A Graph-based RDF Triple Store

Xuchuan Shen[1], Lei Zou[1],M. Tamer Özsu[2], Lei Chen[3], Youhuan Li[1], Shuo Han [1], Dongyan Zhao[1]

[1]Peking University, Beijing, China
{shenxuchuan,zoulei,liyouhuan,hanshuo,zhaody}@pku.edu.cn
[2]University of Waterloo, Canada
tamer.ozsu@uwaterloo.ca
[3]Hong Kong University of Sci. & Tech., Hong Kong
leichen@cse.ust.hk

***Abstract*— In this demonstration, we present the gStore RDF triple store. gStore is based on graph encoding and subgraph match, distinct from many other systems. More importantly, it can handle, in a uniform manner, different data types (strings and numerical data) and SPARQL queries with wildcards, aggregate, range and top-k operators over dynamic RDF datasets. We will demonstrate the main features of our system, show how to search Wikipedia documents using gStore and how to build users' own application using gStore through C++/Java API.**

I. Introduction

Although RDF data management has been studied in the past decade, most early solutions failed to scale up to large RDF repositories. More current native RDF stores, such as RDF-3x [1], Hexastore [2] and SW-store [3] can handle large repositories, but are restricted in the types of queries they can handle (they cannot handle wildcard queries, range and top-k queries, aggregates) and many have difficulties dealing with frequent updates to the data. All of these systems map RDF data to a relational model and translate SPARQL queries to SQL. There are also extensions to commercial relational DBMSs (e.g., DB2-RDF [4]) that provide support for RDF data. The large number of joins (sometimes self-joins) and the difficulty of optimizing them is one reason for their scalability issues.

gStore [5], [6] 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), where each vertex corresponds to a subject or an object. We also represent a given SPARQL query by a query graph Q (Figure 2). 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 tree with a number of associated pruning techniques to speed up subgraph matching. Technical details of gStore have been published before [5], [6] and are summarized in Section III of this paper. In this demonstration paper we present the prototype system architecture and functionality. The associated demonstration will cover execution of SPARQL queries over RDF data extracted from Wikipedia, and the associated optimization techniques that the gStore optimizer uses.

II. System Architecture

In this section, we present the system architecture, as illustrated in Figure 3. The whole system consists of an offline part and an online part.

The offline process is to store a RDF dataset and build the VS*-tree index. We briefly 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. 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.

The online system consists of four modules. A SPARQL statement is the input to the SPARQL parser, which is gen-

TABLE I
RDF Data

Subject	Property	Object
mdb:film/2014	rdfs:label	"The_Shining"
mdb:film/2014	movie:initial_release_date	"1980-05-23"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	mdb:actor/30013
mdb:film/2014	y:hasDuration	7140.0$#s
mdb:film/2014	y:hasBudget	22000000#$
mdb:film/2014	y:hasImdb	"0081505"
mdb:actor/29704	movie:actor_name	"Jack_Nicholson"
mdb:actor/29704	y:wasBornIn	1937-04-22
mdb:actor/29704	rdf:type	movie:actor
mdb:director/8476	movie:director_name	"Stanley_Kubrick"
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A_Clockwork_Orange"
mdb:film/424	y:hasBudget	22000000#$
mdb:film/424	y:hasBoxOffice	26589355#$
mdb:film/424	movie:director	mdb:director/8476
mdb:film/424	rdfs:label	"Spartacus"
mdb:film/424	y:hasBudget	12000000#$
mdb:film/424	y:hasBoxOffice	60000000#$
geo:2635167	gn:name	"United_Kingdom"
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United_Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425
lexvo:iso639-3/eng	rdfs:label	"English"
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn

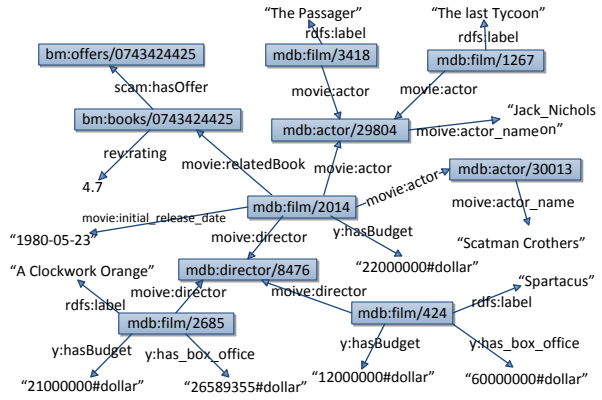

Fig. 1. An RDF graph G

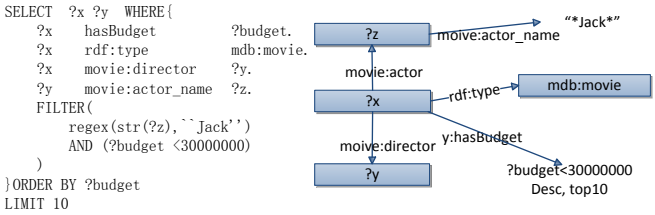

Fig. 2. SPARQL and Query Graph Q

erated by a parser generator library called ANTLR3[1]. 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.

The online query evaluation process consists of two steps: filtering and joining. First, we generate the candidates for each query node using VS*-tree. Then, applying a depth-first search strategy, we perform the multi-way join over these candidate lists to find the subgraph matches of SPARQL query Q over RDF graph G.

III. Techniques

In this section, we briefly discuss the techniques used in gStore system; full details are given in elsewhere [5], [6]. According to our framework in Section II, we solve the SPARQL query processing by subgraph matching over the signature graph. A key issue is that the proposed encoding and pruning strategies should support, in a uniform manner, different kinds of data (such as strings and numeric data), and SPARQL queries with different operators . We discuss the encoding and pruning methods in Section III-A. Another technical issue is the index structure, which is discussed in Section III-B. We also present some system-oriented optimization, such as index caching strategy and multicore-based query optimization.

A. Encoding Techniques

In gStore, answering SPARQL queries is equivalent to finding subgraph matches of query graph Q over RDF graph G. If vertex v (in query Q) can match vertex u (in RDF graph G), each neighbor vertex and each adjacent edge of v should match to some neighbor vertex and some adjacent edge of u.

[1]http://www.antlr3.org/

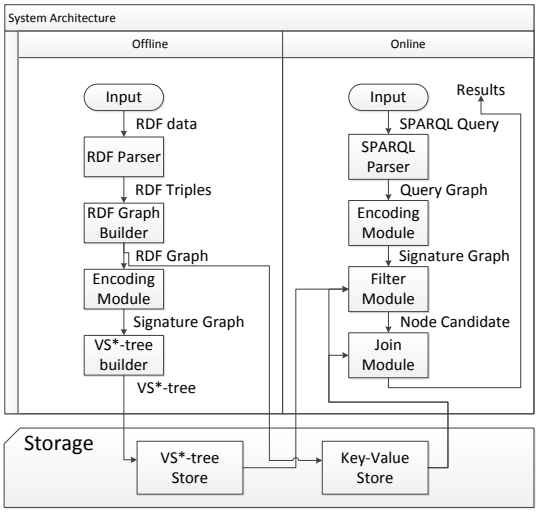

Fig. 3. System Architecture

Thus, given a vertex u in G, we encode each of its adjacent edge labels and the corresponding neighbor vertex labels into bitstrings, denoted as $vSig(u)$. We encode query Q with the same encoding method. Consequently, the match between Q and G can be verified by simply checking the match between corresponding encoded bitstrings.

Given a vertex u, we encode each of its adjacent edges $e(eLabel, nLabel)$ into a bitstring, where $eLabel$ is the edge label and $nLabel$ is the vertex label. This bitstring is called *edge signature* (i.e., $eSig(e)$). It has two parts: $eSig(e).e$, $eSig(e).n$. The first part $eSig(e).e$ (M bits) denotes the edge label (i.e., $eLabel$) and the second part $eSig(e).n$ (N bits) denotes the neighbor vertex label (i.e., $nLabel$). The code of $vSig(u)$ is formed by performing OR operator over all $eSig(e)$. Figure 4 illustrates the process.

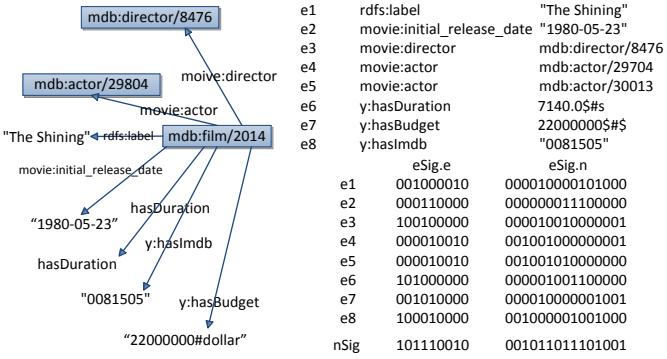

Fig. 4. Encoding Technique

1) Computing $eSig(e).e$: Given an RDF repository, let $|P|$ denote the number of different properties. If $|P|$ is small, we set $|eSig(e).e| = |P|$, where $|eSig(e).e|$ denotes the length of the bitstring, and build a 1-to-1 mapping between the property and the bit position. If $|P|$ is large, we resort to the hashing technique. Let $|eSig(e).e| = M$. Using an appropriate hash function, we set m out of M bits in $eSig(e).e$ to be '1'. Specifically, we employ m different string hash functions H_i ($i = 1, ..., m$), such as BKDR and AP hash functions. For each

hash function H_i, we set the (H_i(*eLabel*) MOD M)-th bit in $eSig(e).e$ to be '1', where H_i(*eLabel*) denotes the hash function value. We have discussed the parameter setting problem in our research paper [6].

2) Computing $eSig(e).n$: A neighbor vertex label (i.e., *nLabel*) has three different kinds of data: URI, string and numerical data, where the latter two are called "literals". We suppose that the number of bits in $e.Sig(e).e$ is N.

If *nLabel* is URI, we have the same encoding strategy with $eSig(e).e$, i.e., setting n out of N bits in $eSig(e).n$ to be '1' by the n different hash string functions.

If *nLabel* is a string, in order to support wildcard operators, we have the following encoding strategy: We first represent *nLabel* by a set of q-grams [7], where an q-gram is a subsequence of q characters from a given string. For example, "Jack_Nicholson" is represented by a set of 3-grams: {(Jac),(ack),(ck_),...,(son)}. Then, we use a string hash function H for each q-gram g to obtain $H(g)$. We set the ($H(g)$ MOD N)-th bit in $eSig(e).n$ to be '1'. We also use n different hash functions for each q-gram. Finally, the string's hash code is formed by performing bitwise OR over all q-gram's hash codes. Figure 5 demonstrates the whole process.

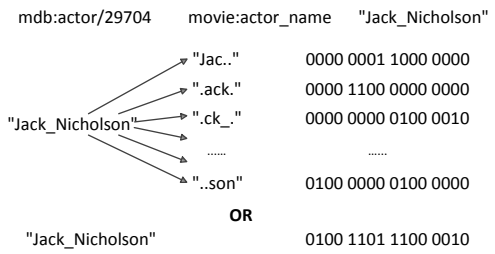

Fig. 5. Encoding Strings

Consider the wildcard filter in the running example, i.e., regex(str(?z),"Jack"); it means that the matching vertex in RDF graph should contain "Jack". Obviously, if a matching vertex v contains "Jack", let X and Y denote the codes of "Jack" and v, respectively, it is straightforward to known $X \wedge Y = X$, where "$\wedge$" denotes the bitwise ADD operator. Otherwise, v does not contain "jack"; this is the basic idea of our pruning techniques, as illustrated in the following theorem.

Theorem 1. *Given a wildcard filter W (in SPARQL) and a string label nLabel (in RDF graph), if $c(W) \wedge c(nLabel) \neq c(W)$, where $c(label)$ and $c(W)$ denotes the corresponding bitstring codes, nLabel cannot contain W.*

If *nLabel* is numerical data, in order to support SPARQL with the range operators (such as "*budget* > 11 AND *budget* < 13"), we work as follows. The most interesting result is that our method uses the same bitwise ADD operator for pruning in range queries. In other words, gStore supports SPARQL queries with different operators (wildcard and range queries) in a uniform manner.

Suppose that a property P is a numeric property, i.e., its property values are numeric data. Let $[L, U]$ denote the value range of the property P, where L and U denote the lower bound and the upper bound, respectively. Let us consider the numeric property "budget" and suppose that its value range is $[0, 64]$. We divide the range into two equal parts, as shown in Figure 6. If *nLabel* ≤ 32, the codes in the first layer is "10"; otherwise, it is "01". Then, in the second layer, we divide the range into 2^2 (=4) equal parts. The code depends on the position of *nLabel* with regard to the four parts. For example, if *nLabel* = 13, it falls in the first part. So, the code is "1000". We iterate the above steps until k-th layer, where k is a user-specified parameter. Generally speaking, in the k-th layer, we divide the whole rang into 2^k equal parts. The code on the k-th layer depends on the position of *nLabel* with regard to the 2^k parts. The full code of *nLabel* is formed by linking the codes in the k layers sequentially. Figure 1 illustrates the codes of 11 and 13, denoted as $c(11)$ and $c(13)$, respectively.

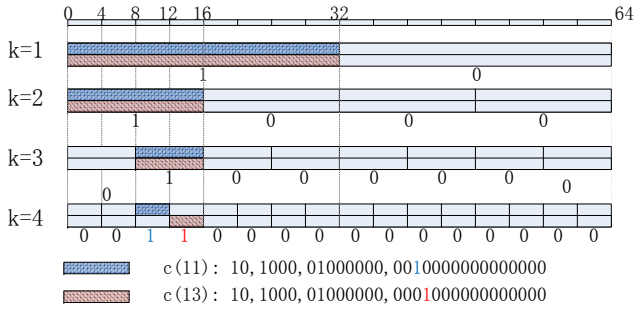

Fig. 6. Encoding Numeric Data

Next, we discuss how to encode a value range in SPARQL, such as (*budget* > 11 AND *budget* < 13). We find that the codes of 11 and 13 share the same bits in the first three layers, because they belong to the same parts in the first three layers' resolutions. They are separated in the 4-th layer. More interestingly, *nLabel* $\in [11, 13]$ also shares the same codes in the first three layers.

Generally speaking, given a value range $[L, U]$, suppose that the codes of L and U are $c(L)$ and $c(U)$, respectively. The range code is $c([L, U]) = c(L) \wedge c(U)$, where "$\wedge$" denotes the bitwise ADD operator. Furthermore, we have the following pruning theorem.

Theorem 2. *Given a value range $[L, U]$ (in SPARQL query) and a numeric data nLabel (in RDF graph), if $c(L, U) \wedge c(nLabel) \neq c(L, U)$, nLabel $\notin [L, U]$.*

The above theorem also adopts bitwise ADD operator for pruning, which is the same with Theorem 1. This is the reason why our method can support different kinds of SPARQL queries in a uniform manner.

B. Index Structure & Query Evaluation

According to the encoding technique, each node in both query graph and RDF graph are encoded into a bitstring, respectively. Theorems 1 and 2 tells us the basic pruning principle. In order to speed up filtering, we design an index, called VS*-tree, which is a height-balanced tree [8], where each node is a bitstring that corresponds to each vertex's code.

It is a multi-level summary tree where the leaves contains the vertices in the original encoded RDF graph, and higher levels summarize the structure of the level below it. In the filtering process, we visit VS*-tree from the root and judge whether the visited nodes are candidates. Once a node does not meet the condition at one level, we prune all its descendant nodes. Then, each vertex in query graph has a candidate list. Finally, applying a depth-first search strategy, we perform the multi-way join over these candidate lists to find subgraph matches.

C. Optimization—A System-Implementation Perspective

In order to improve the query performance, we adopt two optimization techniques. One is to cache the index structure (*VS**-tree except for leaf level) and the intermediate results. The second one is to employ the multi-threading technique. In the pruning technique, each thread corresponds to one query vertex. In this way, we can obtain candidate lists in parallel. Because we perform a depth-first search (DFS) strategy to join candidate lists, we can perform DFS from different candidate vertices in parallel. We will show the performance gains of different optimization techniques in our demonstration.

IV. Demonstration

1) *Searching Wikipedia using gStore*. In this demo, we illustrate how to use gStore to search Wikipedia. As we know, existing Wikipedia only supports keyword search. It is difficult to access Wikipedia based on some complex query semantics. For example, we want to find a movie Wikipage based on the director and the budget amount (such as the running example). Obviously, the keyword search cannot help that. DBpedia is a RDF repository extracting structured content (such as infobox) from Wikipedia. Our demo provides an interface for users to access DBpedia using SPARQL queries. Users can click the results to the corresponding Wikipages.

Figure 7 shows the snapshot. In the upper left corner, users can input the SPARQL query statement in the textbox. Our demo can support SPARQL 1.0 language[2] and some features of SPARQL 1.1 language[3] (such as aggregate queries). When users click the "Query" button, gStore returns the answers to the SPARQL in the bottom left textbox. Each row corresponds to a result. Since URI in DBpedia is the same with URL in Wikipedia, if users click the result, the demo will link to the corresponding Wikipage. Also, the inforbox of the Wikipage is also shown in right upper textbox, and the structural content satisfying the query conditions is highlighted.

2) *Building Users' Own Application Using gStore*. Furthermore, gStore provides a Java/C++ API. Users can build their own front-end application on their own RDF repository. The API provides the interfaces for loading RDF files, building the index and running SPARQL queries (including the updates). In the demonstration, we will show how to use our gStore in users' application software. Figure 8 shows the "Hello World" program calling gStore API.

[2]http://www.w3.org/TR/rdf-sparql-query/
[3]http://www.w3.org/TR/sparql11-overview/

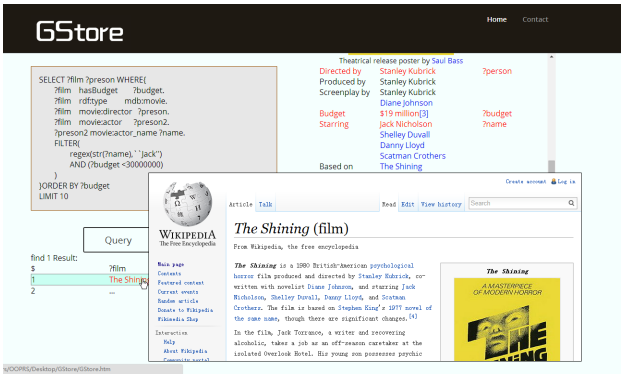

Fig. 7. Searching Wikipedia Using gStore

```
#include<gstore.h> /*include gStore C++ libiary*/
#include<iostream>
#include<string>
using namespace std;
int main()
{
    string rdf_file = "/media/disk1/yago";
    string db_folder = "/media/disk1/db_yago/";

    /* A static function provided by gStore lib */
    /* It is to build an index into db_folder for rdf data in rdf_file */
    gstore::build_index(db_folder, rdf_file);
    /* create a RDF database instance */
    gstore db_yago(db_folder);
    /* load RDF database and index*/
    db_yago.load();
    /* SPARQL query */
    string query = "select ?x1 where { ?x1  rdf:type    <wordnet_actor_109765278> }";
    /* Query Evaluation */
    string answer = db_yago.query(query);
    /*print SPARQL answers*/
    cout << answer << endl;
}
```

Fig. 8. Using gStore API

3) *Demonstrating Optimization Techniques of gStore*. We demonstrate different features of gStore, such as handling different data types in a uniform manner and the low maintenance overhead for RDF updates. Also, we will show the performance gains of different optimization strategies in our demostration.

Acknowledgment

The work was supported by National Science Foundation of China (NSFC) under Grant No. 61370055 and 61272344. The corresponding author of this paper is Lei Zou (zoulei@pku.edu.cn)

References

[1] T. Neumann and G. Weikum, "RDF-3X: a risc-style engine for RDF," vol. 1, no. 1, pp. 647–659, 2008.
[2] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," vol. 1, no. 1, pp. 1008–1019, 2008.
[3] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "Sw-store: a vertically partitioned dbms for semantic web data management," vol. 18, no. 2, pp. 385–406, 2009.
[4] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient RDF store over a relational database," in *SIGMOD*, 2013, pp. 121–132.
[5] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: Answering SPARQL queries via subgraph matching," *PVLDB*, vol. 4, no. 8, pp. 482–493, 2011.
[6] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gstore: a graph-based sparql query engine," *VLDB J.*, vol. 23, no. 4, pp. 565–590, 2014.
[7] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava, "Using q-grams in a DBMS for approximate string processing," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 28–34, 2001.
[8] U. Deppisch, "S-tree: A dynamic balanced signature index for office retrieval," 1986, pp. 77–87.