Front.Comput.Sci.
DOI

RESEARCH ARTICLE

Redesign of gStore System

Li ZENG, Lei ZOU

Peking University, Beijing, 100871, China

Abstract gStore [1] is an open-source native RDF (Resource Description Framework) triple store, which answers SPARQL queries by subgraph matching over RDF graphs. However, there are some deficiencies in the original system design, such as answering simple queries (such as one-triple pattern query). To improve the system's efficiency, we re-consider the system design in this paper. Specifically, we propose a new query plan generation module, which generates different query plans according to the query graphs' structures. Furthermore, we re-design our vertex encoding strategy to achieve more pruning power and a new multi-join algorithm to speed up subgraph matching process. Extensive experiments on synthetic and real RDF datasets show that our method outperforms the state-of-the-art algorithms significantly.

Keywords Graph Database, Subgraph Matching, RDF Management, SPARQL Query

1 Introduction

The RDF (Resource Description Framework) data model was originally proposed by W3C for modeling Web objects as a part of developing the Semantic Web. However, it is now used beyond that. RDF has further gained popularity due to the launching of "knowledge graph" by Google in 2012. An RDF dataset is a collection of *triples* of the form ⟨subject, property, object⟩. A triple can be naturally seen as a pair of entities connected by a named relationship or an entity associated with a named attribute value. In contrast to relational databases, an RDF dataset is self-describing and does not need to have a schema (although one can be defined using RDFS). The simplicity of this representation makes it easy to use RDF for modeling various types of data and favors data integration.

There exist many large-scale RDF datasets, e.g., Freebase [2] has 2.5 billion triples and DBpeida [3] has over 1.1 billion triples . The large volume of RDF repository requires efficient RDF data management systems. Generally, there are two typical approaches to design these systems: relational approaches [4–6] and graph-based approaches [1, 7]. The former maps RDF data to a tabular representation in a number of ways and then executes SPARQL queries on them – sometimes mapping SPARQL queries to SQL.

The second major category is graph-based, which models both RDF data and the SPARQL query as a graph and evaluate the query by subgraph matching using homomorphism, e.g., [1, 7–9]. The advantage of this approach is that it maintains the original representation of the RDF data and enforces the intended semantics of SPARQL. Also, some graph database techniques, such as the structure-aware indices [1, 8] and graph-based query algorithms [7], are more suitable for RDF data.

Our earlier work, gStore [1, 8], is a graph-based RDF data management system (or what is commonly called a "triple store") that maintains the graph structure of the original RDF data. Its data model is a labeled, directed multiedge graph (called RDF graph – See Figure 1 and Table 1), where each vertex corresponds to a subject or an object. We also use a query graph Q (see Figure 2 and Listing 1) to represent a SPARQL query. Query processing involves finding subgraph matches of Q over the RDF graph G. gStore incorporates an index over the RDF graph (called VS*-tree) to speed up query processing. VS*-tree is a height-balanced

Received month dd, yyyy; accepted month dd, yyyy

E-mail: {zengli-bookug, zoulei}@pku.edu.cn; Corresponding author: Lei Zou

tree with a number of associated pruning techniques to speed up subgraph matching. According to our experiment analysis and other related work [10], gStore is faster than other comparative systems, such as Apache-Jena, Virtuoso and RDF-3x, in complex SPARQL queries. The reason is that gStore uses the whole query graph's structure to reduce the search space, but other systems (such as Jena and Virtuoso) utilize step-by-step relational join strategy without considering structure-aware pruning. However, gStore is not good at simple SPARQL queries, e.g., one-triple pattern queries, which occupy a large proportion of practical knowledge graph applications' query workloads. Generally speaking, there are three limitations in the original gStore design.

1. *Rigid Framework*. gStore uses "filter-and-join" framework for all kinds of queries, which does not consider queries' structures. Obviously, if a query contains a single triple pattern, it is easy to figure out the answers by using key-value store directly, without going through "filter-and-join" pipeline.
2. *Vertex Signature Coding Strategy*. As we know, gStore's pruning power depends on the vertex signature compression. However, the original signature coding strategy decreases its pruning power as the dataset size increases, resulting in more cost in join process.
3. *Naive Join Method*. The original join method is a simple BFS-like search to find subgraph isomorphisms, which doesn't consider any pruning strategy in the searching process.

1.1 Our Contributions

Considering the above shortcomings of the original gStore system design, we propose several optimization techniques in this work and re-design gStore to deal with datasets in size of billions. Note that this paper concentrates on BGP (Basic Graph Pattern). Although our current released gStore system can support OPTIONAL, UNION, FILTER and aggregation syntax, these are not the focus of this paper. In particular, we integrate the proposed optimization techniques with the original gStore system on GitHub. Generally, we propose the following three optimization solutions.

First, we introduce "Strategy" module to generate different query plans considering different kinds of query graph structures, which avoids the rigid framework in the original gStore design. Given a query graph Q, this module decides if it goes through the "filter-and-join" framework according to the query graph structure. Even for queries that need to go through such framework, we also divide the vertices in Q into different categories based on the neighborhood structure. Each category generates different query plans.

We also design a new vertex signature encoding method to improve the pruning power of the VS*-tree. As we know, the more powerful the vertex signature encoding is, the fewer candidates will be generated after the VS*-tree filtering. Consequently, it will lead to a big performance gain in the total query response time. In this paper, we adopt several methods to reduce conflicts in vertices' signatures. On one hand, entities and literals are divided apart, and incoming and outgoing edges are also separated. On the other hand, an entity or a literal is combined with the corresponding edge, to describe the restrictions more precisely.

Furthermore, a new join strategy is proposed to accelerate the join process, i.e. multi-join strategy. This strategy generates a processing order based on a heuristic method to prune unpromising intermediate results as early as possible. Specifically, the multi-join strategy uses A* search instead of breadth-first search so that we can utilize the restrictions of several edges simultaneously. Experimental results show that the new join strategy reduces the time cost a lot and lowers the space cost as well.

The rest of the paper is organized as follows: We formally define the preliminary concepts in Section 2. Section 3 is an overview of the proposed approach and explains the intuition behind it. The query plan selection in this general framework is presented in Section 4, and Section 5 introduces the new encoding method of signature. Optimizations of Join module is detailed in Section 6. The result of experiments is presented in Section 7. Finally, we survey the related work in Section 8 and conclude in Section 9. Queries used for experiments are listed in Appendix.

2 Problem Definition

In this section, we review the terminology that we use throughout this paper. As mentioned earlier, this paper only studies BGP (Basic Graph Pattern) queries, which are essentially subgraph homomorphism between query graph Q and data graph G, as defined as follows. Table 2 shows some frequently-used notations.

Definition 1 RDF dataset. Let pairwise disjoint infinite sets I, B, and L denote URI, blank nodes and literals, respectively.

subject	predicate	object
<Mike>	<BornIn>	<Washington>
<Mike>	<Mother>	<Alice>
<Mike>	<Friend>	<Lucy>
<Mike>	<Friend>	<Bob>
<Mike>	<Teacher>	<T_1>
<Mike>	<Teacher>	<T_2>
...	...	...
<Mike>	<Teacher>	<T_{100}>
<Lucy>	<Friend>	<Bob>
<Bob>	<Height>	"175"
<Bob>	<Age>	"22"
<T_1>	<FatherOf>	<Bob>
<T_1>	<Graduate>	<PKU>
<T_2>	<Graduate>	<PKU>
...	...	...
<T_{100}>	<Graduate>	<PKU>
<PKU>	<Type>	<School>

Table 1 rdf table

An RDF dataset is a collection of triples, each of which is denoted as $t(subject, property, object) \in (I \cup B) \times I \times (I \cup B \cup L)$.

A triple can be naturally seen as a pair of nodes connected by a named relationship. Hence, an RDF dataset can be represented as a data graph where subjects and objects are vertices, and triples are edges with property names as edge labels. Note that there may exist more than one property between a subject and an object, that is, multiple-edges may exist between two vertices in an RDF graph.

Definition 2 RDF graph . An *RDF graph* is a four-tuple $G = \langle V, L_V, E, L_E \rangle$, where

1. $V = V_c \cup V_e \cup V_b \cup V_l$ is a collection of vertices that correspond to all subjects and objects in RDF data, where V_c, V_e, V_b and V_l are collections of class vertices, entity vertices, blank vertices and literal vertices, respectively.
2. L_V is a collection of vertex labels. Given a vertex $u \in V_l$, its vertex label is its literal value. Given a vertex $u \in V_c \cup V_e$, its vertex label is its corresponding URI. The vertex label of a vertex in V_b (blank node) is NULL.
3. E is a collection of directed edges $\{\overrightarrow{u_i, u_j}\}$ that connect the corresponding subjects (u_i) and objects (u_j) .
4. L_E is a collection of edge labels. Given an edge $e \in E$, its edge label is its corresponding property.

An edge $\overrightarrow{u_i u_j}$ is an *attribute property* edge if $u_j \in V_l$; otherwise, it is a *link* edge. □

Definition 3 (Basic Graph Pattern). A *basic graph pattern*

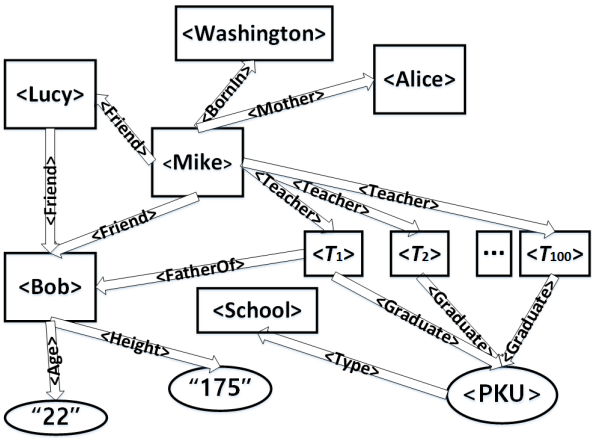

Fig. 1 Example Data Graph

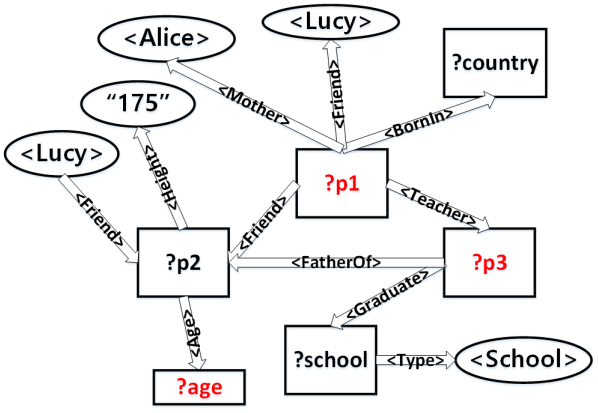

Fig. 2 Example Query Graph

is a connected graph, denoted as $Q = \{V(Q), E(Q)\}$, such that (1) $V(Q) \subseteq (I \cup L \cup V_{Var})$ is a set of vertices, where I denotes URI, L denotes literals and V_{Var} is a set of variables; (2) $E(Q) \subseteq V(Q) \times V(Q)$ is a set of edges in Q; and (3) each edge e in $E(Q)$ either has an edge label in I (i.e., property) or the edge label is a variable.

A data graph G and a BGP (basic graph pattern) queries are given in Figure 1 and 2, respectively, which are used as running examples throughout this paper. The corresponding RDF triples (subject, predicate, object) of the data graph are displayed in Table 2. In the data graph, <T_1>, <T_2>, ...<T_{100}> are all teachers, each with edge labels <Teacher> and <Graduate>. However, only <T_1> has an edge label <FatherOf>. A match of BGP over RDF graph is defined as a partial function μ from V_{Var} to the vertices in the RDF graph. Formally, we define the match as follows:

Definition 4 (BGP Match) Consider an RDF graph G and a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$. A

Listing 1 core-satellite query

```
SELECT ?p1 ?p3 ?age WHERE
{
?p1 <BornIn> ?country .
?p1 <Friend> <Lucy> .
?p1 <Mother> <Alice> .
?p1 <Friend> ?p2 .
?p1 <Teacher> ?p3 .
?p2 <FatherOf> ?p3 .
?p2 <Height> "175" .
?p2 <Age> ?age .
<Lucy> <Friend> ?p2 .
?p3 <Graduate> ?school .
?school <Type> <School> .
}
```

subgraph M with m vertices $\{u_1, ..., u_m\}$ (in G) is said to be a *match* of Q if and only if there exists a *function* μ from $\{v_1, ..., v_n\}$ to $\{u_1, ..., u_m\}$ ($n \geqslant m$), where the following conditions hold:

1. if v_i is not a variable, $\mu(v_i)$ and v_i have the same URI or literal value ($1 \leqslant i \leqslant n$);
2. if v_i is a variable, there is no constraint over $\mu(v_i)$ except that $\mu(v_i) \in \{u_1, ..., u_m\}$;
3. if there exists an edge $\overrightarrow{v_i v_j}$ in Q, there also exists an edge $\overrightarrow{\mu(v_i)\mu(v_j)}$ in G; furthermore, $\overrightarrow{\mu(v_i)\mu(v_j)}$ has the same property as $\overrightarrow{v_i v_j}$ unless that the label of $\overrightarrow{v_i v_j}$ is a variable.

The problem that we studied in this paper is defined as follows:

Problem Statement. Given an RDF graph G and a BGP query graph Q, where $|V(Q)| \ll |V(G)|$ and $|E(Q)| \ll |E(G)|$, our problem is to find all BGP matches (Definition 4) of Q in G.

In this paper, we assume that query graph Q are always connected; otherwise, all connected components are considered separately. In addition, for the convenience of the presentation, we do not consider the case that an edge's label is a variable in the query graph, though such cases can be handled easily by our system. In the rest of this paper, we use $E(G)$ to denote the edge set of the data graph. Besides, we say $(s, p, o) \in E(G)$ if s and o is connected in G and this edge's label is p(p is called the property in RDF dataset, also called predicate).

Table 2 Notations

G, Q	Data graph and query graph, respectively
v, u	Vertex in query graph and data graph, respectively
$\overrightarrow{vv'}$	Single edge with v and v' being the end points
$Sig(v), Sig(u)$	encoding of vertex v or u
$N(v), N(u)$	all neighbors of vertex v or u
$C(v)$	all vertices in data graph that match the criteria associated with vertex v
$Core(Q)$	variables whose degree is bigger than 1 in query graph Q
$Sat(Q)$	variables which are selected(and degree is 1) in query graph Q
$Iso(Q)$	variables which are not selected(and degree is 1) in query graph Q
MRT	the intermediate result table, each row represents a record, each column correspondings to a query variable
$\|P(G)\|$	the number of all different predicates in G

3 System Architecture

Figure 3 presents the whole system architecture of gStore, which consists of two stages (offline and online). The new gStore has the exactly same offline framework with the original design except for the new vertex-encoding strategy, while the new version introduces one more module ("Strategy" module) for query plan generation and updates the "Join" module. The new added/updated modules are highlighted in red font. To make the paper self-contained, we present the whole architecture but underline the new added/updated modules.

The offline process is to store an RDF dataset and build the VS*-tree index. We describe the main components (as shown in Figure 3): RDF parser accepts three popular RDF file formats (RDF/XML, N3, Turtle). The parsing result is a collection of RDF triples. Based on the parsed triples, we build an RDF graph using adjacency list representation, where each entity is a vertex (represented by its URI) and the incident edges to the vertex correspond to the triples containing the entity. We use a key-value store to index the adjacency lists, where URIs are keys. In the encode module, we encode the RDF graph G into a signature graph G^*. Specifically, each vertex in G^* has a bitstring that encodes the neighborhood structure around the vertex. Different from the original design, the new vertex coding strategy considers the more fine-grained neighbor features around one vertex.

Finally, VS*-tree builder is to construct a VS*-tree over

G^*. The signature graph G^* and the VS*-tree are stored in key-value store and VS*-tree store, respectively.

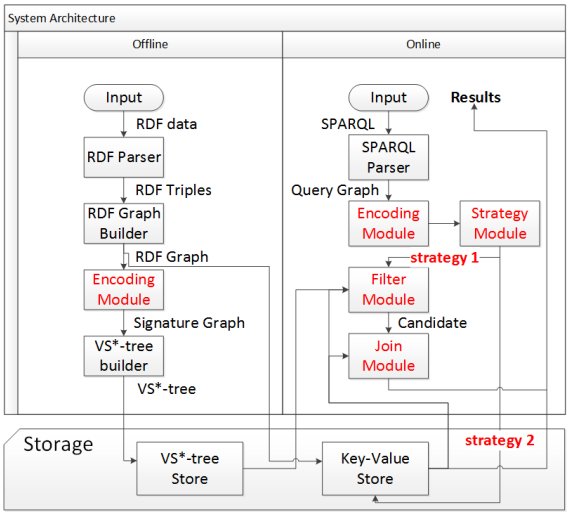

Fig. 3 System Architecture

At the online stage, a SPARQL statement is an input to the SPARQL parser, which is generated by a parser generator library called ANTLR3. The SPARQL query is parsed into a syntax tree, based on which, we build a query graph Q and encode it into a query signature graph Q^*. The encoding strategy is analogue to encoding RDF graphs, and the details of encoding will be talked in Section 5. Different from the original design, the new added module "Strategy" will generate the query plan based on the query graph's structure. For example, if a query is simple (e.g., one triple pattern), it will not go through the "filter-and-join" framework. Even for the queries that need to go through such framework, we also generate different query plans considering the query graph' structures. More details about "Strategy" module are given in Section 4. Furthermore, we also optimize the join strategy (Section 6) in the new gStore design.

4 Query Plan Selection

In the old gStore system, "filter and join" strategy is used for all SPARQL queries. It uses VS*-tree to get candidates for each query variable first, then all variables are joined on the specified predicate to verify the candidates. However, for one-triple queries, we can get their results immediately using key-value store. In such cases, we do not have to consider the cost of VS*-tree module and Join module if we choose to use key-value store instead of the old strategy.

In addition, in the original "filter and join" strategy, we need to find candidates for every variable of a query, and all variables will participate in the join process. VS*-tree is good at pruning candidates for vertices which have many restrictions, so if a variable has few edges and few constant neighbors, then the performance of pruning will weaken a lot.

As it can be seen in Figure 3, we add a Strategy module to generate a good query plan, and the details are as follow.

4.1 Special Structure

If there is only one triple in a query graph Q, then we should answer this query directly using key-value store. These special structures can be divided into three categories:

Listing 2 special: p2so

```
SELECT ?s ?o WHERE
{
        ?s  <Friend> ?o  .
}
```

Listing 3 special: p2s

```
SELECT ?s WHERE
{
        ?s  <Friend> ?o  .
}
```

Listing 4 special: sp2o

```
SELECT ?o WHERE
{
        <Mike> <Friend> ?o .
}
```

Listing 2 is answered via key-value store(key is the predicate, value is the subject-object list), and gStore return the results directly. Listing 3 means that there are two variables in the query, but only one is selected. If the selected one is a subject, then key is the predicate while value is the subject list; if the selected one is an object, then value is the object list instead(key is also the predicate).

Listing 4 means that there is only one variable in the query, and it is out of question the selected one. If this variable is a subject, then we use key-value store to get its results with (object, predicate) pair as key and subject list as value; if this variable is an object, then use (subject, predicate) pair as key and object list as value instead.

Cases discussed above all neglect the "filter and join" framework, and use key-value store directly. Theoretically

this will harvest a big gain in performance, because the cost of filtering and table join is discarded. Experiments show that this strategy behaves well on 1-triple queries, but when dealing with more triples, "filter and join" framework shall be applied.

4.2 General Case

To deal with general cases, we design a new "filter and join" framework which is very different from the older one. We consider the structure of a query graph and divide the variables of this query into several categories: $Sat(Q)$, $Iso(Q)$ and $Core(Q)$. Given a query variables v, if v's degree is greater than 1, then $v \in Core(Q)$. Otherwise, v belongs to $Sat(Q)$ if v is in the select-clause, i.e. v is selected and should be present at the query result. If v is not selected, then $v \in Iso(Q)$. In the new framework, we use different strategies for $Core(Q)$, $Sat(Q)$ and $Iso(Q)$.

Given the SPARQL query in Figure 2 and Listing 1, only ?p1, ?p3 and ?school are selected. The content of all sets are listed below:

- $Sat(Q)$: ?age
- $Core(Q)$: ?p1, ?p2, ?p3, ?school
- $Iso(Q)$: ?country

The new algorithm is given in Algorithm 1. When dealing with general cases, we only retrieve candidates for variables in $Core(Q)$. VS*-tree finishes this process and the cost is expected to be low. For the query in Figure 2, gStore generates candidate sets for ?p1, ?p2, ?p3 and ?school, and the candidate sets are named $C(?p1)$, $C(?p2)$, $C(?p3)$ and $C(?school)$.

After that, we join the candidates of all variables in $Core(Q)$ on corresponding edges. We use MRT to store the temporary results in the join process, and each column of MRT represents all mapping vertices of a query variable. Each row of MRT is a partial answer, named ans, and the mapping vertex of a query variable v in this answer can be denoted as $ans[v]$. For example, in the example query gStore joins ?p1, ?p2, ?p3 and ?school one by one. After these 4 variables are joined, the partial result is placed in MRT.

Later, we consider the restricitions of variables in $Iso(Q)$. Notice that a variable v in $Iso(Q)$ must be linked to one and only one variable in $Core(Q)$. We call this variable v_0 and the edge between v_0 and v is named r_0. The candidate set of v_0(called $C(v_0)$) should all have an edge with a label r_0, which is viewed as a restriction to prune the partial result MRT. In Figure 2, ?country is in $Iso(Q)$, and the edge <BornIn> is used to prune the partial result MRT. No need to generate results for ?country, but all records of MRT should satisfy the restriction of <BornIn>. After this step, the content of MRT is given in Table 3.

Table 3 Partial Result

?p1	?p2	?p3	?school
<Mike>	<Bob>	$<T_1>$	<PKU>

Table 4 Final Result

?p1	?p3	?age
<Mike>	$<T_1>$	"22"

Finally, we generate results for each variable sv in $Sat(Q)$ from MRT directly because sv is also linked to one and only one variable in $Core(Q)$. Assume the variable it links to is named v_1, then v_1 must be in $Core(Q)$. For each record in the partial result MRT, we start from the mapping vertex of v_1 to get the results of sv using key-value store. For example, in Figure 2 ?age's results are generated by calling key-value store from all mapping vertices of ?p3 in MRT(the corresponding predicate is <Age>). The final result is given in Table 4, notice that only the results of selected variables need to be saved in the end.

The point is that gStore does not need to join all variables now, which will save a lot of time because the cost is usually high. Besides, there is no need to retrieve candidates for variables in $Iso(Q)$ and join them, because we do not want to get the results of these variables. As for variables in $Sat(Q)$, also no need for such moves, we just generate their results at last by key-value store. To sum up, the new framework will harvest a great improvement in the system's performance.

By now we have selected a query plan to run according to the query's structure, later we will need to optimize the "filter and join" framework. In the next two sections, we will discuss how to improve the VS*-tree and speed up the join process respectively.

5 A New Encoding Method

As mentioned in previous sections, we need to retrieve candidates for each query variable in $Core(Q)$. The filtering strategy is VS*-tree [1], an index tree which is made up of all vertices' signatures. We have indicated in Section 3, that the efficiency of VS*-tree is very crucial because it has great influence on the cost of the join process. If the VS*-tree is not efficient, then the number of candidates will be very big. As a result, it will be extremely costly to join the candidates of variables because we must enumerate all candidates to

Algorithm 1 Query Plan Selection

Input: query graph Q, empty MRT.
Output: final result in MRT.

```
if Q is 1-triple graph then
    get the final results using key-value store and put them into MRT;
    return MRT;
for all var in Core(Q) do
    get C(var) by VS*-tree;
    if C(var) is empty then
        return empty MRT;
use Algorithm 2 to join all results of variables in Core(Q);
filter the MRT using edges linking to variables in Iso(Q);
generate the results of variables in Sat(Q) from MRT;
return the final result MRT;
```

check if they satisfy the restrictions of the linking edge.

To improve the efficiency of VS*-tree is to improve the precision of the encoding. Notice that we don't care much about the cost of VS*-tree, and it is the filtering efficiency that really matters. For a veretx v in query graph, we denote its neighbor set as $N(v)$ and its encoding as $Sig(v)$. Similarly, for a veretx u in data graph, we denote its neighbor set as $N(u)$ and its encoding as $Sig(u)$. In the VS*-tree, we check if a veretx u in data graph G can be matched to a variable v in query graph Q by comparing their encoding, i.e. $Sig(u)$ and $Sig(v)$. The encoding contains the information of neighbors, and u and v are matched only when $Sig(u)$ & $Sig(v) = Sig(v)$.

Aiming to reduce the number of candidates, the encoding's structure must be optimized to describe neighbors' information more precisely. A vertex can have many neighbors, however, we only have a limited length for its signature. Therefore, a hash function is used to map a neighbor's information(including predicate, entity or literal) to some bits of the signature. To distinguish between different vertices in data graph, we must consider any subtle differences in their neighbors. As a result, our goal is to lower the conflict between different neighbors, then different vertices' encodings are expected to be different.

In the original encoding method, a vertex's encoding is divided into two parts: string part and edge part. The first part is used to encode the linking neighbors(entity or literal), while the second part is used to encode the linking edges(i.e., predicates). We used to assign 590 bits for string signature, and 354 bits for edge signature. Notice that these two parts don't conflict with each other, and the original signature's structure is given in Figure 4.

In order to improve the signature, firstly we divide the whole structure into more parts to reduce the conflicts. Secondly, we bind each edge with the neighbor

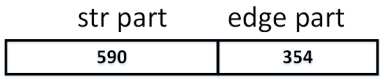

Fig. 4 the old structure of a signature

corresponding to it. These two optimizations are presented in the subsections below.

5.1 Divide the signature

Signature is already divided into two parts in the original structure, i.e. string part and edge part. However, this is not enough. In Figure 2, query variables ?p1 and ?p2 both have an edge whose label is <Friend>. This predicate will be encoded into the edge part of the encoding for ?p1, as well as ?p2. The problem is that the two encodings will be totally the same(now we focus on this predicate, i.e. <Friend>), but they are different in fact. Our solution is to divide the edge part into two subparts: in-edge part(100 bits, denoted as "in-edge" in Figure 5) and out-edge part(100 bits, denoted as "out-edge" in Figure 5). <Friend> is encoded into the out-edge part in ?p1's signature, and the in-edge part in ?p2's signature. By this way, their signatures will be different and more precise. After filtered by VS*-tree, there will be fewer candidates for variables ?p1 and ?p2.

Besides, a neighbor can be either an entity or a literal, and we do not want to see that entities conflict with literals. Considering variable ?p2 in Figure 2, in the old encoding's structure, both <Lucy> and "175" will be encoded into the string part, which may bring conflicts due to the hash function. So we divide the string part into two subparts: entity part(400 bits) and literal part(200 bits). All entities are encoded into the entity part while literals are encoded into the literal part. For example, <Lucy> is encoded into the entity part in ?p2's signature while "175" into the literal part.

What's more, for a neighbor which is an entity instead of a literal, it can be linked by either an incoming or outgoing edge. Just like the query in Figure 2, if we encode <Lucy> into ?p1's signature and ?p2's signature directly, then there will be no difference between the encoding result of <Lucy> in the two signatures. So we choose to divide the entity part into two subparts: incoming-entity part(denoted as "in-entity" in Figure 5) and outgoing-entity part(denoted as "out-entity" in Figure 5). An entity should be encoded into either incoming-entity part or outgoing-entity part according to the corresponding edge's direction. In this example, <Lucy> is encoded into the incoming-entity part in ?p2's

signature and it is encoded into the outgoing-part in ?p1's signature.

5.2 Bind edge with neighbor

We have discussed earlier that the encoding should be divided into more parts to reduce the conflicts, and improve the efficiency of the filter. As a matter of fact, that's not enough. Considering variable ?p1 in Figure 2, <Alice> and <Lucy> are encoded into ?p1's outgoing-entity part while <Mother> and <Friend> into ?p1's out-edge part. We change ?p1's neighbors to get a new query in Listing 5.

Listing 5 encode example

```
SELECT ?p1 WHERE
{
?p1 <Mother> <Lucy> .
?p1 <Friend> <Alice> .
}
```

The new query's answer is empty if searched in Table 2. However, using current encoding method, $C(?p1)$ will not be empty because it will contain <Mike>. The point is that we don't bind the edge with the corresponding neighbor, so ?p1's signature in the new query is almost the same as ?p1's signature in Figure 2. In the new query, <Friend> should be combined with <Alice>, while <Mother> combined with <Lucy>.

Therefore, we append a new part to the current signature, named as neighbor-edge part. As mentioned before, entity and literal should be divided. In addition, edges with different directions should also be divided. So we divide the new part into three subparts: entity-edge-incoming part(denoted as "entity-in" in Figure 5), entity-edge-outgoing part(denoted as "entity-out" in Figure 5), and literal-edge part(denoted as "literal-edge" in Figure 5). Each subpart occupies 48 bits, so the total length of the signature is 944 bits.

For string part and edge part, we can use a hash function to map the entity/literal/predicate to a bit in the signature. But for the neighbor-edge part, given a (neighbor, edge) pair, how can we map it to the signature? The solution is provided by a paper [11], which introduces a special function.

Definition 5 Binding Function. The binding function is denoted as $f(x, y) = x + (x + y + 1) \times (x + y)/2$, assuming that(x, y) is a (neighbor, edge) pair.

It is proved in [11] that $f(x_1, y_1) \neq f(x_2, y_2)$, $\forall(x_1, y_1) \neq (x_2, y_2)$. Notice that the value of $f(x, y)$ may be very large, while encoding's length is limited. So we have to define a mapping function as follows:

Definition 6 Mapping Function. The mapping function is denoted as $g(x, y) = f(x, y)\%48$, assuming that(x, y) is a (neighbor, edge) pair.

For each (neighbor, edge) pair, we use $g(x, y)$ to encode it into one bit in the neighbor-edge part.

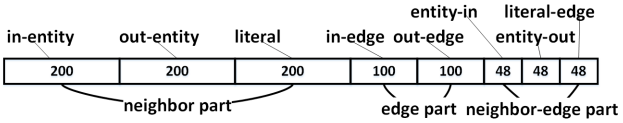

Fig. 5 the new structure of a signature

To sum up, the whole structure of the new signature is given in Figure 5, and it is discovered that new signature's length is the same as that of the old one. Our experiments show that the new encoding method is more efficient than the old one, while the cost of VS*-tree is a little higher.

6 Optimizations of Join Strategy

After the VS*-tree, candidates are already acquired for variables in $Core(Q)$. Now for all variables in $Core(Q)$, they need to be joined on the corresponding edges. The definition of joining two variables is given in Definition 7, and the details are given in Algorithm 3.

Definition 7 Join Two Variables. Given a data graph G and two variables v_1 and v_2 in query graph Q, the join process is to find all valid pairs (s,o), $s \in C(v_1)$, $o \in C(v_2)$ and $(s, p, o) \in E(G)$.(p is the predicate linking v_1 and v_2 in the query graph)

It must be emphasized that the time cost of join process is almost always the highest part when answering a query. However, it's hard to estimate the cost because the result's size is not known before two variables are joined.

To solve this problem, a heuristic algorithm named as Multi-Join is designed in this paper. Firstly, we describe the basic idea of original join strategy and analyze its shortcomings. Secondly, we introduce the new join strategy, i.e. Multi-Join, and give the pseudo code. Thirdly, we compare the new strategy with the old one and explain why the new strategy is better.

6.1 Original Strategy

In the original join strategy, join process starts from the variable with a minimal number of candidates. Next, it

Table 5 Join Result

?p1	?p2	?p3	?school
<Mike>	<Bob>	$<T_1>$	<PKU>

traverses the query graph in a BFS(Breadth First Search) order, joins remaining variables one by one and stores partial results in MRT.

In the original join process, two cases should be discussed when considering an edge:

1. two variables v_1 and v_2 are already in MRT: for each answer in MRT, consider the edge linking two variables using key-value store.
2. only one variable v_1 in MRT: for each answer in MRT, generate another candidate set of v_2 from v_1 using key-value store, and intersect the new candidate set with $C(v_2)$.

When scaling the cost of join process, we don't care about the memory cost but focus on the time cost, which is mainly constituted by two parts: intersection cost and IO cost. The intersection cost is influenced by the size and structure of the lists, which is hard to estimate. However, in real cases, time cost is almost determined by IO cost, which is determined by the times of calling key-value store(the join process calls key-value store frequently). It is impossible for us to finish the join process only in memory because the key-value store is too large to be placed in the memory instead of the disk. Therefore, we use the times of calling key-value store to model the cost in join process.

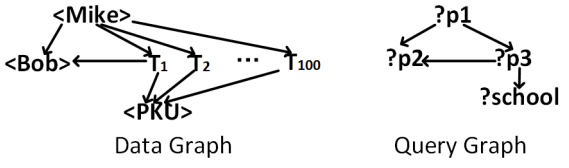

Fig. 6 query using join process

We extract the induced subgraph on $Core(Q)$ from the query in Listing 1, as well as the data graph in Figure 1. Figure 6 shows the two induced subgraphs which need to be considered in join process, next we use them to explain the idea. The corresponding join result is given in Table 5. Assume that C(?p1) = {<Mike>}, C(?p2) = {<Bob>}, C(?p3) = $\{T_1, T_2...T_{100}\}$ and C(?school) = {<PKU>}, later we will join these variables' candidates to get the final result.

Join process starts from ?p1 and the order in the original strategy is ?p2, ?p3 and ?school. First we push all candidates of ?p1 into MRT, so $|MRT| = |C(?p1)|$ now. Later we join ?p1 and ?p2(this is the second case, only ?p1 in MRT), and one answer (<Mike>, <Bob>) is added into the updated MRT(now $|MRT| = 1$). Next, we join other edges of ?p1, as in the BFS manner, ?p3 is selected and joined. When joining ?p3, the restriction is the edge linking ?p1 and ?p3 in the query graph, and we can easily see that all candidates of ?p3 satisfy this restriction.

After joining ?p3, there is totally 100 answers in MRT, and all edges of ?p1 are already visited. Later in the BFS manner, we need to deal with ?p2 and ?p3. If we manage the edge between ?p3 and ?school first, each row of MRT should be used to generate another candidate set of ?school, which means we have to call key-value store 100 times. After joining ?school, we have to consider the edge linking ?p2 and ?p3. This is a strict restriction and only one answer is left after this step, i.e. (<Mike>,<Bob>,T_1,<PKU>). When joining ?p3 and ?p2, these two variables are already in MRT and we should do as the first case. The cost in this step is also 100 times of calling key-value store, and the total cost of join process can be viewed as 202 if we use each call as a unit.

However, if we manage the edge between ?p2 and ?p3 first(also the first case), the cost of this step will be 100. Fortunately, this step will cut a lot of candidates and only an answer survives. Later when we join ?p3 and ?school, the cost is 1 and the total cost is 103. It can be seen that dealing with circle first may be better, and this is usually true. Although 103 is much smaller than 202, the final result only contains a single record. We can assume that most candidates are invalid, and what we should do is to optimize the join strategy, so as to lower the total cost. In different join strategies, intermediate result's sizes vary a lot, while the final result always remains unchanged. To optimize join strategy is to reduce the size of intermediate result, because the join cost is determined by IO cost, and the IO cost is determined by the size of MRT in each step.

Now let's formulate the join cost as follows:

Definition 8 Join Cost. Given a set of variables $v_1...v_n$ and their candidates and edges, the join cost is defined as the IO cost of joining all variables: $Join(v_1...v_n) = O(\sum_{i=1}^{m} S_i)$. In the formula, S_i represents the size of MRT when dealing with the i-th edge.(assume edge's number is m, each join step uses an edge)

It is discovered from the formula that the edges and the final result are the same in all kinds of join strategies, so the key point is to lower the size of MRT in all steps. Notice that we don't know the real result before we join a variable, so

the only choice is to design a better greedy algorithm. If we start from different sources which are not directly connected, then it will be hard to join two intermediate tables. In this paper, join strategy always starts from a center and extends to others in some kinds of searching manner(see Algorithm 2 for details).

The intermediate result is stored in MRT, and when joining two variables(variable V_1 is already in MRT while variable V_2 is not in MRT), we always generate other candidate sets of V_2 from MRT. Each row of MRT is a result of the query(containing a set of query variables, V_1, V_2, ...V_n), and each column corresponds to the mapping vertices of a query variable V_i in the data graph. For a given row pr of MRT, the i-th column of pr is a mapping vertex of the i-th query variable V_i.(see Algorithm 3 for the usage of MRT, which is a $list < vector < int >>$ struct)

6.2 Multi-Join Strategy

The example in Figure 6 has a better answer, which can help us to design a better greedy algorithm. If join process is finished on the order ?p1-?p2-?p3-?school, and when joining ?p3 we consider two edges ?p1-?p3 and ?p2-?p3 simultaneously, then we will harvest minimal join cost. To consider two edges simultaneously, we need to scan each row of MRT, and generate another two candidate sets of ?p3 from <Mike> and <Bob> separately. Assume from <Mike> we get a candidate set called $set1$, from <Bob> we get $set2$, later we should intersect $set1$, $set2$ and C(?p3). The times of calling key-value store are only 2 in this step, and surprisingly, after this step, there is only one answer in MRT instead of 100. Later the cost of joining ?school will also be 1, and the total cost is just 4 times of calling key-value store!

When considering the restriction of multiple edges simultaneously, we can greatly cut the cost down. However, try the order ?p1-?p3-?p2-?school, and it is found that the total cost is 1+100+100+1=202 if we don't adopt the new idea. If we use the new idea, i.e. considering multiple edges simultaneously, in this order the cost is also 202 because we have to consider two edges for each answer in MRT. It's out of question that the join order is the most important part of our algorithm, just like the matching order in subgraph isomorphism problem. In this subsection, we will introduce two ideas to lower the join cost: one is considering multiple edges, the other is selecting a good order. We will present our algorithms first and later analyze the cost of our algorithm and explain why the new strategy is better than the old one.

6.2.1 Multiple Edges

In our algorithm, a variable can only be joined once because each time we join a new variable var, we will consider all the edges between var and the set whose variables are already joined. The whole join algorithm is described in Algorithm 2, and Algorithm 3 is used to join two variables(or join MRT with a new variable), i.e. consider an unvisited edge.

The $P2N$ index used in our algorithm is defined as follows:

Definition 9 P2N Index. P2N index is an array kept in memory. Given a predicate's ID, P2N returns the times it occurs in data graph G. $|E(G)|$ is used to denote the number of all triples in the dataset, which can be acquired when we build the database from this dataset.

Remark 1. ***Restriction of Edges*** *If a variable connects to some variables in the joined set js, the corresponding edge set is es, then the restriction of es is stronger than the restriction of any single edge in es.*

Assumption 1. ***Result Monotony.***
The size of MRT never decreases when the join process proceeds.

The idea of considering multiple edges simultaneously is intuitively when analyzing the example in Figure 6 and it is implemented in Algorithm 3. When we join two variables, if the new variable also connects to other variables in the joined set js, then we can consider the restrictions all at once. If we only consider one edge at this time, then the cost to join other edges later will be higher following the Assumption 1, which is usually true. In addition, Remark 1 tells us that when we join MRT with a new variable, the result will be smaller after this step if we consider all the corresponding edges rather than any one of them.

Using the cost model in Definition 8 to analyze Figure 6, if we don't consider the restriction of ?p2-?p3 when we join ?p3, then the cost of joining ?p2-?p3 and ?p1-?p3 will be $S_2 + S_3$(if we join ?p2-?p3 before ?p3-?school) or $S_2 + S_4$(if we join ?p3-?school before ?p2-?p3), which is greater than $2 \times S_2$(S_2 is the size of MRT after ?p1-?p2 joined). It is obvious that considering the restrictions of multiple edges simultaneously is always better, which helps us to save a lot of time. In addition, the restrictions of multiple edges must be combined when we are going to select a join order, which is talked about in the next section.

Algorithm 2 multi-join

Input: query graph Q and variables' candidates.
Output: all valid matches.

```
1: do the preprocessing using all neighbors(not variable);
2: for all variable var in Core(Q) do
3:     initialize var's score as size of C(var);
4: use Algorithm 5 to select a variable v and push it in set js;
5: update v's neighbors' score using Algorithm 4;
6: for all id in C(v) do
7:     MRT.push_back(vector < int > (1, id));
8:     //this column in MRT corresponds to variable v;
9: while set js != Core(Q) do
10:    use Algorithm 5 to select a variable curvar;
11:    use Algorithm 3 to join MRT and C(curvar);
12:    mark this edge as dealed, push curvar into stack js;
13:    update curvar's neighbors' score using Algorithm 4;
```

Algorithm 3 join_two

Input: query graph Q, current table MRT and joining variable var.

```
1: if C(var) is empty then
2:     return false;
3: for all ans in MRT do
4:     set the temporary table tmp as empty;
5:     for all ele in ans do
6:         //ele is the i-th field of this answer, it is a result of variable
           var2;
7:         if no edge between var and var2 in Q then
8:             continue;
9:         use key-value store to generate var's another candi-
           dates(named list) from ele;
10:        if tmp is empty then
11:            tmp = intersect(list, C(var));
12:        else
13:            tmp = intersect(list, tmp);
14:    for all ele in tmp do
15:        add ele to MRT, the new field corresponding to variable
           var;
16:    if tmp is empty then
17:        remove this record;
```

6.2.2 Order Selection

The order selection is nearly the most important part of the join strategy, and perhaps even so in the whole system. The greedy algorithm is presented in Algorithm 2, and the next variable is chosen based on its score. Each variable is assigned a score, which represents its significance. In each step, only the variable *curvar* with the lowest score is chosen to be joined with MRT, and the scores of other variables(in $Core(Q) \setminus js$) are updated by the *curvar*. The update of the scores is just like the Dijkstra Algorithm which is used to solve the shortest path problem.

Now the problem is how to assign and update variables' scores. We give the solution in Algorithm 4 and Algorithm 5 and will analyze why this works in next subsection. The

Algorithm 4 update neighbors' score

Input: query graph Q, variable var, joined set js.

```
1: for all neigh in N(var) do
2:     if neigh ∈ js then
3:         continue;
4:     assume pid is this edge's ID;
5:     double prob = P2N[pid]/|E(G)|;
6:     neigh's score sco can be updated by sco = sco × prob;
```

definition of the score is given below:

Definition 10 Score. Given current joined set js, result MRT and variable $var \in Core(Q) \setminus js$, the edge set es contains all edges that connect var with js. $Score(var) = |C(var)| \times \prod(\frac{P2N(p_i)}{|E(G)|}) + |es|$, $p_i \in es$.

When the join process starts, we firstly initialize each variable's score as the size of this variable's candidates. Each time we push a new variable into js and update all its neighbors' score based on the formula in Definition 10. Notice that the item $|es|$ in this formula is not updated in this way, Algorithm 4 only adds an addend to the corresponding score. We call this item "basic item", and name another as "foresight item". The former is used to estimate the IO cost in this step, while the latter to give a rough estimate of the table size after this step(and can be used to estimate the cost in next step). Notice that the cost in each step is determined by MRT, but if multiple edges exist this time, for example, 2 edges exist, then the cost will be $2 \times |MRT|$. In a word, "basic item" is never kept with a variable's score and the score updated is not the real score for the variable.

In each step of join process, the real scores are computed for each variable by adding the final item, and we compare them to select a variable with the minimal score in Algorithm 5. Let's review the query graph in Figure 6, at the begining we assign (1, 1, 100, 1) to ?p1, ?p2, ?p3 and ?school as scores. To start, we will compute the real scores of these variables, i.e. (1, 1, 100, 1). Then we select ?p1 and update variables' scores as (-1, $\frac{1}{202}$, $\frac{10000}{202}$, 1).(Notice that ?p1 is in js now, total triple number is 202 in data graph) Next, we recompute the real scores of all variables, and the result is (-1, 1 + $\frac{1}{202}$, 1 + $\frac{10000}{202}$, INF).(INF represents the maxium value of double type) Of course we choose to join ?p2 in this turn, and later the updated scores will be (-1, -1, $\frac{10000}{202^2}$, 1). We can compute the real scores currently and join ?p3 and ?school in order.

6.3 Analysis

We have declared that considering the restrictions of multiple edges simultaneously is always better in the last

Algorithm 5 select a variable with minimal score

Input: query graph Q, joined set js.
Output: a variable to be joined.

```
set minRK = INF and minVar = −1;
for all var in Core(Q) do
    if no edge between var and js then
        set var's real score as INF, continue;
    num = 0; //this value can be adjusted
    for all neigh in N(var) do
        if neigh ∈ js then
            num + +;
    assume var's score is sco;
    set sco2 = sco + num;
    if sco2 < minRK, set minRK = sco2 and minVar = var;
return minVar;
```

subsection, here we will analyse the efficiency of order selection. In Definition 10, $P2N$ index is used to estimate the probability that a triple can satisfy the restriction of the corresponding edge, i.e. $\frac{P2N(p_i)}{|E(G)|}$. Without restrictions, the number of all results after this step will be $|MRT| \times |C(var)|$.

Given the restriction of an edge, we can estimate the result's size using probability, i.e. $|MRT| \times |C(var)| \times \frac{P2N(p_i)}{|E(G)|}$. If multiple edges exist in this step, this means that a triple in the result should satisfy all the restrictions. Following Remark 1, we regard the probabilities of different edges as a unique distribution, and multiply them to get the combined probability, which are presented in Definition 10. In our cost model, we have considered the IO cost in two steps, i.e. current step("basic item") and next step("foresight item"). The cost is formulated as the sum of the cost in two steps, just as the definition of our cost model.

Assumption 2. ***Partial Order.***
Given two edges e_1 and e_2, their variables are (v_{11},v_{12}) and (v_{21},v_{22}) respectively. If $P2N[e_1] < P2N[e_2]$, $C(v_{11}) < C(v_{21})$ and $C(v_{12}) < C(v_{22})$, then the result of joining e_1 is smaller than the result of joining e_2.

If Assumption 2 is true, then our cost model in Definition 10 can be used to represent the real IO cost in join process because it keeps the partial order. Following the assumptions above, we believe that at most time, the new algorithm is better than the old one if our cost model can represent the real case. The main difference between two algorithms is the searching order(old strategy uses BFS). It is impossible to prove that new algorithm always harvests a better performance than the old one because we have considered the cost of next step in cost model, which means that our algorithm is a heuristic method like A*. However, lots of experiments on all kinds of datasets show that the new join strategy is wonderful.

Now let's analyze the complexity of the new join strategy. It's obvious that subgraph isomorphism problem can be transformed into subgraph homomorphism problem in polynomial time. In the gStore system, subgraph homomorphism problem can be transformed into the join process in this section, and the time of transformation is also polynomial. Therefore, we can infer that the join process is an NP-Complete problem, because the subgraph isomorphism is an NPC problem.

The query graph is usually very small, so the cost of this algorithm can be neglected and we only need to focus on the IO cost. We denote Q's induced subgraph on $Core(Q)$ as TQ, size of TQ's vertex set as n and size of TQ's edge set as m. Compared with the large data graph G, we can view n and m as constants. We denote C_i as the candidates of the i-th variable in TQ, and it's ok that the intersection of C_i and $C_j (i \neq j)$ is not empty. Notice that VS*-tree can't guarantee the size of candidates, so in the worst case, we only ensure that $|C_i| \leqslant |V(G)|$.

Observation 1. *Graphs in real life are sparse, connected and the number of predicates won't be too large, so it's guaranteed that $|V(G)|^2 \gg |E(G)| > |V(G)| \gg |P(G)|$. ($E(G)$ is the set of all edges, while $|P(G)|$ is the number of all different predicates in G)*

Assumption 3. *The worst case of our algorithm occurs when each predicate in data graph G shares equal number of edges(i.e. triples), then $\frac{P2N(p_i)}{|E(G)|} = \frac{1}{|P(G)|}$.*

Following Observation 1 and Assumption 3, we can estimate the upper bound of join cost in each step. For example, the cost of the first step is $min|C_i|$, and the upper bound is $|V(G)|$. After the first step, the upper bound of $|MRT|$ is $|V(G)| \times |V(G)| \times \frac{1}{|P(G)|}$ by our cost model. As a result, if we don't consider multiple edges when joining a new variable, the cost in step i will be $\frac{|V(G)|^{i+1}}{|P(G)|^i}$, $i = 0...n-2$.

Here we only consider $n-1$ edges in TQ, but there are $m (m \geqslant n)$ edges in TQ. In our algorithm, we must consider the extra edges as multiple edges when joining a new variable. Following Assumption 1, the worst case is that these extra edges are all considered when joining the final variable, and the IO cost will be $(m-n+1) \times \frac{|V(G)|^{n-1}}{|P(G)|^{n-2}}$. Now we sum all and the total cost in worst case will be $\sum_{i=0}^{n-2} \frac{|V(G)|^{i+1}}{|P(G)|^i} + (m-n+1) \times \frac{|V(G)|^{n-1}}{|P(G)|^{n-2}}$, so we can claim that the time cost of our algorithm is $O((m-n+1) \times \frac{|V(G)|^{n-1}}{|P(G)|^{n-2}})$. The memory cost is determined by the maxium MRT, and we can represent it as $O(n \times \frac{|V(G)^{n-1}}{|P(G)|^{n-2}})$.

7 Experiment

In this section, we evaluate our method over both real and synthetic datasets and compare it with the state-of-the-art algorithms, such as virtuoso-openlinksw [12], apache-jena [13] and the original gStore system. Our optimizations mentioned in this paper are implemented in the new gStore system, and we denote it as gstore2, while the old system is denoted as gstore1. For comparison between gstore1 and gstore2, we will compare the efficiency of VS*-tree, the cost of join process and the time cost in total. Compared with other graph database systems, i.e. virtuoso and jena, only the total cost (response time) will be compared.

7.1 Datasets & Setup

We use LUBM [14] as synthetic datasets and DBpedia as real datasets in our experiments. Statistics about RDF graphs are given in Table 6 and all the queries are given in Appendix.

Table 6 Graph Datasets.

Dataset	Edge	Predicate	Entity
LUBM100M	106,909,064	18	17,473,142
LUBM200M	213,874,370	18	34,874,223
LUBM300M	320,711,327	18	52,254,606
LUBM500M	500,000,000	18	81,342,489
DBpedia1B	1,111,481,066	124,034	139,493,254

We conducted all experiments on a computer with 2.8 GHz Intel(R) Xeon(R) processor, 4T disk and 128 GB memory running CentOS7. We finish the experiment with the latest version of other graph database systems: apache-jena 3.0.1 and virtuoso-openlinksw 7.2.

Note that the original gStore[1)] fails to run on LUBM500M and DBpedia1B; while new gStore, Jena and Virtuoso work on all datasets used in this paper. We do not report the experiment results if the systems can not work.

7.2 Comparison of VS*-tree

We compare VS*-tree's efficiency here, which is scaled by candidates' number after the filter. The result is in Figure 7, and the corresponding dataset is LUBM300M. We use all queries in this subsection, and for queries which don't use the "filter and join" framework(Q_{12}, Q_{13}, Q_{20} and Q_{21}), the numbers of their final results are used in Figure 7. Each query may have many variables that need to retrieve candidates, and we only select the first variable of a query to be displayed.

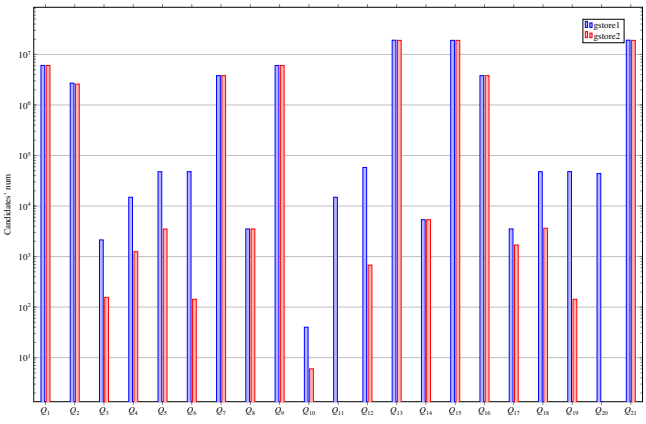

Fig. 7 Candidates' number after filtered by VS*-tree on LUBM 300M

Through the comparison, it is discovered that the new encoding method in Section 5 is never worse than the old one. Furthermore, in some queries, it reduces the number of candidates by 100 times, or even reduces the number to 0. However, for queries with a very large number of candidates, the new encoding method does not improve a lot. The reason is that the minimum number of candidates we can get is already very large in those cases, even if we consider the restrictions of the neighborhood precisely rather than use hash values to represent them roughly. In such cases, the remaining invalid candidates can only be removed in the join process.

7.3 Cost of Join Process

This subsection also uses LUBM300M as the dataset, and the queries are the same as last subsection. Here we focus on the time cost in join process, so as to verify the effectiveness of the new join strategy in Section 6. The result in Figure 8 shows that the new strategy never performs worse than the old one. Sometimes, it even helps lower the time cost by 30 times. In Q_{12}, Q_{13}, Q_{20} and Q_{21}, the cost in join process is 0 because these queries do not need the join process if using the new strategy.

The result can also prove that our cost model in Definition 10 is close to the real cost in join process. Otherwise, the new join strategy will not work well and may choose an awful edge in some steps, which results in a high cost of join process.

[1)] This is the centralized system. The distributed gStore system can run these RDF datasets with billion triples. We only compare the centralized systems in this paper.

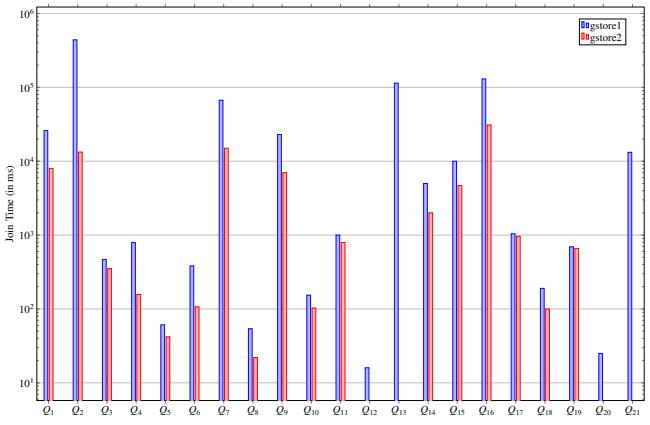

Fig. 8 Cost in the join process

7.4 Comparison of Total Cost

In the last two subsections, we only compare some important parts of the system between gstore1 and gstore2. Here we will conduct a comprehensive experiment in Figure 9, Figure 11, Figure 12 and Figure 10. Generally speaking, the new gStore system (gStore2) beats the old system (gStore1) in almost all queries of all figures, which indicates that our optimization methods are very effective. In some queries like Q_2, Q_{13} and Q_{21} of LUBM, gstore2 is more than 10 times faster than gstore1. The reason is that gstore2 uses key-value store directly to answer Q_{13} and Q_{21}, bringing a big gain. As for Q_2, new gStore doesn't need to do join process while gstore1 has to retrieve candidates for all variables and do join process, which is a heavy cost due to a large size of candidates.

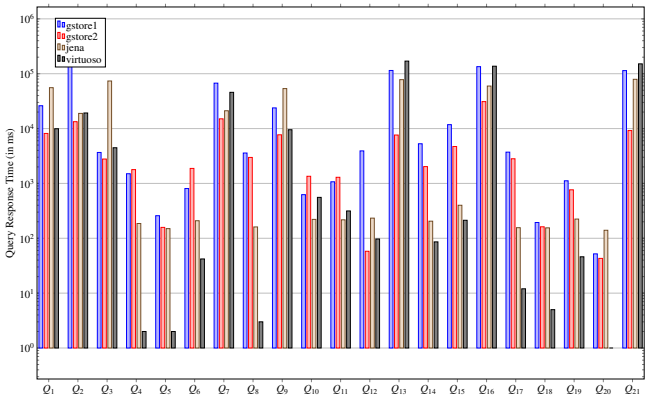

Fig. 9 Query Performance over LUBM300M

On LUBM300M and LUBM500M, gStore2 runs the fastest on 8 queries, while 1 for jena and 12 for virtuoso. However, Figures 9 and 11 shows that gStore2 wins the first place for all time-consuming queries (the response times are larger than 10 seconds) except for Q2 in LUBM500M. In other words, gStore2 is more robust to time-consuming queries. Although Virtuoso wins 12 queries, they are almost fast queries (the response time is less than 1 second). Furthermore, gstore2 wins virtuoso a lot in some queries: Q_3, Q_7 and Q_{16}(these contain circles), Q_{13} and Q_{21}(these only contain one triple).

Figure 10 shows that our system (gStore 2) has a strong scalability because the time of query processing doesn't grow exponentially as data size grows.(Q_{16} is selected because its time cost is the highest) Generally, processing time grows with no more than two times the speed of dataset's size. In addition, our system can run very large datasets(like LUBM500M) within 100s, which is better than other systems(gstore1, jena and virtuoso) because none of them can answer all queries within 100s on LUBM500M.

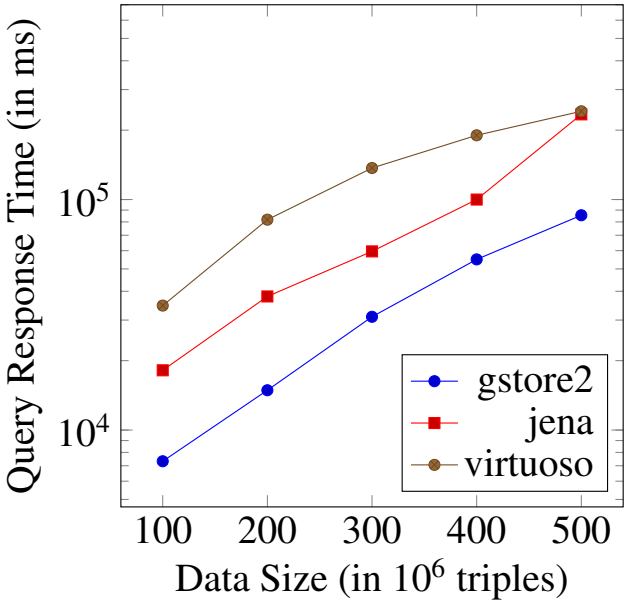

Fig. 10 Performance of Q_{16} over LUBM

On DBpedia1B, gstore2 always performs better except for Q_0, which is a star graph with only one variable. Our system uses too much time in VS*-tree when answering Q_0, while the final result number of this query is only 10. However, gStore2 runs the fastest on other queries.

To sum up, the new gStore system has advantages on 4 kinds of queries:

- queries which only contain 1 triple
- queries which have some satellites(see the definition of $Sat(Q)$ in Table 2 for the meaning of a satellite)
- queries on which cost of retrieving is much smaller than cost of join
- queries which contain circles

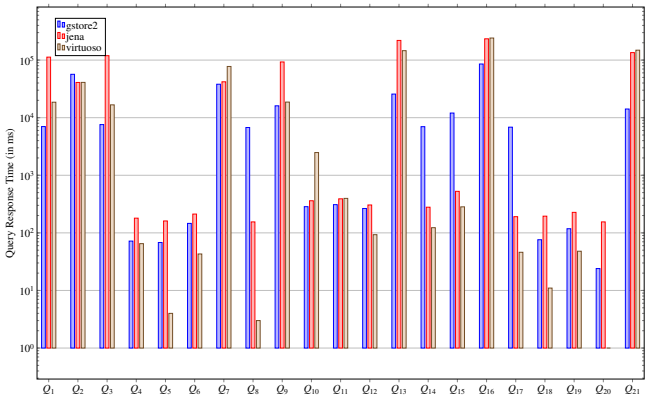

Fig. 11 Query Performance over LUBM500M

Besides, our system has a good scalability, which is important for real-life applications.

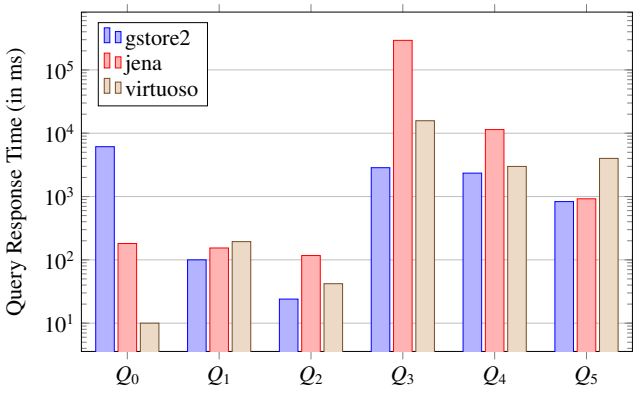

Fig. 12 Query Performance over DBpedia 1B

8 Related Work

Ullmann [15] and VF2 [16] are the two early efforts for subgraph isomorphism problem, and the key problem is how to select a good matching order. Ullmann uses depth-first search strategy, while VF2 considers the connectivity as pruning strategy. In order to speed up query response time, most subgraph search methods pre-compute some structural indices to reduce the search space. They assume that the data graph is a vertex-labeled graph, and build indices based on vertex labels. For example, SPath [17] constructs an index by neighborhood signature that summarizes vertex-labels within the k-neighborhood subgraph of each vertex. In addition, SPath generates a matching order based on the infrequent-paths first strategy to resolve the limitations of only considering vertices and edges. For each vertex in a data graph, NOVA [18] uses a vector to store the label distribution of its neighborhood vertices. ASP [19] divides all edges in a data graph into several classes according to vertex labels and uses bitmap structures to indices them; SSP [20] extends ASP and proposes some optimizations to further improve query performance.

The problem of existing methods is the super-linear space complexity of the index structure. Jinsoo Lee et al. [21] re-implements some of the above methods and provides a fair comparison of them. Then, they present a solution, called $\text{Turbo}_{\text{ISO}}$ [22] to define a concept of the *neighborhood equivalence class(NEC)*. All query vertices in the same NEC have the same matching data vertices. Hence, when $\text{Turbo}_{\text{ISO}}$ finds all subgraph matches, only combinations for each NEC are generated. In a word, $\text{Turbo}_{\text{ISO}}$ merges similar vertices and enumerates all paths to find the best matching order.

Turbo_{HOM++} [23] further extends $\text{Turbo}_{\text{ISO}}$ to handle SPARQL queries over RDF graphs. BoostIso [24] extends the concept of neighborhood equivalence class in the data graph and defines four types of relationships between vertices in the data graph to further reduce duplicate computation. gStore [1] uses the idea of graph encoding to find candidates, while Nauty [25] prefinds all automorphism within a data graph, so as to lower the cost of subgraph isomorphism. QuickSI [26] generates a matching order based on the infrequent-labels first strategy.

Recently, CFL-Match [27] defines a Core-Forest-Leaf decomposition strategy and a cost model to select a good matching order. It divides the query graph into core part(with circles), forest part(no circles, nodes' degree>1) and leaf part(nodes' degree=1). Subgraph matching should always begin from core part, then forest part, and leaf part at last. The matching order within each part is determined by the cost model, which computes the cost of each path in each step. The path selected each time is the one with minimal growth of result size, and after dealing with this path, a new growing path will be selected.

Compared with CFL-Match, our system doesn't think that circles should always be handled first for a specific query, especially when edges contained in this circle are all frequent in the data graph. Instead, we design a cost model to estimate the cost of each step in join process. In addition, our cost model doesn't estimate cost for each path, but analyses the cost for each edge and considers the cost of next and current steps. Adjusting the cost model is more flexible after joining an edge than joining a path. Furthermore, for leaf part, we don't prepare candidates for them but generate their solutions directly at last. To the best of our knowledge, we are the

first to propose an efficient cost model and join strategy in SPARQL query answering problem.

9 Conclusion

In this work, we redesign the gStore system to speed up SPARQL query processing. We study the subgraph search problem over a large general graph and propose several important strategies to optimize the original system. Firstly, we choose different query plan for query graphs with varied structures. In addition, the encoding method is improved to reduce the number of candidates for variables in the query graph. Most important of all, a cost model is built in Section 6 and an efficient join method is used to lower the main part of the cost.

Extensive experiments over large datasets confirm the superiority of our solutions. In the future, we will go on optimizing the gStore system and still keep it as open source. In Section 7 we have discovered that gStore performs worse than other systems on some queries due to the huge cost of retrieving candidates. Therefore, we will find a solution for this problem later. Furthermore, optimization of the join strategy is still a key point and we won't stop working on it.

10 Acknowledgement.

This work is supported by National Natural Science Foundation of China (NSFC)-Young Excellent Talent Project under grant 61622201.

References

1. Zou L, Mo J H, Chen L, Özsu M T, and Zhao D Y. Gstore: answering SPARQL queries via subgraph matching. In: Proceedings of VLDB Endowment, 2011, 4(8):482–493.
2. Bollacker K D, Cook R P, and Tufts P. Freebase: a shared database of structured general human knowledge. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, 2007, 1962–1963.
3. Lehmann J, Isele R, Jakob M, Jentzsch A, Kontokostas D, Mendes P N, Hellmann S, Morsey M, Kleef P V, Auer S, Bizer C. Dbpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web, 2015, 6(2):167–195.
4. Neumann T, Weikum G. RDF-3X: a RISC-style engine for RDF. In: Proceedings of VLDB Endowment, 2008, 1(1):647–659.
5. Neumann T, Weikum G. The RDF-3X engine for scalable management of RDF data. The VLDB Journal, 2009, 19(1):91–113.
6. Weiss C, Karras P, Bernstein A. Hexastore: sextuple indexing for semantic web data management. In: Proceedings of VLDB Endowment, 2008, 1(1):1008–1019.
7. Zeng K, Yang J C, Wang H X, Shao B, and Wang Z Y. A distributed graph engine for web scale RDF data. In: Proceedings of VLDB Endowment, 2013, 6(4):265–276.
8. Zou L, Özsu M T, Chen L, Shen X C, Huang R Z, Zhao D Y. Gstore: a graph-based SPARQL query engine. The VLDB Journal, 2014, 23(4):565–590.
9. Aluç G. Workload matters: a robust approach to physical RDF database design. PhD thesis, University of Waterloo, 2015.
10. Ingalalli V, Ienco D, Poncelet P, Villata S. Querying RDF data using a multigraph-based approach. In: Proceedings of Extending Database Technology, 2016, 245–256.
11. Nabti C, Seba H. A simple algorithm for subgraph queries in big graphs. arXiv preprint arXiv:1703.05547, 2017.
12. Erling O. Virtuoso, a hybrid rdbms/graph column store. IEEE Data Engineering Bulletin, 2012, 35(1):3–8.
13. Mcbride B. Jena: a semantic web toolkit. IEEE Educational Activities Department, 2002.
14. Guo Y B, Pan Z X, Heflin J. Lubm: a benchmark for owl knowledge base systems. Web Semantics Science Services & Agents on the World Wide Web, 2005, 3(2–3):158–182.
15. Ullmann J R. An algorithm for subgraph isomorphism. Journal of the ACM, 1976, 23(1).
16. Cordella L P, Foggia P, Sansone C, Vento M. A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004, 26(10).
17. Zhao P X, Han J W. On graph query optimization in large networks. In: Proceedings of VLDB Endowment, 2010.
18. Zhu K, Zhang Y, Lin X M, Zhu G P, Wang W. Nova: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs. In: Proceedings of Database Systems for Advanced Applications, 2010.
19. Peng P, Zou L, Chen L, Lin X M, Zhao D Y. Subgraph search over massive disk resident graphs. In: Proceedings of Scientific and Statistical Database Management Conference, 2011.
20. Peng P, Zou L, Chen L, Lin X M, Zhao D Y. Answering subgraph queries over massive disk resident graphs. World Wide Web, 2016, 19(3):417–448.
21. Lee J, Han W S, Kasperovics R, Lee J H. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: Proceedings of VLDB Endowment, 2012, 6(2):133–144.
22. Han W S, Lee J, Lee J H. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of SIGMOD Conference, 2013, pages 337–348.
23. Kim J, Shin H, Han W S, Hong S, Chafi H. Taming subgraph isomorphism for RDF query processing. In: Proceedings of VLDB Endowment, 2015, 8(11):1238–1249.
24. Ren X G, Wang J H. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. In: Proceedings of VLDB Endowment, 2015, 8(5):617–628.

25. McKay B D, Piperno A. Practical graph isomorphism, {II}. Journal of Symbolic Computation, 2014, 60(0):94–112.
26. Shang H C, Zhang Y, Lin X M, Yu J X. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In: Proceedings of VLDB Endowment, 2008, 1(1):364–375.
27. Bi F, Chang L J, Lin X M, Qin L, Zhang W J. Efficient subgraph matching by postponing cartesian products. In: Proceedings of the 2016 International Conference on Management of Data, 2016, 1199–1214.
28. Atre M, Chaoji V, Zaki M J, Hendler J A. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In: Proceedings of International Conference on World Wide Web, 2010, 41–50.
29. Peng P, Zou L, Özsu M T, Chen L, Zhao D Y. Processing SPARQL queries over distributed RDF graphs. The VLDB Journal, 2016.

11 Appendix

11.1 LUBM queries

This queries come from two places. $Q_1 \sim Q7$ come from [28] and [29]. $Q_8 \sim Q_{21}$ come from [8] and [14].

Listing 6 LUBM Q_1

```
select ?x where
{
?x      <rdf:type>      <ub:GraduateStudent>.
?y      <rdf:type>      <ub:University>.
?z      <rdf:type>      <ub:Department>.
?x      <ub:memberOf> ?z.
?z      <ub:subOrganizationOf>  ?y.
?x      <ub:undergraduateDegreeFrom>    ?y.
}
```

Listing 7 LUBM Q_2

```
select ?x where
{
?x      <rdf:type>      <ub:Course>.
?x      <ub:name>       ?y.
}
```

Listing 8 LUBM Q_3

```
select ?x where
{
?x      <rdf:type>      <ub:UndergraduateStudent>.
?y      <rdf:type>      <ub:University>.
?z      <rdf:type>      <ub:Department>.
?x      <ub:memberOf> ?z.
?z      <ub:subOrganizationOf>  ?y.
?x      <b:undergraduateDegreeFrom>     ?y.
}
```

Listing 9 LUBM Q_4

```
select ?x ?y1 ?y2 ?y3 where
{
?x      <ub:worksFor>   <http :// www.Department0.
         University0.edu>.
?x      <rdf:type>      <ub:FullProfessor>.
?x      <ub:name>       ?y1.
?x      <ub:emailAddress>       ?y2.
?x      <ub:telephone>  ?y3.
}
```

Listing 10 LUBM Q_5

```
select ?x where
{
?x      <ub:subOrganizationOf>  <http :// www.
         Department0.University0.edu>.
?x      <rdf:type>      <ub:ResearchGroup>.
}
```

Listing 11 LUBM Q_6

```
select ?x ?y where
{
?y      <ub:subOrganizationOf>  <http :// www.
         University0.edu>.
?y      <rdf:type>      <ub:Department>.
?x      <ub:worksFor>   ?y.
?x      <rdf:type>      <ub:FullProfessor>.
}
```

Listing 12 LUBM Q_7

```
select ?x ?y ?z where
{
?x      <rdf:type>      <ub:UndergraduateStudent>.
?y      <rdf:type>      <ub:FullProfessor>.
?z      <rdf:type>      <ub:Course>.
?x      <ub:advisor>    ?y.
?x      <ub:takesCourse>        ?z.
?y      <ub:teacherOf>  ?z.
}
```

Listing 13 LUBM Q_8

```
select ?X where
{
?X      <rdf:type>      <ub:GraduateStudent>.
?X      <ub:takesCourse>        <http :// www.
         Department0.University0.edu/GraduateCourse0>.
}
```

Listing 14 LUBM Q_9

```
select ?X ?Y ?Z where
{
?X      <rdf:type>      <ub:GraduateStudent>.
?Y      <rdf:type>      <ub:University>.
?Z      <rdf:type>      <ub:Department>.
?X      <ub:memberOf> ?Z.
?Z      <ub:subOrganizationOf>  ?Y.
?X      <ub:undergraduateDegreeFrom>    ?Y.
}
```

Listing 15 LUBM Q_{10}

```
select ?X where
{
?X <rdf:type> <ub:Publication>.
?X <ub:publicationAuthor> <http :// www.
    Department0.University0.edu/
    AssistantProfessor0 >.
}
```

Listing 16 LUBM Q_{11}

```
select ?Y1 ?Y2 ?Y3 where
{
?X <rdf:type> <ub:FullProfessor>.
?X <ub:worksFor> <http :// www.Department0.
    University0.edu>.
?X <ub:name> ?Y1.
?X <ub:emailAddress> ?Y2.
?X <ub:telephone> ?Y3.
}
```

Listing 17 LUBM Q_{12}

```
select ?X where
{
?X <ub:memberOf> <http :// www.Department0.
    University0.edu>.
}
```

Listing 18 LUBM Q_{13}

```
select ?X where
{
?X <rdf:type> <ub:UndergraduateStudent>.
}
```

Listing 19 LUBM Q_{14}

```
select ?X ?Y where
{
?X <rdf:type> <ub:Student>.
?Y <rdf:type> <ub:Course>.
?X <ub:takesCourse> ?Y.
<http :// www.Department0.University0.edu/
    AssociateProfessor0> <ub:teacherOf> ?Y.
}
```

Listing 20 LUBM Q_{15}

```
select ?X where
{
?X <rdf:type> <ub:UndergraduateStudent>.
?Y <rdf:type> <ub:Department>.
?X <ub:memberOf> ?Y.
?Y <ub:subOrganizationOf> <http :// www.
    University0.edu>.
?X <ub:emailAddress> ?Z.
}
```

Listing 21 LUBM Q_{16}

```
select ?X ?Y ?Z where
{
?X <rdf:type> <ub:UndergraduateStudent>.
?Z <rdf:type> <ub:Course>.
?X <ub:advisor> ?Y.
?Y <ub:teacherOf> ?Z.
?X <ub:takesCourse> ?Z.
}
```

Listing 22 LUBM Q_{17}

```
select ?X where
{
?X <rdf:type> <ub:GraduateStudent>.
?X <ub:takesCourse> <http :// www.
    Department0.University0.edu/GraduateCourse0>.
}
```

Listing 23 LUBM Q_{18}

```
select ?X where
{
?X <rdf:type> <ub:ResearchGroup>.
?X <ub:subOrganizationOf> <http :// www.
    University0.edu>.
}
```

Listing 24 LUBM Q_{19}

```
select ?X ?Y where
{
?Y <rdf:type> <ub:Department>.
?X <ub:worksFor> ?Y.
?Y <ub:subOrganizationOf> <http :// www.
    University0.edu>.
}
```

Listing 25 LUBM Q_{20}

```
select ?X where
{
<http :// www.University0.edu> <ub:
    undergraduateDegreeFrom> ?X.
}
```

Listing 26 LUBM Q_{21}

```
select ?X where
{
?X <rdf:type> <ub:UndergraduateStudent>.
}
```

11.2 DBpedia queries

These queries are written by us, imitating queries in other benchmarks.

Listing 27 DBpedia Q_0

```
select ?v0 where
{
?v0 <http :// www.w3.org/1999/02/22-rdf-syntax-ns#type
    > <http://dbpedia.org/ class /yago/
    LanguagesOfBotswana> .
?v0 <http :// www.w3.org/1999/02/22-rdf-syntax-ns#type
    >  <http :// dbpedia.org/ class /yago/
    LanguagesOfNamibia> .
?v0 <http :// www.w3.org/1999/02/22-rdf-syntax-ns#type
    > <http://dbpedia.org/ontology/Language> .
}
```

Listing 28 DBpedia Q_1

```
select ?v0 where
{
?v0 <http :// dbpedia.org/ontology/associatedBand> <http
    :// dbpedia.org/ resource /LCD_Soundsystem> .
}
```

Listing 29 DBpedia Q_2

```
select ?v2 where
{
<http :// dbpedia.org/ resource /!! Destroy-Oh-Boy!!> <
    http://dbpedia.org/ property / title > ?v2 .
}
```

Listing 30 DBpedia Q_3

```
select ?v0 ?v2 where
{
?v0 <http :// dbpedia.org/ontology/ activeYearsStartYear
    > ?v2 .
}
```

Listing 31 DBpedia Q_4

```
select ?v0 ?v1 ?v2 where
{
?v0 <http :// dbpedia.org/ property / dateOfBirth> ?v2 .
?v1 <http :// dbpedia.org/ property / genre> ?v2 .
}
```

Listing 32 DBpedia Q_5

```
select ?v0 ?v1 ?v2 ?v3 where
{
?v0 <http :// dbpedia.org/ property / familycolor> ?v1 .
?v0 <http :// dbpedia.org/ property / glotto> ?v2 .
?v0 <http :// dbpedia.org/ property / lc> ?v3 .
}
```

Li Zeng received his BS degree in Computer Science at Peking University in 2016. Now, he is a master student of Peking University majoring in Computer Science. His research interests include graph database and data management.

Lei Zou, awardee of the NSFC Excellent Young Scholars Program in 2016, received his BS degree and Ph.D. degree in Computer Science at Huazhong University of Science and Technology in 2003 and 2009, respectively. Now, he is an associate professor in Peking University. His research interests include graph database, knowledge graph data management.