A Cost-Driven Top-K Queries Optimization Approach on Federated RDF Systems

Ningchao Ge, Zheng Qin, Peng Peng, Mingdao Li, Lei Zou, and Keqin Li, Fellow, IEEE

Abstract—RDF (Resource Description Framework) is a model widely used to construct knowledge bases, while SPARQL (SPARQL Protocol and RDF Query Language) is the standardized structured query language to manipulate RDF data. Recently, many data providers have published their RDF datasets in their own autonomous sites and provided SPARQL query interfaces, called *RDF sources*. In order to integrate multiple RDF sources, researchers put forward the federated RDF system to support the federated SPARQL queries. However, existing studies can only support efficient basic queries but not top-k queries. Toward this end, we propose a cost-driven top-k queries optimization approach in federated RDF systems, which can support both top-k queries for single variable ordering and expression ordering. Firstly, we propose an optimized query decomposition method to decompose the federated query into multiple subqueries. Secondly, while considering the top-k operator, we propose a cost model to evaluate the query cost and join cost of subqueries. The optimal query plan can be obtained by the costed-based query plan generation algorithm. Finally, combined with the characteristics of top-k queries, an incremental query plan execution strategy is developed to minimize the total query cost. Experimental results show that the proposed method is effective, efficient and scalable.

Index Terms—Federated RDF systems, Query optimization, SPARQL, Top-k.

1 Introduction

As the standard organization model for Web of Linked Data, RDF (*R*esource *D*escription *F*ramework) [9] has been widely used in various fields. RDF represents data as a triple in the form of <subject, predicate, object> or <subject, attribute, value>. To manipulate RDF data, the standardized structured query language, SPARQL (*S*PARQL *P*rotocol and *R*DF *Q*uery *L*anguage) [29], is released by W3C (*W*orld *W*ide *W*eb *C*onsortium). In recent years, an increasing number of data providers have published their datasets using the RDF model. These datasets are often maintained at their own sites, which provide the SPARQL interfaces to support users to submit SPARQL queries. An autonomous site with a SPARQL interface is called an *RDF source* in this paper.

The federated RDF systems [8], [23], [33] are put forward to integrate multiple RDF sources. Up to now, many federated RDF systems [4], [19], [24] have been developed and implemented the federated queries. In a federated RDF system, different RDF sources cannot communicate with each other directly. Thus, it is desired to develop a control site to manipulate these RDF autonomous sources. The famous federal RDF systems include the biological information federal RDF systems with 57 RDF sources issued by the European Molecular Biology Laboratory[1].

- *Ningchao Ge, Zheng Qin, Peng Peng and Mingdao Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, 410082.*
  E-mail: {ningchaoge,zqin,hnu16pp,limingdao}@hnu.edu.cn
- *Lei Zou is with Peking University, Beijing, China, 100080.*
  E-mail: zoulei@pku.edu.cn
- *Keqin Li is with Department of Computer Science, State University of New York, USA, NY 12561.*
  E-mail: lik@newpaltz.edu

1. https://www.ebi.ac.uk/rdf/services/sparql

Since SPARQL is the query language designed for centralized RDF system, it cannot be directly executed on federated RDF system. In a general method, for a federated SPARQL query submitted by user, it is first decomposed into multiple subqueries, which can be executed on the corresponding RDF sources separately. Then, the results of subqueries are joined together to obtain the final result. Now, many works have been put forward to optimize the SPARQL processing in federated RDF systems, but most of them mainly focus on implementing and optimizing the basic queries [13], [24], [32], [34] in federated SPARQL systems. Few of them discuss how to evaluate the top-k queries, which are queries returning k answers with the highest rank order by utilizing a ranking function. There are two common types of ranking functions, namely, single variable and expression. For SPARQL, top-k queries can be expressed by including the ORDER BY and LIMIT clauses, as shown in Fig. 1 and Fig. 2.

Example 1.1. A top-k query example with single variable ordering is shown in Fig. 1.

```
SELECT * WHERE{?role swc:heldBy ?writer.
  ?writer foaf:based_near ?geonameplace.
  ?geonameplace dp:capital ?capital.
  ?geonameplace dp:foundingDate ?foundingDate.
  ?place g:countryCode ?countryCode.
  ?place g:population ?population.
  ?place g:long ?longitude.
  ?place g:lat ?latitude.
  ?place owl:sameAs ?geonameplace.
} ORDER BY DESC (?population) LIMIT 3
```

Fig. 1: A top-k query example with single variable ordering: find the top three neighboring cities and its attributes with the largest population for writers.

Example 1.2. A top-k query example with expression ordering is shown in Fig. 2.

```
SELECT * WHERE{
    ?place  org:type <http://dbpedia.org/ontology/Place>.
    ?place dbpedia:areaLand ?areaLand.
    ?place dbpedia:areaWater ?areaWater.
    ?place owl:sameAs ?nytplace.
    ?nytplace nyt:associated_article_count ?newscount.
}ORDER BY DESC (?areaLand+?areaWater) LIMIT 3
```

Fig. 2: A top-k query example with expression ordering: find the top three places and the count of its news with the largest sum of land area and sea area.

Compared with the basic query, top-k query can quickly provide users with the most concerned information. Especially, in the federated RDF system with a large amount of data, efficient top-k query is more important. There exist a few previous work [13], [32] can support top-k federated SPARQL query, but they only support single variable ordering queries in a materialize-then-sort processing scheme, which computes all the matching solutions (e.g. thousands) even if only a limited number k (e.g. ten) are requested. It leads to the huge query cost and network communication overhead, especially for large federated RDF systems.

To support efficient top-k SPARQL queries for the single variable ordering on federated RDF systems, an effective top-k query optimization scheme is developed in the conference version of this paper [12]. This paper extends our previous work, and adopts a new optimization method to support both single variable ordering and expression ordering top-k SPARQL queries.

In summary, the proposed scheme has following mainly contributions:

- We propose a query decomposition and source selection optimization strategy, which allow to merge some triple patterns with the same multiple sources into one subquery. It can improve query efficiency by reducing the number of remote requests.
- We construct a cost model and design a cost-driven optimal query plan generation algorithm with dynamic programming, which can optimize the join order by controlling the execution order and execution strategy (serial and parallel) of top-k queries.
- We propose an incremental query plan execution strategy to support efficient evaluation of top-k queries. The strategy can effectively improve the query efficiency by avoiding many unnecessary results.
- We implement a federated RDF system, named `FedTopKPro`, which can support both top-k queries for single variable ordering and expression ordering. The experimental results on `FedTopKPro` show that our method is much better than previous works with effectiveness and total run time.

2 Related Work

Top-k Query Optimization. Top-k query optimization is a practical research, which has been well studied in relational databases. For SPARQL, it is expressed by ORDER BY and LIMIT clauses. The existing research studies [6], [22], [26], [35] mainly focus on top-k query optimization over centralized RDF system. Bozzon et al. [6], [22] improved the efficiency of top-k query on RDF graph by extending the SPARQL algebra and SPARQL-RANK. Wang et al. [35] utilized the graph-exploration to further improve the query efficiency instead of the join method. The works of Wang et al. [37] and Yang et al. [38] are aimed at specific query types (such as star query). Ihm et al. [18] improved the query efficiency of top-k by building partition index. Jiang et al. [20] quickly obtained query results by adopting heuristic pruning and incremental algorithm.

SPARQL Query Evaluation in Federated RDF Systems. According to the standard of W3C, SPARQL is only valid for centralized RDF system. It can be running over federated RDF system with some extra design. At present, there are many federation RDF system [28], [30], [36] which can support SPARQL basic queries. Harth et al. [14] and Prasser et al. [28] converted a SPARQL query into a minimum bounding boxes connection by using an index similar to the R-Tree [7], named QTree, and then the RDF sources of each triple in the SPARQL query can be obtained. DARQ [30] obtained the relevant RDF sources according to an index called service description, which describes which triple patterns can be answered. Different from DARQ [30], HiBISCuS [32] constructed the query graph into a directed labeled hypergraph in the stage of determining RDF sources, which further reduces the number of candidate RDF sources for each subquery. SPLENDID [13] built an inverted index based on the VOID (vocabulary of interlinked datasets) of each RDF source. FMQO [25] further discussed how to optimize multiple queries evaluation by rewriting the set of input queries into a smaller set of rewritten queries. FedX [34] transfered all triples in the query statement to all RDF sources, and determine the relevant RDF sources through ASK in SPARQL syntax.

These above methods were only efficient for basic queries, but not for top-k queries. The conference version of this paper, FedTopK [12], implemented a federated RDF system to support efficient top-k SPARQL queries for the single variable ordering, but it does not consider expression ordering top-k SPARQL queries. In this paper, the optimization and expansion are made to further improve the efficiency of top-k query with single variable ordering based on FedTopK. In addition, the incremental query scheme proposed in this paper can also effectively support top-k query with expression ordering.

3 Background

In Section 1, we introduce the information of federated RDF systems. The Web resources are expressed by unique identification IDs in RDF, which are called Internationalized Resource Identifiers (IRIs). SPARQL is a standardized query language used to query RDF datasets. In the context of federated RDF system, we extend the idea of authoritative source in [17]. In order to facilitate readers' understanding of the follow-up content, this section will give definitions of related terms and research issues in this paper. The

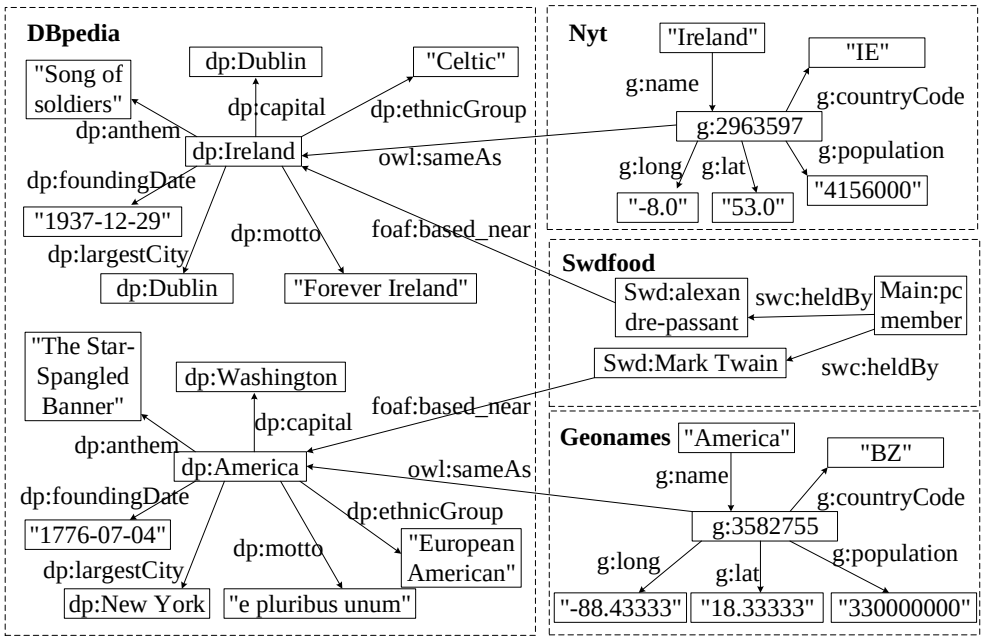


Fig. 3: Example RDF graph over federated RDF systems.

definitions of related terms are similar as found in [5], [15], [16], [24], [27].

Definition 3.1. **(RDF Graph)**. An RDF dateset is a set of triples. Triple patterns are the description of subjects, predicates and objects, can be expressed as: $\mathcal{TP} = (\mathcal{I} \cup \mathcal{N}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{N} \cup \mathcal{L})$. Among that, $\mathcal{I}$ is the set of IRIs, $\mathcal{N}$ is the set of blank elements and $\mathcal{L}$ the set of literals. An RDF graph is a graphical description of triple patterns. In RDF graph, the vertexes are transformed from subjects and objects of $\mathcal{TP}$ and predicates are the label of edges.

Definition 3.2. **(Federated RDF System)** A *federated RDF system* can be expressed as $F = (S, g, d)$, where (1) S is a set of source sites that can be obtained by looking up IRIs in an implementation of F; (2) $g : S \rightarrow 2^{\mathcal{TP}}$ is a mapping that associates each source with a subgraph of RDF graph $\mathcal{TP}$; and (3) $d : \mathcal{I} \rightarrow S$ is a partial, surjective mapping that models the fact that looking up IRI of resource u matches in the retrieval of the source represented by $d(V) \in S$. $d(V)$ is called the *host* source of V, and is unique for a given URL of vertex V.

Example 3.1. Fig. 3 shows a federated RDF system as a graph distributed among four different sources. Given a resource with the IRI "$dp{:}Ireland$", where "dp" is abbreviation of "DBpedia". d("$dp{:}Ireland$")=DBpedia, this means that "$dp{:}Ireland$" is dereferenced by the host DBpedia.

Definition 3.3. **(Basic Graph Pattern)** SPARQL is a structured query language to manage RDF dataset, and the basic graph pattern (BGP) is its basic query block. We use $\mathcal{V}$ to express the variables set of a SPARQL query, and those variables all bind to RDF triple patterns from $\mathcal{I} \cup \mathcal{N} \cup \mathcal{L}$. *triple patterns* are consisted with some triples $ts \in (\mathcal{I} \cup \mathcal{N}) \times (\mathcal{I} \cup \mathcal{N}) \times (\mathcal{I} \cup \mathcal{N} \cup \mathcal{L})$. For convenience of explanation, we neglect the blank element of each triple pattern. Let $\mathcal{TP}$ be the set of all triple patterns. Then, a basic graph pattern (BGP) is a set $Q \subset \mathcal{TP}$, and the set of queries is $\mathcal{Q} \subset 2^{\mathcal{TP}}$.

In this paper, we support top-k queries in SPARQL with an ORDER BY clause that can be formulated as a ranking criterion on a variable.

Definition 3.4. **(Top-k SPARQL Query)**. A top-k SPARQL query can be expressed as: $TSQ =< Q, f, k >$, where Q is a BGP pattern, f is the ranking function, including single variable ordering and express ordering. And k is the maximum number of results.

Figure 1 shows the single variable ordering example top-k SPARQL query, where f is the single variable $?population$ and k is 3. Figure 2 shows the express ordering example top-k SPARQL query, where $(?areaLand + ?areaWater)$ is the express f and k is 3. A match of BGP Q may involve different RDF sources over a federated RDF system. Specifically, a match distributed over a set of sources $S' \subseteq S$ is a function μ from variables in Q to RDF terms in $\bigcup_{\tau \in S'} g(\tau)$.

Definition 3.5. **(Match of Top-k SPARQL Query over Federated RDF System)** Firstly, we need to denote two functions. The first function is $\mu : \mathcal{V} \rightarrow \mathcal{I} \cup \mathcal{N} \cup \mathcal{L}$. It is a *mapping* μ from $\mathcal{V}$ to $\mathcal{I} \cup \mathcal{N} \cup \mathcal{L}$. For a triple pattern t of a SPARQL query, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. Secondly, for a federated RDF system $F = (S, g, d)$ and a BGP Q. Given $S' \subseteq S$, a *mapping* μ is said to be a *match* of Q if and only if $\mu(tp) \in \bigcup_{\tau \in S'} g(\tau)$ for each triple pattern e in Q. Finally, the match result of a top-k SPARQL query $TSQ =< Q, f, k >$ is a no-more-than-k-sized list of matches of Q, with the highest rank order by the ranking function f.

The problem to be studied in this paper is defined as follows:

Given a federated RDF system $F = (S, g, d)$ and a top-k SPARQL query $TSQ =< Q, f, k >$, the problem to be researched is to obtain the query result of TSQ.

4 Framework

As shown in Fig. 4, the framework of top-k query processing mainly consists of three parts: *query decomposition and source localization*, *cost-driven query plan generation* and *incremental query plan execution*. For a top-k SPARQL query submitted by a user, we propose an auxiliary index to decompose the query into subqueries according to its RDF sources (see Section 5). Then, the optimal query plan can be obtained by a cost-driven query plan generation algorithm with dynamic programming (see Section 6). Finally, according to the query plan, these decomposed subqueries are sequentially sent to their corresponding RDF sources for execution in serial or parallel. Among that, combined with the characteristics of top-k query, an incremental query plan execution optimizations strategy is carried out. It can further reduce the cost of subqueries execution to ensure that the overall cost and network communication are minimized (see Section 7).

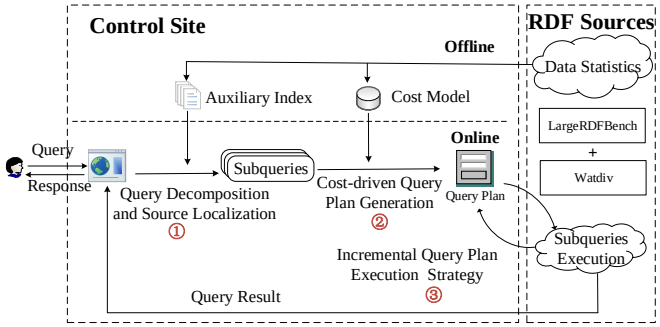


Fig. 4: The scheme of top-k query processing in federated RDF systems.

5 Query Decomposition and Source Localization

SPARQL is the query language designed for centralized RDF system, and cannot be directly executed on federated RDF system. In order to evaluate a SPARQL query over federated RDF systems, it needs to be decomposed into a series of subqueries, which can be executed on single RDF source. Basically, each triple pattern in the BGP of a SPARQL query maps to a set of RDF sources based on the constant values of subject, predicate and object. If a triple pattern is variables-only, it maps to all RDF sources in the federated RDF system. For the top-k SPARQL query $Example$ 1.1, the mapping between triple patterns and its relevant RDF sources is shown in Fig. 5.

The basic query decomposition and source localization methods combine the triple patterns with the same single RDF source into one subquery. For the example query in Fig. 1, it should be decomposed into seven subqueries, as shown in Figure 6. The subquery q_1 is composed of the triple patterns $<?role\ swc\ :\ heldBy\ ?writer\ >$ and $<?writer\ foaf\ :\ based\_near\ ?geonameplace\ >$, because

Top-K SPARQL Query	RDF Sources
select * where {?role swc:heldBy ?writer.	Swdfood
?writer foaf:based_near ?geonameplace.	Swdfood
?geonameplace dp:capital ?capital.	DBpedia
?geonameplace dp:foundingDate ?foundingDate.	DBpedia
?place g:countryCode ?countryCode.	Geonames, Nyt
?place g:population ?population.	Geonames, Nyt
?place g:long ?longitude.	Geonames, Nyt, DBpedia
?place g:lat ?latitude.	Geonames, Nyt, DBpedia
?place owl:sameAs ?geonameplace.	Geonames, DBpedia, Swdfood, Nyt
} order by desc (?population) limit 3	

Fig. 5: Relevant RDF sources for each triple pattern in top-k query $Example$ 1.1.

they have the same single RDF source $\{Swdfood\}$. Note that, because the relevant RDF sources of $<?place\ g\ :\ long\ ?longitude\ >$ and $<?place\ g\ :\ lat\ ?latitude\ >$ are not single, they cannot be merged into one subquery even if their RDF sources are the same.

q_1@{swdfood}select * where { ?role swc:heldBy ?writer. ?writer foaf:based_near ?geonameplace}	q_4@{gnames, nyt}select * where { ?place geonames:population ?population}
q_2@{dbpedia}select * where { ?geonameplace dbpedia:capital ?capital. ?geonameplace dbpedia:foundingDate ?foundingDate.}	q_5@{gnames, nyt, dbpedia} select * where {?place geo:long ?longitude}
q_3@{gnames, nyt}select * where { ?place geonames:countryCode ?countryCode}	q_6@{gnames, nyt, dbpedia} select * where {?place geo:lat ?latitude}
	q_7@{gnames, dbpedia, swdfood, nyt} select * where {?place owl:sameAs ?geonameplace}

Fig. 6: Basic query decomposition and source localization result for $Example$ 1.1.

However, triple patterns $<?place\ g\ :\ long\ ?longitude\ >$ and $<?place\ g\ :\ lat\ ?latitude\ >$ can be merged into one subquery in reality, because they always appear in pairs over each RDF source. The number of subqueries affects the number of remote accesses, which will take up a lot of time overhead in distributed environment. Thus, we propose an index-based optimized query decomposition and source localiztion optimization strategy to reduce the number of subqueries, which allow to merge some triple patterns with the same multi-sources into one subquery.

There are two stages during our optimized query decomposition and source localization for a top-k SPARQL query. In the first stage, all the predicates of each RDF source are maintained as the meta data. Then an auxiliary index can be built by utilizing meta data. The auxiliary index can be used to determine whether triple patterns with the same multi-sources can be merged into one subquery. In the second stage, for each triple pattern of a top-k SPARQL query: if its predicate is a constant, its RDF sources can be located by meta data. Otherwise, all the RDF sources of federated RDF system are its RDF sources. Furthermore, if the subject or object of the triple pattern is a constant, we can use *ASK* query of SPARQL to prune irrelevant RDF sources. For triple patterns with the same multi-sources, it needs to judge whether they can be merged into one subquery by auxiliary index. For other triple patterns, it adopts the basic query decomposition method.

As the core of the optimization method, the first stage can also be called the auxiliary index building stage. We need to get meta data at the control site as key-value pairs $< P,\ S >$ first, where S is the RDF sources set of a constant predicate P. The time complexity of generating the meta data is $O(|E|)$, which E represent the set of edges in the federated RDF system. Then, the auxiliary index is built by utilizing the meta data, as shown in Algorithm 1.

Algorithm 1: Auxiliary Index Generation

Input: The meta data $Map = \{< P,\ S >\}$
Output: Auxiliary index Mul_merge
1 **for** $i = 1$ *to* $|Map|$ **do**
2 **if** $|S_i| == 1$ **then**
3 $Map.remove(P_i, S_i)$;
4 Initialize an empty query result map Map_R;
5 **for** $i = 1$ *to* $|Map| - 1$ **do**
6 $getResult(P_i, Map\_R)$;
7 **for** $j = i + 1$ *to* $|Map|$ **do**
8 **if** $S_i == S_j$ **then**
9 $getResult(P_j, Map\_R)$;
10 $R_{tmp} = query(P_i \circ P_j)$;
11 **if** $R_i \ltimes R_j == R_{tmp}$ **then**
12 $Mul\_merge.put(P_i \circ P_j,\ true)$;
13 **else**
14 $Mul\_merge.put(P_i \circ P_j,\ false)$;
15 Return mul_merge

The key-value pairs with single RDF source are removed from meta data (Lines 1-3 in Algorithm 1). Generally, the order of magnitude of remaining key-value pairs with multi-sources is in the hundred, and the number of predicates pairs $< P_i, P_j >$ with same multi-sources is not much, donated as M. For a predicate pair $< P_i, P_j >$ with the same multi-sources, we obtain the query results of P_i, P_j and $P_i \circ P_j$ respectively (Lines 4-10 in Algorithm 1). Finally, the auxiliary index is obtained by comparing the join results of P_i and P_j with query result of $P_i \circ P_j$ (Lines 11-14 in Algorithm 1). Among them, we utilize the strategy of space for time to reduce the time cost caused by multiple queries on the same predicate, as shown in Algorithm 2. Let C and J express the execution times of function $query()$ and joining operation respectively, the time complexity of Algorithm 1 is $O(M \times (2C + J))$.

Algorithm 2: Function $getResult(P, Map\_R)$

1 **if** $Map\_R.getKeySet().contain(P)$ **then**
2 $R = Map\_R.get(P)$;
3 **else**
4 $R = query(P)$;
5 $Map\_R.put(P, R)$;
6 Return R

Generally, the federated RDF system consists of published RDF datasets. A version of RDF dataset will not change unless the version is updated. When a new version of an RDF dataset is released, rigorous organizations usually publish data update logs. In this case, we can update the auxiliary index incrementally by analyzing the data update log. In the worst case, if the index needs to be reconstructed when the update log cannot be obtained. Because the index construction is completed in the offline stage before the system is used. this is acceptable in the process of federated RDF system upgrade.

For the top-k SPARQL query Example 1.1, five subqueries can be obtained by utilizing the proposed method, as shown in Fig. 7. We use $\mathcal{Q}$ to express the set of subqueries.

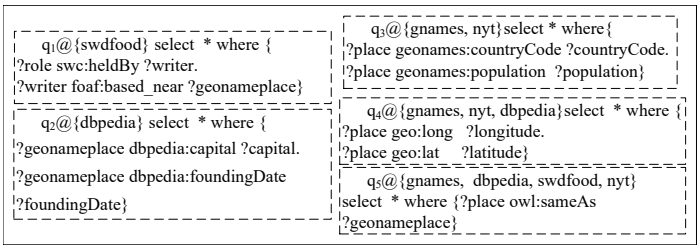


Fig. 7: Optimized query decomposition and source localization result for $Example$ 1.1.

6 Cost-driven Query Plan Generation

There are huge differences of query overhead between different subqueries execution orders. A subqueries execution order is called a query plan. In order to evaluate the query cost of a query plan, we designed a cost model. The query cost and join cost of each subquery can be evaluated by this model.

6.1 Cost Model Design

According to the definition of distributed database system books [11], the total execution time for a database query can be expressed as follows:

$$\begin{aligned} Total\_time =& T_{CPU} \times \#insts + T_{I/O} \times \#I/Os \\ &+ T_{MSG} \times \#msgs + T_{TR} \times \#bytes \end{aligned} \tag{1}$$

The first two terms of the Equation (1) express the query execution times of server. They depend on the performance of the server. The data transmission time is determined by the two last terms. In the distributed database system, the data communication often occupies a large amount of overhead and it is the goal that needs to be optimized. In order to evaluate and optimize that communication overhead, this paper designs a cost model.

For a federated RDF system with m RDF sources, the set of RDF sources can be represented as $S = \{s_1, s_2, s_3, ..., s_m\}$. For each RDF source s_i of S, the set of its predicates is denoted as $P_i = \{P_{i1}, P_{i2}, P_{i3}, ...P_{in}\}$, where n is the number of distinct predicates. In RDF source s_i, the number of triple patterns containing the predicate P_{ij} can be expressed as follows:

$$Sum(P_{ij}) = card(\sigma_{P=P_{ij}}(s_i)) \tag{2}$$

where σ represents the selection operation of database, and $card()$ is a counting function. Among that, the number of distinct subjects and objects of triple patterns containing the predicate P_{ij} is denoted as follows:

$$Subject(P_{ij}) = card(dom[\pi_S(\sigma_{P=P_{ij}}(s_i))]) \tag{3}$$

$$Object(P_{ij}) = card(dom[\pi_O(\sigma_{P=P_{ij}}(s_i))]) \tag{4}$$

where π represents the projection operation of database, and $dom()$ is a deduplication function. On this basic, we define

the front join factor and rear join factor for the predicate P_{ij}, donated as $\beta_S(P_{ij})$ and $\beta_O(P_{ij})$. They can be calculated as follows:

$$\beta_S(P_{ij}) = \frac{Sum(P_{ij})}{Subject(P_{ij})} \tag{5}$$

$$\beta_O(P_{ij}) = \frac{Sum(P_{ij})}{Object(P_{ij})} \tag{6}$$

The above five parameters will be calculated statistically in the offline stage to form the meta data of the cost model. They need to calculate only once until the RDF data is updated. For a subquery with two triple patterns, their predicates are P_1 and P_2 respectively, if they have no common vertex, then the query cost of the subquery is as follows:

$$cost(q) = Sum(P_1) \times Sum(P_2) \tag{7}$$

Otherwise, there are three types of connection: $subject - subject$, $subject - object$ and $object - object$ between the two triple patterns. Here we assume that the connection type is $subject - subject$, the query cost of the subquery can be calculated as follows:

$$cost(q) = \beta_S(P_1) \times \beta_S(P_2) \times Min\left\{\frac{Sum(P_1)}{\beta_S(P_1)}, \frac{Sum(P_2)}{\beta_S(P_2)}\right\} \tag{8}$$

For other connection types, it is only necessary to change $\beta_S(P)$ in the Equation (8) into $\beta_O(P)$. All the query costs of subquery collection $\mathcal{Q}$ will be calculated by Equation (7) and (8).

For two subqueries q_1 and q_2, the query cost of them are $cost(q_1)$ and $cost(q_2)$ respectively. If there is no common column between the query result of them, the join cost of their query result is as follows:

$$cost(q_1 \bowtie q_2) = cost(q_1) \times cost(q_2) \tag{9}$$

Otherwise, their execution order determines the join cost of their query result. Because SPARQL has a common characteristic, which allows adding VALUES clause after the current query to narrow the matching range of subgraph. The content of VALUES clause is the results of the previous query. We assume that the q_1 is executed first, and the common column between the query result of them is the subject of a triple pattern P in q_2. The join cost of their query result is as follows:

$$cost(q_1 \bowtie q_2) = cost(q_1) \times \beta_S(P) \tag{10}$$

If the common column between the query result of them is the object of the triple pattern P, it needs to change $\beta_S(P)$ in the Equation (10) into $\beta_O(P)$. Similarly, if the q_2 is executed first, and the common column between the query result of them is the subject of a triple pattern P in q_1. The join cost of their query result is as follows:

$$cost(q_2 \bowtie q_1) = cost(q_2) \times \beta_S(P) \tag{11}$$

6.2 Optimal Query Plan Generation

A query plan represents an execution order and execution mode (serial or parallel) of subqueries $\mathcal{Q}$. There are huge differences in query efficiency between different query plans. Given a subquery set $\mathcal{Q}$, how to find the optimal query plan which can minimize the query cost. This problem has been solved well in relational database [11], and it is a typical dynamic programming problem.

Therefore, based on the cost model in Subsection 6.1, we design an optimal query plan generation algorithm with dynamic programming as shown in Algorithm 3. If the cost of optimal query plan with i subqueries is expressed by $C[dp[i]]$, and its recurrence is as follows:

$$\begin{aligned} C[dp[i]] = Min\{&C[dp[i-2] \bowtie (q_l \bowtie q_i)], \\ &C[(q_i \bowtie q_r) \bowtie dp[i-2]]\} \end{aligned} \tag{12}$$

Algorithm 3: Cost-Driven Optimal Query Plan Generation Algorithm

Input: A set of subqueries $\mathcal{Q}$
Output: A query plan QP, which owning the min total cost

```
1  Initialize an empty hash map dp;
2  Initialize an empty set QP and TC(total cost) =
     Double.MAX_VALUES;
3  for i = 1 to |Q| do
4      Q' = Q − {q_i}, S = {q_i};
5      while Q' ≠ ∅ do
6          Q_j = Select(Q', S);//See Algorithm 4
7          while Q_j exists do
8              S = S ⋃ {Q_j}, Q' = Q − {Q_j};
9              if !dp.keyset().contains(S) then
10                 SC = cost(S), dp.put(S, SC);
11                 if SC > TC then
12                     Go to the next repetition of the
                       foor-loop;
13             Q_j = Select(Q', S);
14     if dp.get(S) < TC then
15         TC = SC, QP = S;
16 Return QP
```

where q_l and q_r are subqueries in $dp[i-1]$, which can be left joint and right joint with q_i, respectively. Among that, $C[dp[1]] = cost(q_1)$, $C[dp[2]] = min\{cost(q_1 \bowtie q_2), cost(q_2 \bowtie q_1)\}$. The time complexity of Algorithm 3 is $O(n^3)$, where n is the size of the subqueries set $\mathcal{Q}$.

Algorithm 4: Function $Select(\mathcal{Q}, S)$

```
1  Initialize an empty set R;
2  // Get result columns collection of subqueries S.
3  Column_S = getColumns(S);
4  for i = 1 to |Q| do
5      // Get result columns collection of subquery q_i.
6      Column_q_i = getColumn(q_i);
7      if Column_q_i.retainAll(Column_S).size() > 0
         then
8          R.add(q_i);
9  Return R
```

Example 6.1. As shown in Fig. 7, the set of subqueries for the input query, $\mathcal{Q}$, contain five elements from q_1 to q_5. The optimal query plan corresponds to S when $i = 2$ in line 3 of Algorithm 3. Firstly, $\mathcal{Q}' = \mathcal{Q} - \{q_2\}$, $S = \{q_2\}$ in line 4. Then, the subquery $\{q_1, q_5\}$ is selected as Q_j in line 6, because the result columns of

them can intersect with result columns of S, as shown in Algorithm 4. The cost SC for $S = \{\{q_2\}, \{q_1, q_5\}\}$ can be calculated by the Equation (10) and (11), and put $< S,\ SC >$ to the hash map dp in line 10. If the cost of the current partial query plan SC is bigger than the cost of an overall query plan generated before, it means that the current query plan is not optimal, and the loop should be out in line 12. Otherwise, continue the above operations until $Q' = \varnothing$ in line 5. Finally, the optimal query plan of the top-k query Example 1.1 can be obtained as: $QP = \{\{q_2\}, \{q_1,\ q_5\}, \{q_3,\ q_4\}\}$. It represents the execution order and execution model as shown in Fig. 8.

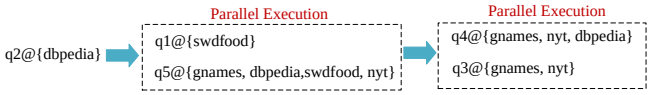


Fig. 8: The view of query plan QP.

7 Incremental Query Plan Execution

The query plan generated by Algorithm 3 can minimize the query cost when querying all the results that satisfy the triple patterns condition. However, as we all know, for top-k queries, an efficient query approach does not need to get all the intermediate results that meet the triple patterns condition. Therefore, we put forward to an incremental query plan execution optimization strategy to further improve the query efficiency of a top-k query.

7.1 Execution for Single Variable Ordering Top-k Query

For the query plan QP of single variable ordering example top-3 query in Example 1.1, we assume that its query execution plan can be shown in Fig. 9. Before the final result is selected, all the results of each subquery need to be queried, even if only top three results need to be returned to users.

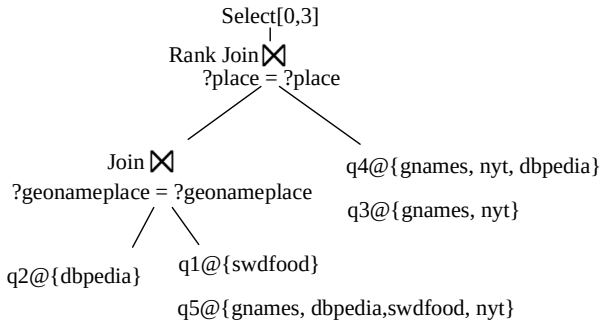


Fig. 9: The basic execution process of query plan QP.

In order to reduce the unnecessary intermediate results, we adopt an incremental query plan execution method, and the query plan execution process of this method is shown in Fig. 10. Its idea is to execute the subquery with ordering variable first under the same priority. The rank join will be done for the current results, and then some pieces of data are incrementally selected from the ordered results as the VALUES clause of the subsequent subqueries. During the incremental selection process, we assume that the data is evenly distributed.

Algorithm 5 gives the detail incremental execution algorithm of query plan. In the first stage, the query executor

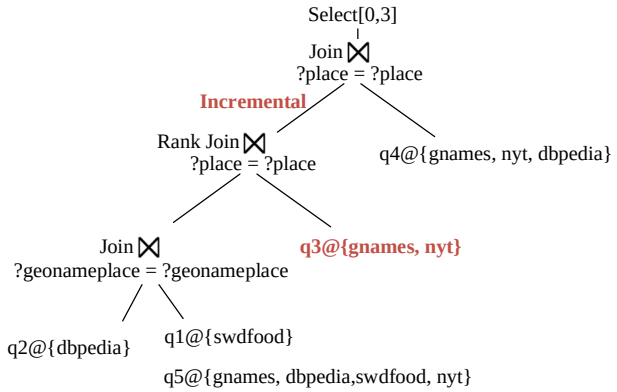


Fig. 10: The incremental execution process of query plan QP.

Algorithm 5: Incremental Execution Algorithm of Query Plan

Input: A query plan QP
Output: The results R of the query plan QP

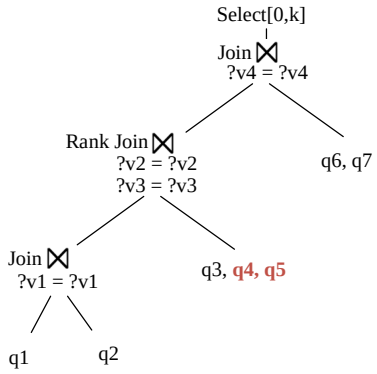
```
1  Initialize an empty result set R;
2  Initialize INDEX = 0, COUNT = 0, M = 0, N = 0, SUM = 0;
3  for i = 0 to |QP| do
4      for j = 0 to |QP_i| do
5          Execute subquery QP_ij in parallel, update set R;
6          if QP_ij contains the ordering variable then
7              Backup current query status as T;
8              SUM = |R|, INDEX = i, M = K, N = 1;
9              Remove QP_ij from QP_i;
10             Sort and select the first M results as the current result;
11         Set the values clause of the next subquery;
12 COUNT = |R|;
13 while COUNT < K and N × M < SUM do
14     Restore query status T;
15     if COUNT > 0 then
16         N = K/COUNT;
17     else
18         N = 2 × N;
19     M = M × N;
20     Select the first M results as the current result;
21     for i = INDEX to |QP| do
22         for j = 0 to |QP_i| do
23             Execute subquery QP_ij in parallel, update set R;
24             Set the values clause of the next subquery;
25     COUNT = |R|;
26 Return R
```

executes one round for all subqueries according to the query plan (lines 3 to 11). For the query plan QP in Example 6.1, the subquery q_2 is executed first. Then, subqueries q_1 and q_5 is executed with a values clause that generated from the result of q_2 in parallel, and so on. Among them, the subquery with ordering variable will be executed first

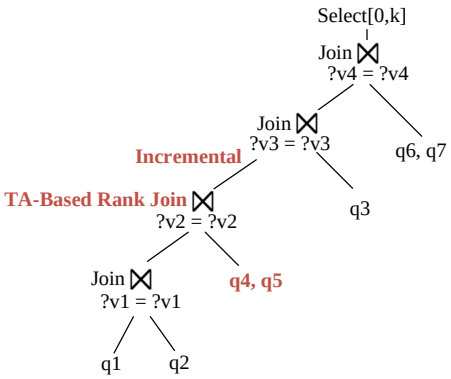
under the same priority. Then, the current query results are sorted immediately (lines 6 to 10). For QP, the subquery with ordering variable, q_3, is executed before q_4. Let SUM represent the number of query results after q_3 is executed (line 8), and let $COUNT$ represent the number of results after the first stage query (line 12). While $COUNT < k$ and $M < SUM$, it means that selecting M pieces of the sorting results is not enough to generate the final top-k results. Then, the query executor enter the second stage for the loop query (lines 13 to 25) until the query end condition is met.

7.2 Execution for Expression Ordering Top-k Query

The expressions are usually the addition and subtraction of several variables in expression ordering top-k queries. To deal with the problem of ordering the calculation results after addition and subtraction, there is a classical threshold accepting algorithm [10]. On this basic, we propose a TA-based rank join method to deal the intermediate results that generating by subqueries with variables in expression. For an expression ordering top-k query, the expression usually contains two or more variables. According to the distribution of variables, the basic execution process of query plan can be divided into three categories: (1) The ordering variables are distributed in subqueries set with the same priority. (2) The ordering variables are distributed in subqueries sets with adjacent priorities. (3) The ordering variables are distributed in subqueries sets with nonadjacent priorities.



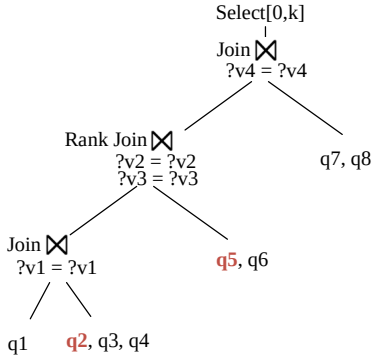
(a) **The basic execution process**



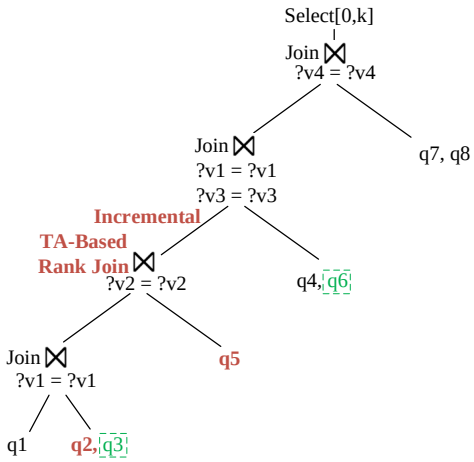
(b) **The optimal execution process**

Fig. 11: The ordering variables are distributed in subqueries set with the same priority.

As shown in Fig. 11(a), the subqueries q_3, q_4 and q_5 are at the same execution priority set and the ordering variables are distributed in subqueries q_4 and q_5. For this category, it optimization strategy is shown in Fig. 11(b), the subqueries with ordering variables q_4 and q_5 are executed together firstly. Then, the TA-Based rank join will be used to order the current query results. Finally, for the remaining subqueries, the incremental execution method is adopted, just like single variable ordering top-k query. Among that, if there are other subqueries in the same priority subqueries set that contain order variables. Those subqueries (here is q_3) should be executed between the current priority subqueries set (q_4 and q_5) and the next priority subqueries set (q_6 and q_7).



(a) **The basic execution process**

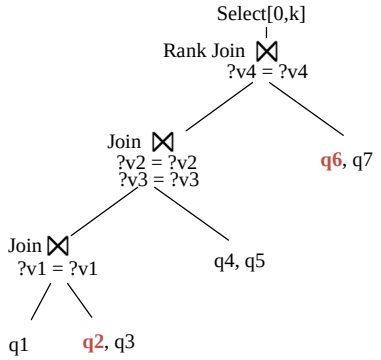


(b) **The optimal execution process**

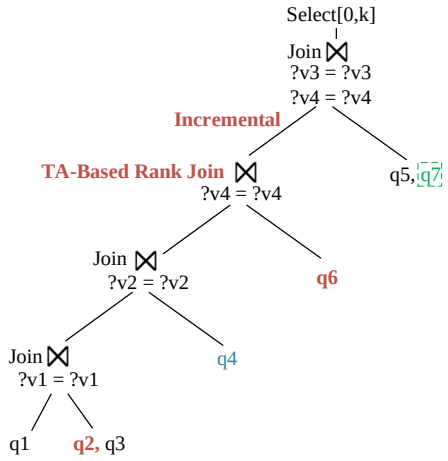
Fig. 12: The ordering variables are distributed in subqueries sets with adjacent priorities.

For the second category, as shown in Fig. 12(a), the subqueries with ordering variables q_2 and q_5 are distributed in subqueries sets with adjacent priorities. It optimization strategy is shown in Fig. 12(b), the subquery q_2 and the subquery q_3 with the same variables as q_5 will be executed firstly. In particular, if q_2 and q_5 have the same variables, the q_3 will be remove from the subqueries set $\{q_2, q_3\}$ to $\{q_4, q_6\}$. Then, after the execution of q_5, the TA-Based rank join method will be used to ordering the current query results. Finally, the remaining subqueries in the two adjacent priority subqueries sets are merged together, and the incremental execution method is adopted until the remaining

query plan execution is completed. Among that, if the variable $?v3$ is contained in q_4 but not q_2 and q_3, the q_6 will be executed after q_4.



(a) **The basic execution process**



(b) **The optimal execution process**

Fig. 13: The ordering variables are distributed in subqueries sets with different and nonadjacent priorities.

For the last category, as shown in Fig. 13(a), the subqueries with ordering variables q_2 and q_6 are distributed in subqueries sets with different and nonadjacent priorities. As shown in Fig. 12(b), the subquery q_4 will be selected and executed from the subqueries set $\{q_4, q_5\}$. Because q_4 has the same variables with q_6. Then, after the execution of q_6, the TA-Based rank join method will be used to ordering the current query results. The rest of the execution process is the same as the second category.

8 Experiments

In this section, we evaluate our proposed federated top-k queries evaluation method (denoted as `FedTopKPro`) over both synthetic and real RDF benchmarks, **WatDiv** and **LargeRDFBench**. In the comparative experiment, we build up four indicators from previous work [1], [2], [39]: **#SST**, **#QET**, **#NRA** and **#TRT**. **#SST** is the time of source selection, and **#QET** represent the time of query execution. The number of remote accesses is denoted as **#NRA**, and **#TRT** denote the total run time from the beginning to the end of a query evaluation, which contains resource initialization time, source selection time, query execution time and resource release time.

8.1 Setting

LargeRDFBench. LargeRDFBench [31] is a comprehensive benchmark for evaluating and analyzing both the effectiveness and performance of federated RDF systems. It contains 13 datasets, involving Life Sciences, Cross Domain and Large Data. There are more than a billion triple patterns.

WatDiv. WatDiv [3] is a benchmark that enables diversified stress testing of RDF data management systems. In WatDiv, instances of the same type can have the different sets of attributes. We generate three datasets varying sizes from 100 million to 300 million triples and divide the schema graph of the collection into several connected subgraphs with METIS [21].

We conduct all experiments on a cluster of six machines running Linux. Five machines are used as RDF sources, and the other one is used as control site. There is 16GB memory and 150GB disk storage for each site, and each machine has one CPU with four-cores of 3.06GHz. Our top-k query optimization method code implemented by Java is deployed on the control site. To assess the performance of our approach, we design ten top-k queries for LargeRDFBench and WatDiv ($F_1 - F_{10}$ for LargeRDFBench and $W_1 - W_{10}$ for WatDiv. Among that, F_1, F_3, F_5, F_7, F_9 and W_1, W_3, W_5, W_7, W_9 are top-k queries with single variable and the others are top-k queries with expression), respectively.

8.2 Evaluation of Proposed Optimization Strategies

In this experiment, we verify the effective of our proposed optimization strategies.

8.2.1 Evaluation of Auxiliary Index Construction

we analyze the time complexity of the auxiliary index construction Algorithm 1, which is a polynomial time. We do some relevant experiments on **WatDiv**, and the experiments show that with the increase of the dataset size, the construction time cost of the auxiliary index increases linearly. It is worth noting that in time complexity M represents the number of same multi-source predicate combination. Therefore, only C and J increase the time cost of auxiliary index construction when the predicate is fixed and only the triples of datasets increase, as shown in the Fig. 14. Therefore, the auxiliary indexing algorithm is efficient.

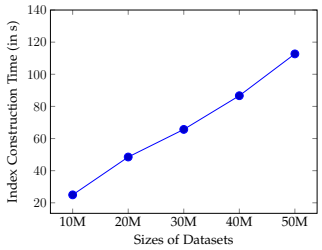
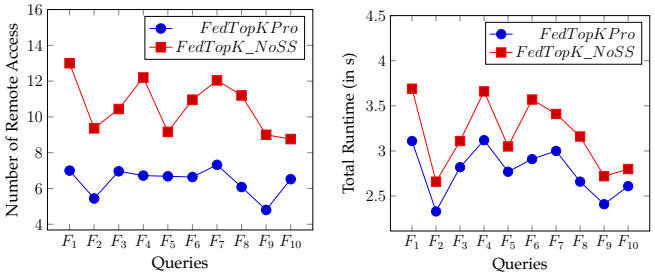


Fig. 14: Index construction times on different sizes of datasets.

8.2.2 Evaluation of Query Decomposition and Source Localization

In this experiment, we evaluate the effect of our proposed query decomposition and source localization using LargeRDFBench. We use `FedTopK_NoSS` to denote the first baseline method without the optimization of source localization strategy in Section 5. The difference between

FedTopKPro and FedTopK_NoSS is the number of subqueries, so the performance comparison between them can be evaluated by **#NRA** and **#TRT**. As shown in Fig. 15, experimental results show that the number of remote access of FedTopKPro are reduced by 50% on average compared with FedTopK_NoSS, and the total runtime can be reduced by 20% on average.

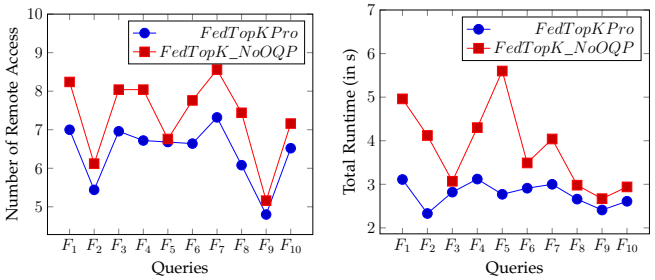


(a) **#NRA** (b) **#TRT**

Fig. 15: Effectiveness comparison between FedTopKPro and FedTopK_NoSS.

8.2.3 Evaluation of Cost-based Query Plan Generation

Here, we also use LargeRDFBench to test the cost-based query plan generation method. We propose a baseline named FedTopK_NoOQP to represent a baseline method without optimal query plan generation strategy in Section 6. It is the baseline method that does not adopt the cost-based optimal query plan strategy. As shown in Fig. 16, the average **#NRA** and **#TRT** of FedTopKPro is less 33% on average than FedTopK_NoOQP. The reason for this result is that the optimal query plan can preferentially execute other subqueries with lower cost before executing subqueries containing sorting variables, avoiding the unnecessary overhead in the circular execution strategy.

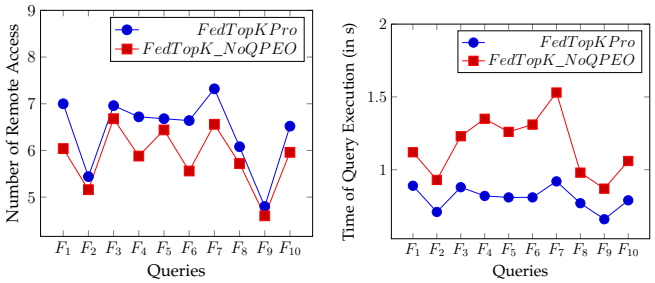


(a) **#NRA** (b) **#TRT**

Fig. 16: Effectiveness comparison between FedTopKPro and FedTopK_NoOQP.

8.2.4 Evaluation of Incremental Query Plan Execution

We design a baseline method FedTopK_NoQPEO, which does not include this query execution optimization strategy in Section 7, and also use LargeRDFBench to evaluate our incremental query plan execution method. As shown in Fig. 17, the average remote access times of FedTopKPro increase slightly compared with FedTopK_NoQPEO, because the query plan execution optimization strategy exists circular execution process. However, the incremental execution method can effectively reduce the range of subgraphs matching of subqueries to improve the query efficiency of subqueries. Therefore, the average **#QET** of FedTopKPro is less 50% on average than FedTopK_NoQPEO.

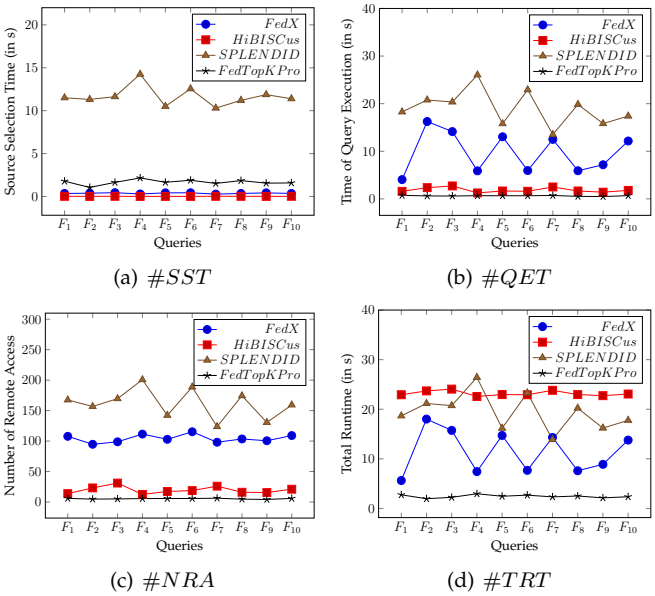


(a) **#NRA** (b) **#QET**

Fig. 17: Effectiveness comparison between FedTopKPro and FedTopK_NoQPEO.

8.3 Comparison with Existing Methods

Previous researchers have implemented several federated RDF systems, such as FedX, HiBISCuS, SPLENDID. By comparing our proposed method with three previous methods, we can verify the efficiency of our proposed method.



(a) $\#SST$ (b) $\#QET$

(c) $\#NRA$ (d) $\#TRT$

Fig. 18: Efficiency comparison with existing methods.

Fig. 18 shows that the comparison results among FedX, HiBISCuS, SPLENDID and FedTopKPro on four indicators over **LargeRDFBench**. In Figure 18(a), SPLENDID has the highest average **#SST** because it does not optimize the source selection. Both FedX and HiBISCuS regard source selection as one of their main optimization objective, so the results of these two methods on this indicator are excellent and slightly better than our method. Our method outperforms the previous three methods on the other three performance indicators in Figure 18(b), Figure 18(c) and Figure 18(d). The **#QET** of our method is 10 times faster than FedX and 18 times faster than SPLENDID on average. Because we build a cost model and propose an algorithm to generate the optimal query plan based on this model. Similarly, HiBISCuS proposed a hypergraph-based source selection approach, which reduced the number of RDF sources in the query process, thus reducing its query time. It can be found that the trend of performance comparison results of **#NRA** is roughly the same as that of **#QET**. Therefore, we

think that the number of remote access is proportional to the query execution time. At the beginning of FedTopKPro, we put forward an optimization idea focusing on shortening communication overhead of remote access, and this result can verify the correctness of our idea. Finally, the **#TRT** of our method is 10 times faster than FedX, 18 times faster than HiBISCuS and 22 times faster than SPLENDID on average. It is worth noting that the total query time of HiBISCuS is obviously abnormal. Its source selection time, remote access times and query execution time are short, but the total query time is long. The reason for this result is that the resource initialization and resource release of this method occupy a large overhead.

8.4 Evaluation of Scalability

In order to further explore the superiority of our method, we compare the robustness of our approach with previous methods on synthetic datasets, **WatDiv**, with different scales. We generate three datasets varying sizes from 100 million to 300 million triples, and Fig. 19 shows the result. We can find that with the larger of datasets' scales, the total runtime of four method all increases. However, the query cost of our method is always better than other methods, and the rate of increasing for our method is the smallest. Therefore, we think our method has strong robustness.

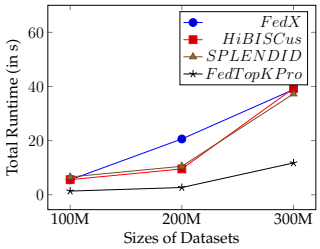


Fig. 19: Scalability comparison on different sizes of datasets.

8.5 Evaluation of K

In order to evaluate the query efficiency of optimization schemes under different values of k, we have conducted experiments on **WatDiv**. A total of 10 top-k queries are set up in the experiment, among which 5 queries are single variable top-k query with k values from 10 to 80. The other five queries are expression top-k queries, and the values of k also vary from 10 to 80. The 10 top-k queries are executed 10 times respectively, and the performances of different k values obtained by averaging them are shown in the Fig. 20. The experimental results show that with the increase of k, the query time can increases linearly.

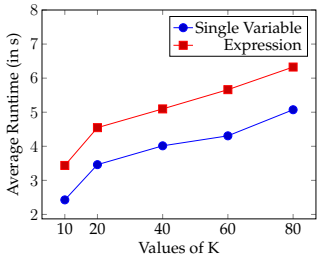


Fig. 20: Efficiency on different values of k.

9 Conclusion

In this paper, we have studied the optimization both for single variable ordering and expression ordering top-k queries over federated RDF systems. The proposed scheme mainly involves query decomposition and source localization, the construction of cost model, the cost-based optimal query plan generation and the incremental query plan execution. In order to evaluate the reliability of the proposed scheme, we have done a lot of experiments. Firstly, through the horizontally comparison experiment between the three baselines. The effectiveness of the various optimization strategies have been verified. Secondly, compared with the previous three typical query methods in the real datasets, the efficiency of our proposed optimal method have been verified. Finally, the robustness and scalability of proposed scheme have been verified by comparing with the previous three methods on synthetic datasets with different scales.

References

[1] M. Acosta, M. Vidal, F. Flöck, S. Castillo, C. B. Aranda, and A. Harth. SHEPHERD: A Shipping-Based Query Processor to Enhance SPARQL Endpoint Performance. In *ISWC*, pages 453–456, 2014.
[2] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *ISWC*, pages 18–34, 2011.
[3] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*, pages 197–212, 2014.
[4] K. Anyanwu. A Vision for SPARQL Multi-Query Optimization on MapReduce. In *Workshops of ICDE*, pages 25–26, 2013.
[5] C. B. Aranda, M. Arenas, Ó. Corcho, and A. Polleres. Federating Queries in SPARQL 1.1: Syntax, Semantics and Evaluation. *J. Web Semant.*, 18(1):1–17, 2013.
[6] A. Bozzon, E. Della Valle, and S. Magliacane. Extending SPARQL algebra to support efficient evaluation of top-k SPARQL queries. In *Search Computing*, pages 143–156. Springer, 2012.
[7] M. Cai and P. Revesz. Parametric R-tree: An index structure for moving objects. In *Proc. of the COMAD*, 2000.
[8] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos. SemaGrow: Optimizing Federated SPARQL Queries. In *SEMANTICS*, pages 121–128, 2015.
[9] B. Dan and R. V. Guha. Resource Description Framework (RDF) Schema Specification: Proposed Recommendation. 2000.
[10] G. Dueck and T. Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90(1):161–175, 1990.
[11] Garcia-Molina, Hector, Ullman, D. Jeffrey, and Jennifer. *Database Systems: The Complete Book*. Hector Garcia-Molina and Jeffrey D. Ullman and Jennifer Widom, 2008.
[12] N. Ge, Z. Qin, P. Peng, and L. Zou. FedTopK: Top-K Queries Optimization over Federated RDF Systems. In *DASFAA*, pages 595–599, 2021.
[13] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD*, 2011.
[14] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-demand Queries over Linked Data. In *WWW*, pages 411–420, 2010.
[15] A. Harth and S. Speiser. On Completeness Classes for Query Evaluation on Linked Data. In *AAAI*, 2012.
[16] O. Hartig. SPARQL for a Web of Linked Data: Semantics and Computability. In *ESWC*, pages 8–23, 2012.
[17] A. Hogan, A. Harth, and A. Polleres. Scalable Authoritative OWL Reasoning for the Web. *Int. J. Semantic Web Inf. Syst.*, 5(2):49–90, 2009.
[18] S.-Y. Ihm, K.-E. Lee, A. Nasridinov, J.-S. Heo, and Y.-H. Park. Approximate convex skyline: A partitioned layer-based index for efficient processing top-k queries. *KBS*, 61:13–28, 2014.
[19] Y. Izquierdo, M. A. Casanova, G. García, F. Dartayre, and C. H. Levy. Keyword search over federated RDF datasets. In *ER Forum/Demos*, pages 86–99, 2017.

[20] T. Jiang, B. Zhang, D. Lin, Y. Gao, and Q. Li. Incremental evaluation of top-k combinatorial metric skyline query. *KBS*, 74:89–105, 2015.
[21] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *ICPP*, pages 113–122, 1995.
[22] S. Magliacane, A. Bozzon, and E. Della Valle. Efficient Execution of Top-K SPARQL Queries. In *ISWC*, pages 344–360, 2012.
[23] G. Montoya, H. Skaf-Molli, and K. Hose. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *ISWC*, pages 471–489, 2017.
[24] P. Peng, Q. Ge, L. Zou, M. T. Özsu, Z. Xu, and D. Zhao. Optimizing Multi-Query Evaluation in Federated RDF Systems. *TKDE*, 2019.
[25] P. Peng, L. Zou, M. T. Özsu, and D. Zhao. Multi-query Optimization in Federated RDF Systems. In *DASFAA*, pages 745–765, 2018.
[26] P. Peng, L. Zou, and Z. Qin. Answering top-k query combined keywords and structural queries on rdf graphs. *IS*, 67(JUL.):19–35, 2017.
[27] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
[28] F. Prasser, A. Kemper, and K. A. Kuhn. Efficient Distributed Query Processing for Autonomous RDF Databases. In *EDBT*, pages 372–383, 2012.
[29] E. PrudHommeaux. SPARQL query language for RDF. *http://www.w3. org/TR/rdf-sparql-query/*, 2008.
[30] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *ESWC*, pages 524–538, 2008.
[31] M. Saleem, A. Hasnain, and A. N. Ngomo. LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation. *J. Web Semant.*, 48:85–125, 2018.
[32] M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC*, pages 176–191, 2014.
[33] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *ISWC*, pages 585–600, 2011.
[34] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, pages 601–616, 2011.
[35] D. Wang, L. Zou, and D. Zhao. Top-k Queries on RDF Graphs. *Information Science*, 316:201–217, 2015.
[36] Q. Wang, P. Peng, T. Tong, Z. Tian, and Z. Qin. Keyword Search over Federated RDF Systems. In *DASFAA*, pages 613–622, 2020.
[37] Y. Wang, X. Xu, Q. Hong, J. Jin, and T. Wu. Top-k star queries on knowledge graphs through semantic-aware bounding match scores. *KBS*, 213:106655, 2021.
[38] Z. Yang, X. Zhou, K. Li, G. Xiao, Y. Gao, and K. Li. Efficient processing of top k group skyline queries. *KBS*, 182:104795, 2019.
[39] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: A Graph-based SPARQL Query Engine. *VLDB J.*, 23(4):565–590, 2014.

Ningchao Ge received his BS degree at Hunan University in 2016 and he is pursuing a Ph.D. degree advised by Dr. Zheng Qin and Dr. Peng Peng in Computer Science and Technology at Hunan University at now. His research is focused in federated RDF systems and knowledge graphs.

Zheng Qin received the Ph.D. degree in computer software and theory from Chongqing University, China, in 2001. He has rich experience in products development and application services, such as financial, medical, military, and education sectors. He is currently a Professor of computer science and technology with Hunan University, China. His main research interests include computer networks and information security, cloud computing, big data processing, and software engineering. He is a member of the China Computer Federation (CCF) and ACM.

Peng Peng received his BS degree and Ph.D. degree in Computer Science at Beijing Normal University and Peking University in 2009 and 2016, respectively. Now, he is an associate Professor of Hunan University. His research interests include graph database and distributed RDF system.

Mingdao Li received his BS degree at Hunan University in 2017 and he is currently pursuing a Ph.D. degree advised by Dr. Peng Peng and Dr. Zheng Qin in Computer Science and Technology at Hunan University.His research is focused in graph database and community search over large graphs.

Lei Zou received his B.S. degree and Ph.D. degree in Computer Science at Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. Now, he is a professor in Peking University. His research interests include graph database and semantic data management.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York. He is also a national distinguished professor with Hunan University, China. His current research interests include cloud computing, fog computing andmobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored more than 830 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds more than 60 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 5 most influential scientists in parallel and distributed computing based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Cloud Computing, IEEE Transactions on Services Computing, and IEEE Transactions on Sustainable Computing.