \( Q_2 \) in Fig. 3 contains two variables, \(?y\) and \(?z\). Based on the structure filtering, the possible values of both \(?y\) and \(?z\) are 3.05E10, 3.13E11, 3.57E11, 6.75E11 and 1.71E13. In the basic method, if \(?z\) was chosen to be the starting variable, i.e., \(?y\) is fixed as 1.71E13, the score bound decreases following the path 1.71E13–3.05E10, 1.71E13–3.13E11, 1.71E13–3.57E11, 1.71E13–6.75E11 and 1.71E13–1.71E13. Note that the first four bounds are close to 1.71E13. In the optimized method, the first items of the two priority queues are 3.05E10 and 1.71E13. Thus, the first score bound is also 1.71E13–3.05E10. Then, due to 1.71E13 being far larger than the second value 6.75E11 in the queue, 1.71E13 is shifted and the next value is 6.75E11. The next bound is 6.75E11–3.05E10 = 6.44E11. Clearly, the score bound decreases faster in the optimized method.

7. Experimental results

7.1. Dataset and setup

7.1.1. Dataset and queries

7.1.1.1. Berlin SPARQL benchmark. The Berlin SPARQL Benchmark (BSBM [4]) has an RDF dataset generator and 12 SPARQL queries to evaluate the efficiency of the RDF query engines. The design of BSBM is based on an e-commerce use case in which a set of products are offered by different producers, and consumers post reviews about these products. The scale of the BSBM datasets can be set to arbitrary sizes by using the number of products as the scale factor. We generated five datasets with different sizes or evaluations, where the sizes of the datasets were 100k, 250k, 500k, 1M and 5M triples, respectively.

In BSBM datasets, the numerical values are the numerical properties of a product, the rating score of a product in a review, and the price of an offer. Unfortunately, no query in the BSBM query set contains top-\( k \) semantics. Alternatively, we employed the top-\( k \) query set (which contains eight queries) in [18] to evaluate the performance of our solution. In the eight queries, all measures were normalized to \([0, 1]\), and the measures were summed to get the score of a candidate. Table 1 shows the number of measures in different queries.

7.1.1.2. YAGO2. YAGO2 [14] is a large, real dataset, and the latest version of YAGO2 has more than 10 million entities and 440 million statements. The data in YAGO2 is based on Wikipedia, WordNet and GeoNames. By removing some statements that describe the date when a statement is extracted, or the URL where a statement is extracted from, we generated an RDF dataset from YAGO2. The condensed dataset had more than 10 million entities/classes and more than 180 million statements. Based on our dataset, we had 10,557,223 signature nodes.

In YAGO2, there are 22 numerical properties, where the properties describe the height and weight of a person, the page number of a book, the area of a geographic entity, etc. As far as we know, there is not a benchmark to the real datasets (YAGO, DBpedia and Freebase, etc.) for the top-\( k \) SPARQL query evaluation. As a result, we manually designed eight queries for our evaluation. Table 2 shows the measures count for each query.

7.1.2. Setup

The experiments were separated into two parts. In the first part, we used the BSBM datasets to evaluate the effectiveness and the efficiency of our solution. The state-of-the-art solution SPARQL-RANK was employed as a competitor. In the second part, we evaluated the performance of our solution on the large, natural dataset YAGO2. The SPARQL-RANK solution and a post-process baseline method were used as competitors.

All experiments were evaluated on a PC with an Intel Xeon E5645 and 16 GB main memory. The MS-tree’s default node capacities of the BSBM datasets and the YAGO2 dataset were set to 20 and 50, respectively. Since the response time of the different queries may fluctuate at large scales, we used the geometric mean as the “average” in the experiments, by default.

7.2. Evaluations on BSBM

In this experiment, we (1) evaluated the effect of the tree parameter, (2) reported the correlation between the running time and the node access, and (3) compared the efficiency of our solution and the state-of-the-art solution. The BSBM 5 M dataset was employed in all these evaluations, by default.

7.2.1. Tree node capacity

7.2.1.1. Off-line Cost. The off-line cost of the index construction is shown in Fig. 7. Fig. 7(a) shows the time cost to build the index, where the “FindBest” denotes the process that finds the best position of the incoming entity, and “Split” denotes the
The overall time cost of splitting the full nodes. The “Total” bar represents the overall time cost to construct the MS-tree index. The “FindBest” bar decreases, and the “Split” bar increases, as the node capacity increases.

Fig. 7(b) shows the tree-node number and the index size as the node capacity varies. Clearly, the node number decreases as the node capacity increases. In contrast, the index size first drops since the node number decreases faster than the node size increases, and then increases as the empty part of the node increases.

7.2.1.2. On-line cost. Larger node capacities may place numerous irrelevant tree nodes into the priority queues, thus decreasing the query’s performance. At the same time, a smaller node capacity may increase the tree height, and separate the results into different branches, which may also increase the search space. In this evaluation, the node capacities were set to be 10, 15, 20, 30, 50, 100, 150, respectively.

Fig. 8 shows the index tree height when the node capacity varies. Fig. 9 shows the average count of accessed nodes during the search, and the average time cost of each query. The query time cost is proportional to the count of data points involved during the search in most cases. With an increase of node capacity, the response time has no stable trend when \( k = 1 \) or \( k = 10 \), and the response time slightly increases when \( k \) is large (\( k = 100, 1000 \)).

7.2.2. Running time vs. accessed nodes

We show the number of accessed nodes and the query response time for different datasets in Fig. 10. Since the BSBM datasets are synthetic datasets, the distribution of different statements and the numerical values were fixed, making the number of accessed nodes nearly proportional to the dataset size, especially for the top-1 queries (Fig. 10(a)). When comparing Fig. 10(a) and (b), it is seen that the response time is almost proportional to the number of accessed nodes.

7.2.3. Optimization in multiple-variable case

In Section 6.2, we proposed two methods to deal with the top-\( k \) problem when the scoring function involved more than one variable. Fig. 11 shows the effect of the optimization procedure. The first method is called the basic method, and the second is called the optimized method. The optimized method outperformed the basic method in all situations. When \( k = 1 \) or \( k = 10 \), the time cost of the optimized method was much smaller than the basic method, and when \( k = 100 \) or \( k = 1000 \), the time cost of the two methods became similar, as the ratio of redundant candidates increased.
Fig. 8. Index-tree height vs. node capacity.

Fig. 9. On-line cost of the node capacity evaluations.

Fig. 10. Accessed nodes and time cost for different datasets.
Here, we evaluated the performance of the SPARQL-RANK [18] and our solution. Since the algebraic average was used in [18] to calculate the performance, we also used the algebraic average as the average method in this evaluation. We note that the SPARQL-RANK approach performed much better with a warm cache (when compared to the first execution). In Fig. 12, the SPARQL-RANK approach outperformed our solution significantly.

### Fig. 11. Effect of the upper bound optimization method.

### Fig. 12. Performance evaluation of the BSBM datasets.

**7.2.3. Performance evaluation**

Here, we evaluated the performance of the SPARQL-RANK [18] and our solution. Since the algebraic average was used in [18] to calculate the performance, we also used the algebraic average as the average method in this evaluation. We note that the SPARQL-RANK approach performed much better with a warm cache (when compared to the first execution). In Fig. 12,
the “U-Topk” bar denotes the average response time of our solution, and the “S-RANK (Cached)” bar and the “S-RANK (No Cache)” bar denotes the average response time costs of SPARQL-RANK running on a warm cache and a cold cache, respectively. In most cases our solution performed as well as the warm-cached SPARQL-RANK, and our solution outperformed the cold-cached SPARQL-RANK every time. The results show that for small datasets, our solution performed no worse than the join-based, state-of-the-art solution.

7.3. Performance report on YAGO2

The effectiveness and the efficiency of our solution were also evaluated with the real dataset, YAGO2. Since YAGO2 is built up based on entries from Wikipedia, the experimental results show that our approach can be used in practical applications.

The response time of answering the top-k queries on YAGO2 by employing different approaches is reported in Table 3. In Table 3, “U-Topk” denotes our MS-tree-based solution, “SRANK” denotes the SPARQL-RANK approach with a warm cache, and the “Baseline” denotes the post-process approach. The post-process approach first generated all of the candidates by employing the gStore [38] query engine, then sorted the candidates based on the scoring function, and finally, reported the best k candidates to be the top-k results. Additionally, we report the number of all possible candidates of each query in the second line of Table 3. The results show that our solution outperformed all other approaches by at least one order of magnitude in most cases.

8. Conclusion

In this paper, we investigated the top-k SPARQL queries in RDF data that consist of three components: a SPARQL query pattern, a scoring function and a result number limit k. Given a top-k SPARQL query, k sub-graphs in the RDF graph with the highest k scores that satisfy the given query pattern were output.

In order to effectively and efficiently evaluate top-k SPARQL queries, we introduced a novel tree index called an MS-tree. Given an RDF dataset, we encoded the entities and the classes into bit strings, which were considered as leaves in the MS-tree. The node also maintained the maximum and minimum values of each numerical property. The MS-tree was built based on the nodes, where the father node followed two rules: (1) the bit string of the father node equaled the union of the children’s bit strings, and (2) the maximum (minimum) value of the father node equaled the maximal maximum (minimal minimum) value among the child nodes. The MS-tree helped us construct the candidates in an appropriate order.

The MS-tree-based top-k SPARQL query processing can be seen as a threshold algorithm that consists of two stages. In the first stage, tree nodes were accessed top down, appropriately order by their numerical property values that were involved in the scoring function. When a leaf node was accessed, it was most likely to build a candidate from the leaf node. In the second stage, the graph exploration method was employed to obtain candidates that satisfied the query pattern. When we guaranteed that top-k results were generated, the process immediately stopped. When the scoring function involved more than one variable, we proposed a cost-model-based optimization method to improve the query performance. The optimized method iteratively chose a variable for which the next entity candidate maximized the decrease of threshold. The experimental results based on both synthetic and real datasets demonstrated the effectiveness and efficiency of our solution. Our method significantly outperformed the state-of-the-art approach when applied to large datasets.

In the current MS-tree model, the storage cost of an MS-tree significantly depends on the number of numerical properties in RDF data. Since the scale of an RDF dataset can be extremely huge, and there can be far more than tens of numerical properties, our method may face the scalability problem. In the future, we plan to improve the structure of MS-tree to reduce the storage cost, and utilize cloud platforms to make our method applicable for huge RDF datasets.
Acknowledgment

This work was supported by National High-Tech R&D Program (No. 2015AA015402) and NSFC under Grant No. 61370055, 61272344.

References