gStore: Answering SPARQL Queries via Subgraph Matching *

Lei Zou[1], Jinghui Mo[1], Lei Chen[2], M. Tamer Özsu[3], Dongyan Zhao[1,4]

[1]*Peking University, China;*
[2] *Hong Kong University of Science and Technology, China;*
[3] *University of Waterloo, Canada;*
[4] *Key Laboratory of Computational Linguistics (PKU), Ministry of Education, China*
{ zoulei,mojinghui,zdy}@icst.pku.edu.cn, leichen@cse.ust.hk, tamer.ozsu@uwaterloo.ca

ABSTRACT

Due to the increasing use of RDF data, efficient processing of SPARQL queries over RDF datasets has become an important issue. However, existing solutions suffer from two limitations: 1) they cannot answer SPARQL queries with wildcards in a scalable manner; and 2) they cannot handle frequent updates in RDF repositories efficiently. Thus, most of them have to reprocess the dataset from scratch. In this paper, we propose a graph-based approach to store and query RDF data. Rather than mapping RDF triples into a relational database as most existing methods do, we store RDF data as a large graph. A SPARQL query is then converted into a corresponding subgraph matching query. In order to speed up query processing, we develop a novel index, together with some effective pruning rules and efficient search algorithms. Our method can answer exact SPARQL queries and queries with wildcards in a uniform manner. We also propose an effective maintenance algorithm to handle online updates over RDF repositories. Extensive experiments confirm the efficiency and effectiveness of our solution.

1. INTRODUCTION

The RDF (**R**esource **D**escription **F**ramework) data model was proposed for modeling Web objects as part of developing the semantic web. It has been used in various applications. For example, Yago and DBPedia extract facts from Wikipedia automatically and store them in RDF format to support structural queries over Wikipedia [19, 3]. Biologists also build RDF data collections, such as Bio2RDF (bio2rdf.org) and Uniprot RDF (dev.isb-sib.ch/projects/uniprot-rdf), for recording experimental data.

Generally speaking, RDF data can be represented as a collection of triples denoted as SPO (*subject*, *property*, *object*). A running example is given in Figure 1(a). Note that, an RDF dataset can also be modeled as a graph (called RDF graph), as shown in Figure 1(b). In order to query RDF repositories, SPARQL query language [16] has been proposed by W3C. For example, we can retrieve the names of individuals who were born on February 12, 1809 and died on April 15, 1865 from the RDF dataset by the following SPARQL query:

Q_1: *Select ?name Where { ?m <hasName> ?name. ?m <BornOn Date > "1809-02-12". ?m <DiedOnDate> "1865-04-15". }*

Although RDF data management has been studied in the past decade, most existing solutions do not scale to large RDF repositories and cannot answer complex queries efficiently. Recent studies have focused on scalable techniques for large RDF repositories (e.g. [2, 12, 13, 25, 22]). Although these existing RDF query engines, such as RDF-3x [12], Hexastore [22] and SW-store [1], are designed to address the scalability of SPARQL queries, they have some common limitations: (1) they cannot support SPARQL with wildcards in a scalable manner; and (2) it is very difficult for some existing systems to handle frequent updates in RDF repositories, forcing them to reprocess the dataset from scratch when there is an update. x-RDF-3x [15], the advanced version of RDF-3x system, can support updates, but, it still fails to support wildcard queries.

1.1 SPARQL Queries With Wildcards

In real applications, having full knowledge about a query object may not be practical; thus, it may not be possible to specify exact query criteria. For example, we may know that an important politician was born on February 12 and died on April 15, but we have no idea about his exact birth and death years. In this case, we have to perform a query with wildcards, as shown below:

Q_2:*Select ?name Where { ?m <hasName> ?name. ?m <BornOnDate> ?bd. ?m <DiedOnDate> ?dd. FILTER regex(str(?bd), "02-12"), regex(str(?dd), "04-15") }*

Although there are techniques for supporting SPARQL queries with wildcards and for managing large RDF datasets, to the best of our knowledge, no technique exists to support both, i.e., the ability to execute SPARQL queries with wildcards in a scalable manner. Existing RDF storage systems, such as Jena [23], Yars2 [11] and Sesame 2.0 [5], cannot work well in large RDF datasets (such as Yago dataset). SW-store[1], RDF-3x [12], x-RDF-3x [15] and Hexastore [22] are designed to address scalability, however, they can only support exact SPARQL queries, since they replace all literals (in RDF triples) by ids using a mapping dictionary.

1.2 Frequent Updates Over RDF Repositories

In some applications, RDF repositories are not static. For example, Yago and DBpedia datasets are continually expanding to

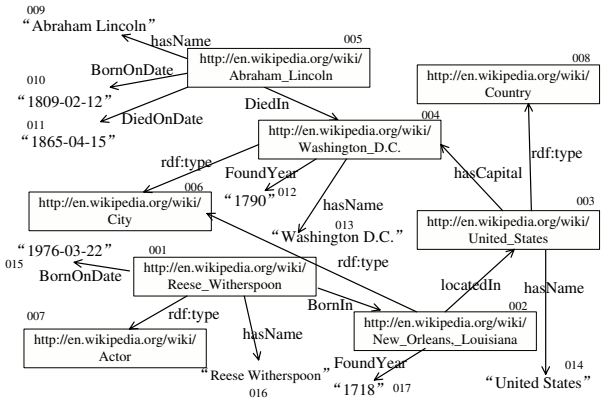
*Lei Zou, Jinghui Mo and Dongyan Zhao were supported by NSFC under Grant No.61003009 and RFDP under Grant No. 20100001120029. Lei Chen's work was supported in part by RGC NSFC JOINT Grant under Project No. N_HKUST61_2/09, and NSFC Grant No. 60736013 and 60803105. M. Tamer Özsu's work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Proceedings of the VLDB Endowment, Vol. 4, No. 8

Prefix: y= http://en.wikipedia.org/wiki/

Subject	Predict	Object
y:Abraham_Lincoln	hasName	"Abraham Lincoln"
y:Abraham_Lincoln	BornOnDate	"1809-02-12"
y:Abraham_Lincoln	DiedOnDate	1865-04-15
y:Abraham_Lincoln	DiedIn	y:Washington_D.C
y:Washington_D.C	hasName	"Washington D.C."
y:Washington_D.C	FoundYear	1790
y:Washington_D.C	rdf:type	y:city
y:United_States	hasName	"United States"
y:United_States	hasCapital	y:Washington_D.C
y:United_States	rdf:type	Country
y:Reese_Witherspoon	rdf:type	y:Actor
y:Reese_Witherspoon	BornOnDate	"1976-03-22"
y:Reese_Witherspoon	BornIn	y:New_Orleans,_Louisiana
y:Reese_Witherspoon	hasName	"ReeseWitherspoon"
y:New_Orleans,_Louisiana	FoundYear	1718
y:New_Orleans,_Louisiana	rdf:type	y:city
y:New_Orleans,_Louisiana	locatedIn	y:United_States

(a) RDF Tripes



(b) RDF Graph G

Figure 1: RDF Graph

include the newly extracted knowledge from Wikipedia. The RDF data in social networks, such as the FOAF project (foaf-project.org), are also frequently updated to represent the individuals' changing relationships. In order to support queries over such dynamic RDF datasets, query engines should be able to handle frequent updates without much maintenance overhead.

1.3 Our Approach

In this work, we treat RDF datasets from a graph database perspective. A SPARQL query is transformed into a subgraph matching query over a large RDF graph. Specifically, we can model an RDF dataset (a collection of triples) as a labeled, directed multi-edge graph (*RDF graph*), where each vertex corresponds to a subject or an object. Each triple represents a directed edge from a subject to its corresponding object. Given a subject and an object, there may exist more than one property between them, that is, multiple-edges may exist between two vertices. Consequently, an RDF graph is a multi-edge graph. Given a SPARQL query, we can also represent it by a query graph, Q. Thus, a SPARQL query can be transformed to a subgraph matching query over the RDF graph.

For example, Figure 1(b) shows an RDF graph corresponding to RDF triples in Figure 1(a). We formally define an RDF graph in Definition 2.1. Note that, the numbers next to boxes in Figure 1(b) are not vertex labels, but vertex IDs that we introduce to simplify the description. A SPARQL query can also be represented as a directed labeled graph Q (referred as *query graph* in Definition 2.2). Figure 2(a) shows the query graph corresponding to the SPARQL query Q_2. In this setting, answering SPARQL query Q reduces to finding the matches of Q in RDF graph G.

However, the characteristics of an RDF graph are different from a typical graph considered in the existing graph database literature in three aspects. First, the size of an RDF graph (i.e., the number of vertices and edges) is larger than what is considered in typical graph databases by orders of magnitude. Second, the cardinality of vertex and edge labels in an RDF graph is much larger than that in traditional graph databases. For example, a typical dataset (i.e., the AIDS dataset) used in the existing graph database work [17, 24] has 10,000 data graphs, each with an average number of 20 vertices and 25 edges. The total number of distinct vertex labels is 62. The total size of the dataset is about 5M bytes. However, the Yago RDF graph has about 500M vertices and the total size is about 3.1GB. Therefore, I/O cost becomes a key issue in RDF query processing. However, most existing subgraph query algorithms are memory-based. Third, SPARQL queries combine several attribute-like properties of the same entity, thus, they tend to contain stars as subqueries [12]. A *star query* refers to the query graph that is a star, formed by one central vertex and its neighbors.

Considering the three properties of an RDF graph, we propose a novel indexing schema to speed up query processing. Firstly, we store an RDF graph as a disk-based adjacency list table T. Then, for each *entity* or *class vertex* (Definition 2.1) in the RDF graph, according to its adjacent edge labels and neighbor vertex labels (Definition 2.1), we assign a bitstring as its *vertex signature*. In this way, an RDF graph is converted into a *data signature graph* G^* (Definition 4.3). Then, we propose a novel index (called VS*-tree) over G^*. At run time, we also encode all vertices of Q into vertex signatures, and then convert Q into its corresponding *query signature graph* Q^*. Finding all matches of Q^* over G^* will lead to all candidate matches (denoted as CL) without any false negative. Note that, we propose a novel filtering rule (Theorem 5.1) to reduce the search space in finding CL. Finally, according to CL, we can fix results by checking a small portion of the adjacency list table T.

The advantages of our methods lie in: 1) supporting exact SPARQL queries and queries with wildcards in a uniform manner; and 2) having a light maintenance overhead of our index VS*-tree, as other height-balanced trees (such as B$^+$-tree and R-tree) do.

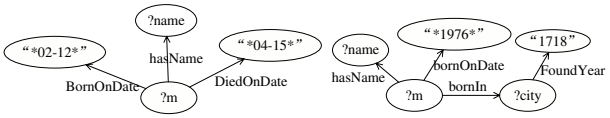
To summarize, in this work, we make the following contributions.

1. We adopt the graph model as the physical storage scheme for RDF data. Specifically, we store RDF data in disk-based adjacency lists.

2. We transform an RDF graph into a data signature graph by encoding each entity and class vertex. Then, a novel index (VS*-tree) is proposed over the data signature graph with light maintenance overhead.

3. We develop a filtering rule for subgraph query over the data signature graph, which can be seamlessly embedded into our query algorithm that answers exact SPARQL queries and queries with wildcards in a uniform manner.

4. We demonstrate through experiments that the performance of our method is superior to existing methods in answering both exact SPARQL queries and queries with wildcards, and our solutions well support online updates with small overhead.



(a) Query Q_2 (b) Query Q_3

Figure 2: Query Graphs

2. PRELIMINARIES

RDF data are a collection of triples denoted as SPO (**s**ubject, **p**roperty, **o**bject), where *subject* is an entity or a class, and *property* denotes one attribute associated to one entity or a class, and *object* is an entity, a class, or a literal value. According to the RDF standard, an entity or a class is denoted by a URI (Uniform Resource Identifier). For example, in Figure 1, "http://en.wikipedia.org/wiki/United_States" is an entity, "http://en.wikipedia.org/wiki/Country" is a class, and "United States" is a literal value. In this work, we will not distinguish between an "entity" and a "class" since we have the same operations over them. RDF data can also be modeled as an RDF graph, which is formally defined as follows:

DEFINITION 2.1. *A* RDF graph *is denoted as* $G = \langle V, L_V, E, L_E \rangle$*, where (1)* $V = V_c \cup V_e \cup V_l$ *is a collection of vertices that correspond to all subjects and objects in RDF data, where* V_c*,* V_e*, and* V_l *are collections of class vertices, entity vertices, and literal vertices, respectively. (2)* L_V *is a collection of vertex labels. Given a vertex* $v \in V_l$*, its vertex label is its literal value. Given a vertex* $v \in V_c \cup V_e$*, its vertex label is its corresponding URI. (3)* $E = (v_1, v_2)$ *is a collection of directed edges that connect the corresponding subjects and objects. (4)* L_E *is a collection of edge labels. Given an edge* $e \in E$*, its edge label is its corresponding property.*

Figure 1(b) shows an example of an RDF graph. The vertices that are denoted by boxes are entity or class vertices, and the others are literal vertices. A SPARQL query Q is also a collection of triples. However, some triples in Q have *parameters* or *wildcards*. In Q_2 (in Section 1), "?m" is a parameter and "?dd" in FILTER(regx(?dd,"04-15")) is called a wildcard. Thus, as shown in Figure2(a), we can rewrite "?dd" and FILTER(regx(?dd,"04-15")) as "*04-15*".

DEFINITION 2.2. *A* query graph *is denoted as* $Q = \langle V, L_V, E, L_E \rangle$*, where (1)* $V = V_c \cup V_e \cup V_l \cup V_p \cup V_w$ *is collection of vertices that correspond to all subjects and objects in a SPARQL query, where* V_p *and* V_w *are collections of parameter vertices and wildcard vertices, respectively, and* V_c *and* V_e *and* V_l *are defined in Definition 2.1. (2)* L_V *is a collection of vertex labels. For a vertex* $v \in V_p$*, its vertex label is* ϕ*. The vertex label of a vertex* $v \in V_w$ *is the substring without the wildcard. A vertex* $v \in V_c \cup V_e \cup V_l$ *is defined in Definition 2.1. (3) E and* L_E *are defined in Definition 2.1.*

Figure 2(a) shows a query example that corresponds to Example 2. "*02-12*" is a wildcard vertex, and its label is "02-12". "?m" is a parameter vertex and its label is ϕ.

DEFINITION 2.3. *Consider an RDF graph G and a query graph Q that has n vertices* $\{v_1, ..., v_n\}$*. A set of n distinct vertices* $\{u_1, ..., u_n\}$ *in G is said to be a* match *of Q, if and only if the following conditions hold:*

1. *If* v_i *is a literal vertex,* v_i *and* u_i *have the same literal value;*
2. *If* v_i *is an entity or class vertex,* v_i *and* u_i *have the same URI;*
3. *If* v_i *is a parameter vertex, there is no constraint over* u_i*;*
4. *If* v_i *is a wildcard vertex,* v_i *is a substring of* u_i *and* u_i *is a literal value.*
5. *If there is an edge from* v_i *to* v_j *in Q with the property p, there is also an edge from* u_i *to* u_j *in G with the same property p.*

Given a query graph Q_2 in Figure 2(a), vertices (005,009,010,011) in RDF graph G form a match of Q_2. Answering a SPARQL query is equivalent to finding all matches of its corresponding query graph in RDF graph.

DEFINITION 2.4. (Problem Definition) *Given a query graph Q over an RDF graph G, find all matches of Q over G according to Definition 2.3.*

3. OVERVIEW OF gStore

Our general framework consists of both offline and online processes. During offline processing, we first represent an RDF dataset by an RDF graph G and store it by its adjacency list table T, as shown in Figure 4. Then, we encode each entity and class vertex into a bitstring (called *vertex signature*). The encoding technique will be discussed in Section 4. According to RDF graph's structure, we link these vertex signatures to form a *data signature graph* G^*, in which, each vertex corresponds to a class or an entity vertex in the RDF graph, as shown in Figure 3. Specifically, G^* is induced by all entity and class vertices in G together with the edges whose endpoints are either entity or class vertices. At run time, we can also represent a SPARQL query by a query graph Q and encode it into a *query signature graph* Q^*. Then, finding matches of Q^* over G^* leads to candidates (denoted as CL). Finally, we verify each candidate by checking adjacency list table T. Note that, the matches of Q over G are denoted as RS.

Figure 3 shows an example of a data signature graph G^*, which corresponds to RDF graph G in Figure 1(b). Note that each entity and class vertex in G is encoded into a signature. We also encode query Q_3 (in Figure 2(b)) into a query signature graph Q^*, as shown in Figure 3. There is only one match of Q^* over G^*, that is $CL = \{(001, 002)\}$. Finally, by checking the adjacency list T (in Figure 4), we can find that $(001, 002)$ is also a match of Q over G.

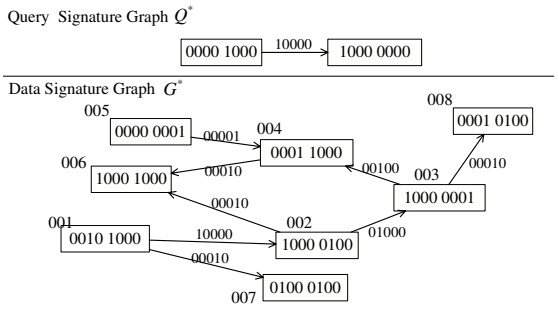


Figure 3: Signature Graphs

Finding matches of Q^* over G^* is known to be NP-hard since it is analogous to subgraph isomorphism. Therefore, we propose an index and filtering strategy to reduce the search space over which we do matching. Reducing the search space has been considered in other works as well (eg. [17, 24]).

According to this framework, there are two issues to be addressed. First, the encoding technique should guarantee that there are no *no-false-negatives*, i.e., $RS \subseteq CL$. Second, an efficient subgraph matching algorithm is required to find matches of Q^* over G^*. To address the first issue, we propose a coding technique in Section 4. For the second issue, we design novel index structures (called VS and VS*-trees) and query algorithms in Sections 5 and 6.

4. STORAGE SCHEME AND ENCODING TECHNIQUE

We propose a graph-based storage scheme for RDF data. Specifically, we store an RDF graph G using a disk-based adjacency list

table. Each (class or entity) vertex u is represented by an adjacency list, whose format is $[vID, vLabel, adjList]$, where vID is the vertex ID, $vLabel$ is the corresponding URI, and $adjList$ is the list of its *outgoing* edges and the corresponding neighbor vertices. Formally, $adjList = \{(eLabel, nLabel)^+\}$, where $eLabel$ is v's outgoing edge label that corresponds to some property and $nLabel$ is v's neighbor vertex label. Vertex labels and edge labels of an RDF graph are defined in Definition 2.1. Figure 4 shows the corresponding adjacency list table (T) for the RDF graph in Figure 1(b).

Prefix: y= http://en.wikipedia.org/wiki/

vID	vLabel	adjList {(eLabel, nLabel)+}
001	y:Abraham_Lincoln	(hasName, "Abraham Lincoln") (BornOnDate, "1809-02-12"), (DiedOnDate, "1865-04-15") (DiedIn, y:Washington_D.C)
002	y:Washington_D.C	(hasName, "Washington D.C.") (FoundYear , "1790") (rdf:type, y:city)
003	y:United_States	(hasName, "United States") (hasCapital,y:Washington_D.C) (rdf:type, y:country)
004	y:Reese_Witherspoon	(hasName, "ReeseWitherspoon") (BornOnDate, "1976-03-22") (hasCapital, y:New_Orleans,_Louisiana) (rdf:type, y:Actor)
005	y:New_Orleans,_Louisiana	(FoundYear, "1718"), (locatedIn, y:United_States) (rdf:type, y:city)

Figure 4: Disk-based Adjacency List Table T

According to Definition 2.3, if a vertex v (in query Q) can match a 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. Thus, given a vertex u in G, we encode each of its adjacent edge labels and the corresponding neighbor vertex labels into bitstrings. We encode query Q with the same encoding method. In this way, we can verify the match between Q and G by simply checking the match between corresponding encoded bitstrings. A similar encoding strategy has been proposed in our earlier work [26]. The differences are that in this work we encode strings to their bitstring representation, while in the previous work we encode the eigenvalues of the adjacency matrix.

As mentioned earlier, each row in table T corresponds to an entity vertex or a class vertex. We encode each of its outgoing edge labels and the corresponding neighbor vertex label into a bitstring. Specifically, we first encode each adjacent edge $e(eLabel, nLabel)$ into a bitstring. This bitstring is called *edge signature* (i.e., $eSig(e)$).

DEFINITION 4.1. *Given an adjacent edge $e(eLabel, nLabel)$, the* edge signature *of e is a bitstring, denoted as $eSig(e)$, which 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).*

Given an edge $e(eLabel, nLabel)$, we discuss how to generate $eSig(e).e$ and $eSig(e).n$, respectively. Let $|eSig(e).e| = M$. Using some appropriate hash functions, we set m out of M bits in $eSig(e).e$ to be '1'. Specifically, in our implementation, we employ m different string hash functions H_i ($i = 1, ..., m$), such as BKDR and AP hash functions [6]. 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.

In order to encode neighbor vertex label $nLabel$ into $eSig(e).n$, we adopt the following technique. We first represent $nLabel$ by a set of n-grams [9], where an n-gram is a subsequence of n characters from a given string. For example, "1809-02-12" can be represented by a set of 3-grams: {(180),(809),(09-),...,(-12)}. Then, we adopt some string hash function H for each n-gram g. We use $H(g)$ to denote hash value of g. Finally, we set the ($H(g)$ MOD N)-th bit in $eSig(e).n$ to be '1'. The above encoding technique introduces some parameters, such as M, m, N and n. We discuss the parameter settings in Appendix D. Figure 10(a) (given in Appendix C) shows a running example of edge signatures. Considering an edge (hasName,"Abraham Lincoln"), we first map the edge label "hasName" into a bitstring of length 12, and then map the vertex label "Abraham Lincoln" into a bitstring of length 16.

DEFINITION 4.2. *Given a class or entity vertex v in the RDF graph, the* vertex signature *$vSig(v)$ is formed by performing bitwise OR operations over all its adjacent edge signatures. Formally, $vSig(v)$ is defined as follows:*

$$vSig(v) = eSig(e_1)|.....|eSig(e_n)$$

where $eSig(e_i)$ is the edge signature for edge e_i adjacent to v and "|" is the bitwise OR operation.

Considering vertex 005 in Figure 1(b), there are four adjacent edges. We can encode each adjacent edge by its edge signature, as shown in Figure 10(a) (given in Appendix C). A vertex signature is defined in Definition 4.2. Figure 10(b) shows the signature of vertex 005.

DEFINITION 4.3. *Given an RDF graph G, its corresponding data signature graph G^* is induced by all entity and class vertices in G together with the edges whose endpoints are either entity or class vertices. Each vertex v in G^* has its corresponding vertex signature $vSig(v)$ (defined in Definition 4.2) as its label. Given an edge $\overrightarrow{v_1v_2}$ in G^*, its edge label is also a signature, denoted as $Sig(\overrightarrow{v_1v_2})$, to denote the property between v_1 and v_2.*

Note that we adopt the same hash function in Definition 4.1 to define $Sig(\overrightarrow{v_1v_2})$. Specifically, we set m out of M bits in $Sig(\overrightarrow{v_1v_2})$ to be '1' by some string hash functions. Figure 3 shows an example of data signature graph G^*.

Actually, we can also encode the query graph Q by an analogous method. Specifically, considering an entity or class vertex v in Q, for each adjacent edge pair $e(eLabel, nLabel)$ of v in Q, we encode e into a bitstring $eSig(e)$ according to Definition 4.1. Note that, if the adjacent neighbor vertex of v is a parameter vertex, we set $eSig(e).n$ to be a signature with all zeros; if the adjacent neighbor vertex of v is a wildcard vertex, we only consider the substring without "wildcard" in the label. For example, in Figure 2(a), we can only encode substrings "02-12" and "04-15" for the wildcard vertices "*02-12*" and "*04-15*", respectively. The vertex signature $vSig(v)$ can be obtained by performing bitwise OR operations over all adjacent edge signatures.

Given a query graph Q, we can obtain a query signature graph Q^* induced by all entity and class vertices in Q together with all edges whose endpoints are also entity or class vertices. Each vertex v in Q^* is a vertex signature $vSig(v)$, and each edge $\overrightarrow{v_1v_2}$ in Q^* is associated with an edge signature $Sig(\overrightarrow{v_1v_2})$. Figure 3 shows Q^* that corresponds to query Q_3 in Figure 2(b).

DEFINITION 4.4. *Consider a data signature graph G^* and a query signature graph Q^* that has n vertices $\{v_1, ..., v_n\}$. A set of n distinct vertices $\{u_1, ..., u_n\}$ in G^* is said to be a* match *of Q^*, if and only if the following conditions hold:*

1. *$vSig(v_i)\&vSig(u_i) = vSig(v_i)$, $i = 1, ..., n$, where '&' is the bitwise AND operator.*
2. *If there is an edge from v_i to v_j in Q^*, there is also an edge from u_i to u_j in G^*.*

Note that, each vertex u (and v) in data (and query) signature graph G^* (and Q^*) has one vertex signature $vSig(v)$. For the simplicity of symbols, we use u (and v) to denote $vSig(u)$ in G^* (and $vSig(v)$ in Q^*) when the context is clear.

Given an RDF graph G and a query graph Q, their corresponding signature graphs are G^* and Q^*, respectively. The matches of

Q over G are denoted as RS, and the matches of Q^* over G^* are denoted as CL.

THEOREM 4.1. *$RS \subseteq CL$ holds.*

5. INDEXING STRUCTURE AND QUERY ALGORITHM

The key problem to be addressed is how to find matches of Q^* (query signature graph) over G^* (data signature graph) efficiently. A straightforward method can work as follows: first, for each vertex $v_i \in V(Q^*)$, we find a list $R_i = \{u_{i_1}, u_{i_2}, ..., u_{i_n}\}$, where $v_i \& u_{i_j} = v_i$, $u_{i_j} \in V(G^*)$, and $u_{i_j} \in R_i$. Then, we perform a multi-way join over these lists R_i to find matches of Q^* over G^* (finding CL). Actually, the first step (finding R_i) is a classical inclusion query [7].

Given a set of objects with set-valued attributes, an inclusion (or subset) query searches for all objects containing certain attribute values [20]. Usually, signatures are used to indicate the presence of individuals in sets. Therefore, we can represent a set of objects with set-valued attributes as a set of signatures $\{s_i\}$ and an inclusion query as a query signature q. An inclusion (or subset) query is to find all signature s_i, where $q \& s_i = q$. In order to reduce the search space, S-tree [7], a height-balanced tree, is proposed to organize all signatures $\{s_i\}$. Each intermediate node is formed by superimposing all child signatures in S-tree. Therefore, we can employ a S-tree [7] to support the first step efficiently, i.e., finding R_i. An example of S-tree is given in Figure 11 of Appendix C.

However, S-tree cannot support the second step (i.e. a multi-way join), which is NP-hard as discussed earlier. Although many subgraph matching methods have been proposed (e.g., [17, 24]), they are not scalable to very large graphs. Therefore, we propose new index structures for a large data signature graph G^*.

5.1 Indexing Structures–A Simple Version

In this subsection, we propose a simple method to build a VS-tree (*v*ertex *s*ignature *tree*). Although it is not optimized for query performance, it illustrates the main idea of our methods.

Given a data signature graph G^*, we first build a S-tree over all vertex signatures in G^* (i.e.,$V(G^*)$). S-tree is a classical height balanced tree that can support inclusion queries efficiently. Given a query signature q and a set of data signatures $\{s_i\}$, an *inclusion query* is to find all data signatures s_i, where $q \& s_i = q$. In our problem, each leaf entry of the S-tree is a vertex signature in G^*. Interested readers can refer to [7] for details of the S-tree.

As mentioned earlier, S-tree cannot support the second step (i.e., multi-way join processing) efficiently. The proposed VS-tree supports the second step for finding matches of Q^* over G^*. The intuition behind VS-tree is as follows: Based on a S-tree, we can build *a multi-resolution summary graph*, which can be used to reduce the search space of subgraph query processing (as discussed in Theorem 5.1). We adopt a bottom-up strategy to build a VS-tree.

First, a S-tree is built over all vertex signatures in G^*, namely, each leaf entry of S-tree corresponds to one vertex signature in G^*. Then, we link these leaf entries according to G^*'s structure. Specifically, given two leaf entries d_1 and d_2 in a S-tree, we introduce an edge between them, if and only if there is an edge between u_1 and u_2 in G^*, where d_1 (d_2) corresponds to u_1 (u_2) in G^*. We also introduce an edge signature $Sig(\overrightarrow{v_1 v_2})$ (Definition 4.3) as the edge label of $\overrightarrow{d_1 d_2}$ in a VS-tree. A running example is given in Figure 5.

Second, given two leaf nodes d_1^l and d_2^l in the S-tree, we introduce a super edge from d_1^l to d_2^l, if and only if there is at least one edge from d_1's children (i.e., leaf entries) to d_2's children. Specifically, if there are n ($n > 1$) edges from d_1^l's children to d_2^l's children in the VS-tree, we introduce a super edge from d_1^l to d_2^l. Furthermore, we assign an edge label for the edge $\overrightarrow{d_1^l d_2^l}$ by performing bitwise "OR" over these n edge labels from d_1's children to d_2's children. Figure 12 (in Appendix C) illustrates the process. Note that, we can also introduce a self-edge for a leaf node d_1^l, if there is at least one edge from one child of d_1^l to another child of d_1^l. The above process is iterated until the root of the VS-tree is reached.

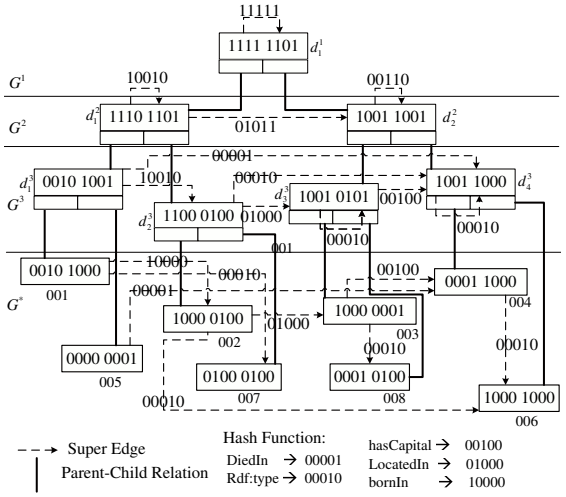


Figure 5: VS-tree

Figure 5 shows a running example of the VS-tree over G^* in Figure 3. Note that, we use d_i^l to denote one node in the I-th level of the VS-tree, which corresponds to the same node in the S-tree (Figure 11). We use $d_i^l.Sig$ to denote the signature associated with node d_i^l. For simplicity, we use d_i^l to denote $d_i^l.Sig$ when the context is clear. The I-th level of the VS-tree is a *summary graph*, denoted as G^I, which is formed by all nodes at the I-th level together with all edges between them in the VS-tree.

DEFINITION 5.1. *Consider a query signature graph Q^* with n vertices v_i (i=1,...,n) and a summary graph G^I in the I-th level of VS-tree. A set of nodes $\{d_i^l\}$ ($i = 1, ..., n$) at G^I is called a* summary match *of Q^* over G^I, if and only if the following conditions hold:*

1. *$vSig(v_i) \& d_i^l.Sig = vSig(v_i)$, $i = 1, ..., n$;*
2. *For any edge $\overrightarrow{v_1 v_2}$ in Q^*, there must exist a super edge $\overrightarrow{d_1^l d_2^l}$ in G^I and $Sig(\overrightarrow{v_1 v_2}) \& Sig(\overrightarrow{d_1^l d_2^l}) = Sig(\overrightarrow{v_1 v_2})$.*

Note that, a summary match is not an injective function from $\{v_i\}$ to $\{d_i^l\}$, namely, d_i^l can be identical to d_j^l ($i \neq j$). For example, given a query signature graph Q^* (in Figure 3) and a summary graph G^3 of VS-tree (in Figure 5), we can find one summary match $\{(d_1^3, d_2^3)\}$. An interesting finding is that summary matches can be used to reduce the search space for subgraph search over G^*.

5.2 Query Algorithm–A Simple Version

In this section, we discuss how to find matches of Q^* over G^* using a VS-tree. We employ a top-down search strategy over the VS-tree to find matches of Q^* over G^*. According to Theorem 5.1, the search space at the lower level of the VS-tree is bounded by the summary matches over the upper level. Consequently, we can reduce the total search space.

THEOREM 5.1. *Given a query signature graph Q^*, a data signature graph G^* and VS-tree built over G^*:*

1) Given a summary graph G^I in VS-tree, if there exists no summary match over G^I, there must exist no match of Q^* over G^*.

2) Assume that n vertices $\{u_1, ..., u_n\}$ forms a match (Definition 4.4) of Q^* over G^*. Given a summary graph G^I in VS-tree, u_i's ancestor in G^I is node d_i^I, $i = 1, ..., n$. $\{d_1^I, ..., d_n^I\}$ must form a summary match (Definition 5.1) of Q^* over G^I.

We first illustrate the query algorithm (*VS-query*) using a running example Q_3^* (in Figure 3). Figure 6 shows the query process. First, we find summary matches of Q_3^* over G^1 in VS*-tree, which are $\{(d_1^1, d_1^1)\}$. Then, we push the summary matches into queue H. We always pop one summary match from H and expand it to its *child states* (defined in Definition 5.2). Given a summary match (d_1^1, d_1^1), its child states are formed by $d_1^1.children \times d_1^1.children = \{d_1^2, d_2^2\} \times \{d_1^2, d_2^2\} = \{(d_1^2, d_1^2), (d_1^2, d_2^2), (d_2^2, d_1^2), (d_2^2, d_2^2)\}$. For each child state, we check whether it is a summary match of Q^*. If so, we call it a *valid child state*. We push all valid child states into queue H. In this example, only $\{(d_1^2, d_1^2)$ is summary match of Q^*, i.e., a valid child state. Thus, we put it into H. Iteratively, we pop some summary match and expand it to its child states in each step. The above process is iterated until reaching the leaf entries (i.e., vertices in G^*) of VS-tree. Finally, we can find matches of Q^* over leaf entries of VS-tree, namely, the matches of Q^* over G^*. The pseudo code are given in Algorithm 1 in Appendix B.

DEFINITION 5.2. Child State. *Given a query signature graph Q^* with n vertices v_i ($i = 1, ..., n$), n nodes $\{d_1^I, ..., d_n^I\}$ in VS-tree forms a summary match of Q^*, n nodes $\{d_1^{I'}, ..., d_n^{I'}\}$ is a child state of $\{d_1^I, ..., d_n^I\}$, if and only if $d_i^{I'}$ is a child node of d_i^I, $i = 1, .., n$. Furthermore, if $\{d_1^{I'}, ..., d_n^{I'}\}$ is also a summary match of Q^*, $\{d_1^{I'}, ..., d_n^{I'}\}$ is called a* valid child state *of $\{d_1^I, ..., d_n^I\}$.*

THEOREM 5.2. *Given a query signature graph Q^* and a data signature graph G^*, VS-query algorithm (Algorithm 1) can find all matches of Q^* without any false positive and negative.*

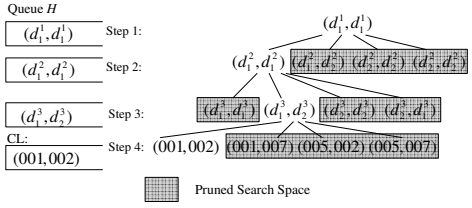


Figure 6: Algorithm Process

6. OPTIMIZED METHODS

For illustration purposes, we presented a conceptually simple strategy, including both index structure (VS-tree) and query algorithm (VS-Query), in Section 5. We discuss optimizations to the method in this section. First, let us discuss three limitations of VS-tree and VS-query algorithm as presented in Section 5. Then, the corresponding optimized methods will be presented.

As discussed in Section 5.2, we employ Theorem 5.1 to reduce the search space in VS-query. It is straightforward to conclude that the performance of VS-query depends on the number of summary matches of query Q^*. A negative finding of VS-tree is that high level summary graphs G^I have much larger densities ($\alpha = |E(G^I)|/|V(G^I)|$) than that in G^*. Consequently, there may exist a large number of summary matches over G^I, which leads to low pruning power on some high levels of the VS-tree. One obvious way to improve the performance is to reduce the number of super edges in each G^I, i.e. the summary graph over each level in the VS-tree. As we note, VS-tree is based on S-tree, whose operations, such as, node insertion, split, deletion, and merge, are optimized for inclusion queries, not for reducing the number of super edges in G^I.

For example, we always insert a vertex signature v (in G^*) into one node d, where v and d has the minimal Hamming distance [7], which is a popular method to measure the similarity between two bitstrings, i.e., signatures. For example, we insert a vertex u_5 and its adjacent edges into G^*, as shown in Figure 7(a). Figure 7(b) shows the VS*-tree T_1 that corresponds to the original G^*. When we insert u_5 into T_1, according to the Hamming distance, we insert u_5 into d_1^2 as its child entry, since $\delta(u_5, d_1^2) = 2 < \delta(u_5, d_2^2) = 5$, where $\delta(u_5, d_1^2)$ denotes that Hamming distance between u_5 and d_1^2. According to the method in Section 5.1, we need to introduce an extra super edge $\overrightarrow{d_1^2 d_2^2}$ in G^2, as shown in the updated VS*-tree T_2 in Figure 7(c). Figure 7(d) shows another way of inserting u_5 into the same VS-tree T_1, which introduces no new super edge into the updated VS-tree T_3. Given the same query signature graph Q^*, there are two summary matches over G^2 of T_2, but only one summary match over $G^2$of T_3. This example motivates us to optimize the operations over the VS-tree.

Another limitation of VS-query is that the multi-way join processing always begins from the root of the VS-tree. Actually, some high level summary graphs may provide little pruning power as mentioned above. In order to optimize query performance, an "oracle" algorithm should "magically" know which level of VS-tree to begin with to reduce the number of summary matches. Therefore, a cost model will be proposed to guide our query algorithm.

Finally, let us recall Lines 5-9 of Algorithm 1. Given a summary match $J = \{d_1^I, ..., d_n^I\}$, we first find children of d_i^I, $i = 1, ..., n$. Then, we find valid child states of J. Specifically, we materialize all child states of J and check whether each one is a summary match (or match) of Q^*. Essentially, finding valid child states of J is to perform multi-way join over $d_i^I.children$, $i = 1, ..., n$. Obviously, the above brute-force enumeration is too expensive. Instead, we can employ a DFS (depth-first search) strategy to find valid child states.

Due to space limitation, in the body of the paper, we only address the first issue regarding index construction in this section. The optimization methods for VS-query algorithm (the last two problems mentioned above) will be discussed in Appendix B, where we also propose an optimized query algorithm called VS*-query.

6.1 Indexing Structure-An Optimized Method

In this section, we propose a new way to build the index structure, called VS*-tree, which has the analogue structure with VS-tree. However, the operations over VS*-tree, such as insertion, deletion and split, are optimized for subgraph query. Given a data signature graph G^*, we build the corresponding VS*-tree over G^* by inserting the vertices of G^* sequentially.

6.1.1 Insertion

Given a vertex u (in G^*) to be inserted, an insertion operation begins at the root of VS*-tree and iteratively chooses a child node until it reaches a leaf node. After inserting v in a suitable leaf node d, the signature of that leaf node must be updated. Furthermore, the summary graph at the leaf level of VS*-tree is also updated. Specifically, if u has an edge (in G^*) adjacent to its other endpoint in another leaf node, we need to introduce a super edge to d, or update the edge signature associated with the super edge. If the leaf signature and leaf summary graph have changed, the change must be propagated upwards within the VS*-tree. The main challenge of insertion is the criterion for choosing a child node. The criterion in the VS-tree only depends on the Hamming distance between the

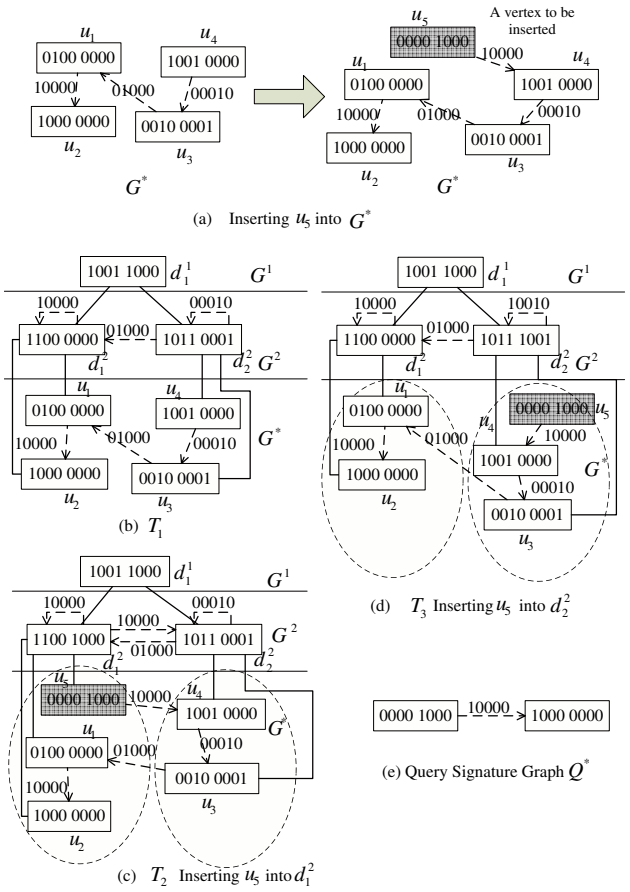


Figure 7: Motivation of Building VS*-tree

signatures of u and the node in VS-tree. Now, the criterion in VS*-tree depends on both node signatures and G^*'s structure.

Given a vertex u and a non-leaf node d, d has n children $d_1,...,d_n$. The distance between u and d_i ($i = 1, ..., n$) is formally defined as follows:

$$Dist(u, d_i) = \frac{\delta(u, d_i)}{|u|} \times \frac{\beta(u, d_i)}{Max_{j=1}^{n}(\beta(u, d_j))} \quad (1)$$

where $\delta(u, d_i)$ is the Hamming distance between u and d_i and $|u|$ is the length of the vertex signature (bitstring), and $\beta(u, d_i)$ is the number of newly introduced super edges adjacent to d_i, if u chooses node d_i.

As mentioned earlier, after inserting vertex u into a suitable leaf node, the signature of that leaf node and super edges adjacent to it may be updated, the change must be propagated upwards within the VS*-tree. Note that, we can update the super edges adjacent to that leaf node, according to the adjacent edges to u in G^*.

6.1.2 Split

Like other height balanced trees, insertion into a node that is already full will invoke node split. Specifically, the B+1 entities of the node will be partitioned into two new nodes, where B is the maximal fanout for a node in VS*-tree. We illustrate our strategy as follows: First, we find two entities that have the maximal Hamming distance between them as two *seed* nodes. Second, we associate each left entry with the nearest seed node, according to Equation 1. Note that, after node splitting, we have to update the signatures and the super edges associated with the two new nodes. The updates are very straightforward. Node splitting invokes insertions over the upper level of VS*-tree, which also leads to the splitting that may be propagated to the root of the VS*-tree.

6.1.3 Deletion

To delete a vertex u from VS*-tree, we find the leaf node d where u is stored, and delete u. After deleting u, the nodes along the path from the root down to d will be affected. We adopt the bottom-up strategy to update the signature of and super edges associated with the nodes. After deletion, if some node d has less than b entries, where b is the minimal fanout of node in VS*-tree, then d is deleted and its entries are reinserted into VS*-tree.

7. MAINTENANCE IN gStore

In gStore, the updates over the adjacency list table (Figure 4) are straightforward. The key challenge is the maintenance of VS*-tree to support updates over RDF datasets. We have discussed the maintenance of VS*-tree in Section 6. Further details about the maintenance of gStore are given in Appendix E.

8. EXPERIMENTS

In this section, we evaluate our method over two real large RDF datasets, and compare it with SW-store [1], RDF-3x [12], and x-RDF-3x [15]. We also compare our method with one commercial system BigOWLIM [1] and graph-based solution GRIN [21].

8.1 Datasets & Setup

We use two large real datasets in our experiments: 1) Yago (http://www.mpi-inf.mpg.de/yago-naga/yago/) extracts facts from Wikipedia and integrates them with the WordNet thesaurus. It contains about 20 million RDF triples and consumes 3.1GB; 2) DBLP (http://sw.de ri.org/ aharth/2004/07/dblp/) contains a large number of bibliographic descriptions. There are about 8 million triples consuming 0.8GB. Our algorithm is implemented using standard C++. The experiments are conducted on a P4 3.0GHz machine with 2G RAM running Ubuntu Linux. We test our method and all competitors over both exact and wildcard queries. Since none of the competitors, except for BigOWLIM, can support wildcard queries, in order to enable comparison, we propose the following method: Given a SPARQL query Q with wildcards, for each wildcard vertex, we rewrite it as a parameter vertex. In this way, we can get a SPARQL query Q' without wildcards. Then, we employ RDF-3x, SW-store, x-RDF-3x and GRIN to answer Q'. Finally, for each result of Q', we verify whether it is a result of Q based on the wildcard condition.

For exact query evaluation, we use all SPARQL queries in [12] over the Yago dataset. We also define 6 queries over DBLP dataset. Due to space limitation, we do not list our sample queries in this paper. More details about sample queries can be found in the full version of this work[2]. For wildcard query evaluation, we rewrite all SPARQL queries in [12] into queries with wildcards. Specifically, for each exact SPARQL query Q, we replace each literal vertex in Q as a wildcard vertex. In this way, we can get a query Q'' with wildcards. We also define 6 wildcard queries over DBLP dataset. All sample queries are given in the full version of this work.

8.2 Offline Performance

We compare our method (gStore) with five competitors over both Yago and DBLP datasets. For a fair comparison, we adopt the settings in [12], i.e., each dataset is first converted into a factorized form: one file T with RDF triples represented as integer triples, and one dictionary file M to map from ids to literals. All methods utilize the same input files and load them into their own systems.

[1] http://www.ontotext.com/owlim/big/

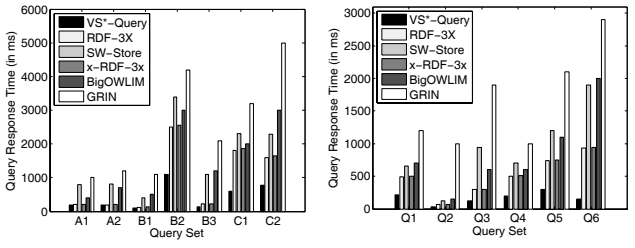
[2] http://www.icst.pku.edu.cn/intro/leizou/RDFQuery.pdf

The loading time is defined as the total offline processing time. The total space cost is defined as the size of the whole database including the corresponding indexes. We show load time and the total space cost in Figures 15(a) and 15(b) (given in Appendix F), respectively. Figure 15(a) and 15(b) show that our method has the shortest loading time and least index sizes. Note that, x-RDF-3x is slower than RDF-3x in indexing building, and they have the same index sizes.

8.3 Online Performance

8.3.1 Exact Queries

We compare the performance of our method (VS*-query) with five competitors over both Yago and DBLP datasets. Figure 8 shows that VS*-query is much faster than other methods. From Figure 8, x-RDF-3x is a little slower than RDF-3x, since x-RDF-3x introduces extra transactional overhead [15].

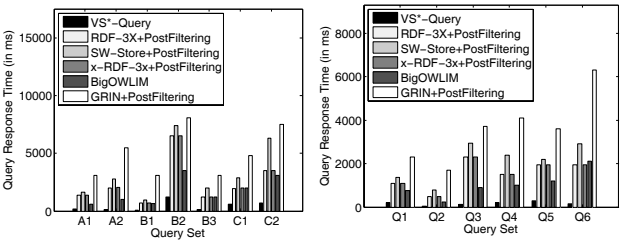


(a) Yago (b) DBLP

Figure 8: Exact Query Response Time

8.3.2 Wildcard Queries

In order to enable comparison over wildcard queries, we adopt the post-filtering method in Section 8.1 in RDF-3x, SW-store, x-RDF-3x and GRIN. Since BigOWLIM has embedded full-text index, it can support wildcard queries. Figure 9 shows query response times of different methods. It is observed that our method has the same query response time as that in exact queries. As mentioned earlier, we generate a wildcard query Q' by replacing each literal vertex into a wildcard vertex. Actually, Q' and Q correspond to the same query signature graphs. VS*-query can answer both exact and wildcard queries in a uniform manner, thus, they have the same query response time. However, the query performance degrades dramatically in other methods, since they cannot support wildcard queries directly, as shown in Figure 9.



(a) Yago (b) DBLP

Figure 9: Wildcard Query Response Time

9. CONCLUSIONS

In this paper, we propose to store and query RDF data from graph database perspective. In order to speed up query processing, we propose two novel indexes, VS-tree and VS*-tree. The most important contribution in this paper is that our method can support both exact and wildcard SPARQL queries in a scalable manner. Furthermore, it can support online updates efficiently.

10. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.

[2] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[3] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. Dbpedia - a crystallization point for the web of data. *J. Web Sem.*, 7(3):154–165, 2009.

[4] V. Bönström, A. Hinze, and H. Schweppe. Storing RDF as a graph. In *LA-WEB*, pages 27–36, 2003.

[5] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, pages 54–68, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[7] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *SIGIR*, pages 77–87, 1986.

[8] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288, 1984.

[9] 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.*, 24(4):28–34, 2001.

[10] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *WWW*, pages 90–101, 2003.

[11] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC*, pages 211–224, 2007.

[12] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[13] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, 2009.

[14] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[15] T. Neumann and G. Weikum. X-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 1(1):256–263, 2010.

[16] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[17] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[18] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604, 2008.

[19] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[20] E. Tousidou, P. Bozanis, and Y. Manolopoulos. Signature-based structures for objects with set-valued attributes. *Inf. Syst.*, 27(2):93–121, 2002.

[21] O. Udrea, A. Pugliese, and V. S. Subrahmanian. Grin: A graph based RDF index. In *AAAI*, pages 1465–1470, 2007.

[22] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[23] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *SWDB*, pages 131–150, 2003.

[24] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[25] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in RDF triple stores. In *ICDE*, pages 1263–1266, 2009.

[26] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.

APPENDIX

A. RELATED WORK

As noted earlier, three kinds of approaches are generally used to store and query RDF data: one giant triple table, clustered property tables and vertically partitioned tables.

1) *One giant triple table*. The methods in this category store all RDF triples in a single three-column table, enabling them manipulate all RDF triples in a uniform manner. However, these require performing lots of self-joins over this table to answer a SPARQL query. Some efforts have been made to address this issue, such as, RDF-3x [12, 13] and Hexastore [22], which build several clustered B^+-trees for all permutations of three columns.

2) *Property tables*. There are two kinds of property tables. The first one is called a *clustered property table*. The properties that tend to occur in the same subjects are grouped into one cluster. Each property cluster is mapped to a property table. The second type is a *property-class table*. The subjects with the same type of property are clustered into one property table.

3) *Vertically partitioned tables*. For each property, this approach builds a single two-column (subject, object) table ordered by subject [1]. The advantage of the ordering is to perform fast merge join during query processing. However, this approach does not scale well as the number of properties increases.

As discussed earlier, although the above methods are designed for the scalability of RDF data, they only support exact SPARQL queries, and fail to support wildcard queries. For example, RDF-3x and SW-store store RDF triples by replacing all literals with ids. In this way, they can only support exact queries.

Furthermore, most of existing methods cannot handle online updates over the underlying RDF repositories efficiently. For example, in clustered property table-based methods (such as Jena and SOR), if there are some updates over properties in RDF triples, we have to re-do property clustering and re-build the property tables. Although RDF-3X uses one giant triple table, it needs to modify six clustered B^+-trees to handle updates, and does Hexastore. In SW-store, it is potentially expensive to insert data since each update requires writing to many columns [1]. In order to address this issue, it uses "overflow table + batching write", meaning online updates are recorded to overflow tables and SW-store periodically scans the overflow tables to materialize the updates. Obviously, this kind of maintenance method cannot work well for online social network systems that require real time access.

The recent work xRDF-3x [15] proposes an efficient online maintenance algorithm, but, it fails to support wildcard SPARQL queries. There exist some works that discuss the possibility of storing RDF data as a graph (e.g., [4, 22]), but these approaches do not address the scalability issues. Some are based on main memory implementations [18], while others utilize graph partitioning to reduce self-joins of triple tables [25]. The key problem with graph partitioning method [25] is that it cannot support updates efficiently. Once the RDF graph is updated, we have to re-partition the graph from scratch. Otherwise, the correctness of results cannot be guaranteed.

B. VS*-QUERY

In Section 5, we propose VS-query algorithm for finding matches of Q^* over G^*, as shown in Algorithm 1. As discussed earlier, there are three limitations of VS-query. Due to space limitation, we only addressed the first problem in Section 6, i.e., VS-tree is not optimized for subgraph search. We propose VS*-tree to optimize subgraph search in Section 6.1.

The second problem of VS-query is that it always begins the multi-way join processing from the root of VS*-tree. Consequently, we may generate a lot of summary matches. Actually, in order to speed up query processing, an oracle algorithm should magically know which level to begin with to reduce the number of summary matches.

Algorithm 1 Query Algorithm Over VS-tree (VS-Query)

Require: **Input**: a query signature graph Q^* and a data signature graph G^* and a VS-tree.
Output: CL: All matches of Q^* over G^*.
1: Set $CL = \phi$
2: Find summary matches of Q^* over G^1, which are pushed into queue H.
3: **while** ($|H| > 0$) **do**
4: Pop one summary match from H, denoted as J.
5: **for** each child state S of J **do**
6: **if** S reaches leaf entries and S is a match of Q^* **then**
7: Insert S into CL
8: **if** S does not reach the leaf nodes and S is a summary match of Q^* **then**
9: Push it into queue H.
10: Report CL.

Finally, given a summary match J, we need to materialize all child states of J and verify each one whether it is a summary match (or match) of Q^* in Algorithm 1, which is quite expensive.

In order to address the above two problems, some optimized methods are proposed in the following subsections.

B.1 Which Level To Begin

As mentioned earlier, VS-query algorithm always begins its multi-way join process from the root of VS-tree, which leads to a large number of intermediate summary matches. In order to optimize query performance, a cost model is needed to guide the level of VS*-tree that the algorithm should begin with. Specifically, we introduce a concept "pruning power" of G^I with regard to Q^* (denoted as $P(Q^*, G^I)$). Then, we propose a simple but effective method to estimate $P(Q^*, G^I)$. Optimized query algorithms should begin its multi-way join processing from G^I that has the maximal pruning power.

DEFINITION B.1. *Given a query signature graph Q^* with n edges e_i, $i = 1, ..., n$ and m vertices v_j, $j = 1, ..., m$, and summary graph G^I at the I-th level of VS*-tree, the* pruning power *of G^I with regard to Q^* is defined as follows:*

$$P(Q^*, G^I) = 1 - \frac{|N(Q^*, G^I)|}{\prod_{j=1}^{j=m} |N(v_j, G^I)|} \quad (2)$$

where $N(Q^*, G^I)$ denotes the set of summary matches of Q^* over G^I, and $N(v_j, G^I)$ denotes the set of nodes d (in G^I) and ($d\&v_j = d$).

Note that, in Equation 2, $\prod_{j=1}^{j=m} (|N(v_j, G^I)|)$ denotes the total search space of Q^* over G^I, and $\prod_{j=1}^{j=m} |N(v_j, G^I)| - |N(Q^*, G^I)|$ denotes the search space that can be pruned. As we know, finding $N(Q^*, G^I)$ has the exponential time complexity. It is inefficient to find $N(Q^*, G^I)$ exactly to compute the pruning power. Therefore, we propose a simple but effective method to estimate $P(Q^*, G^I)$.

DEFINITION B.2. *Given a query signature graph Q^* with n edges e_i, $i = 1, ..., n$ and m vertices v_j, $j = 1, ..., m$, and summary graph G^I at the I-th level of VS*-tree, the* estimated pruning power *of G^I with regard to Q^* is defined as follows:*

$$\widetilde{P}(Q^*, G^I) = \prod_{i=1}^{i=n} P(e_i, G^I)$$

where $P(e_i, G^I)$ is the pruning power *of G^I with regard to edge e_i, $i = 1, ..., n$.*

Let edge e_i=$\overrightarrow{v_{i_1}v_{i_2}}$. $P(e_i, G^I)$ is defined as follows:

$$P(e_i, G^I) = 1 - \frac{|N(e_i, G^I)|}{|N(v_{i_1}, G^I)| * |N(v_{i_2}, G^I)|}$$

where $N(v_{i_1}, G^I)$ denotes the set of nodes d in G^I and (d&$v_{i_1} = d$), and $N(e_i, G^I)$ denotes the set of summary matches of e_i over G^I.

Given an edge $e_i = \overrightarrow{v_{i_1}v_{i_2}}$ (in Q^*) and G^I, it is very fast to find $N(G^I, v_{i_1})$ by invoking inclusion operation over VS*-tree. The main challenge of estimating $P(e_i, G^I)$ arises from computing $|N(e_i, G^I)|$, i.e., the number of summary matches of e_i over G^I.

In order to compute $|N(G^I, e_i)|$, we propose the following method that has the linear time complexity. For each node d_{i_1} in $N(v_{i_1}, G^I)$, we insert d_{i_1}'s adjacent neighbors into $NN(v_{i_1}, G^I)$. Finally, we compute $|N(e_i, G^I)| = |NN(v_{i_1}, G^I) \cap N(v_{i_2}, G^I)|$. Obviously, this method has the linear time complexity, i.e, $O(|N(v_{i_1}, G^I)| + |N(v_{i_2}, G^I)|)$.

At run time, given a query signature graph Q^*, for each level summary graph G^I of VS*-tree, it is effective to compute $\widetilde{P}(Q^*, G^I)$ according to Definition B.2. Then, query algorithm begins its multi-way join processing from some summary graph G^I that has the maximal estimated pruning power, i.e., $\widetilde{P}(Q^*, G^I)$. The pseudo codes of query algorithm will be given in Algorithm 3.

Algorithm 2 Find Valid Child States

Require: Input: a query signature graph Q^* with n vertices v_i, $i = 1, ..., n$, and a summary match of Q^* over G^I, which is denoted as $M(d_1^I, ..., d_n^I)$.
Output: S: all valid states of M with regard to Q^*.

```
1: Set S = φ and Q' = φ.
2: for each node v_i in Q* do
3:     Compute N(v_i, d_i^I.children)
4: Select some vertex v_i, where |N(v_i, d_i^I.children)| is minimal among all vertices in Q*.
5: Q' = Q' ∪ v_i and M(Q') = N(v_i, d_i^I.children).
6: while Q'! = Q* do
7:     for each backward edge e_i = v_{i1}v_{i2} that is adjacent to Q' do
8:         M(Q' ∪ e_i)=BackWard(e_i, M(Q'))
9:         Q' = Q' ∪ e_i
10:    for each forward edge e_i = v_{i1}v_{i2} that is adjacent to Q' do
11:        M(Q' ∪ e_i)=ForWard(e_i, M(Q'))
12:        Q' = Q' ∪ e_i
13: Set S=M(Q') and return S
```

Backward($e_i = \overrightarrow{v_{i_1}v_{i_2}}, M(Q')$)

```
1: for each tuple t in M(Q') do
2:     If t cannot form a summary match of Q' ∪ e_i
3:     Delete t from M(Q')
4: M(Q' ∪ e_i) = M(Q')
5: Return M(Q' ∪ e_i).
```

Forward($e_i = \overrightarrow{v_{i_1}v_{i_2}}, M(Q')$)

```
1: if v_{i1} ∈ Q' ∧ v_{i2} ∉ Q' then
2:     for each tuple t in M(Q') do
3:         for each node d in d_{i2}^I.children do
4:             if t ⋈ d is a summary match of Q' ∪ e_i then
5:                 Insert t ⋈ d into M(Q' ∪ e_i)
6: if v_{i2} ∈ Q' ∧ v_{i1} ∉ Q' then
7:     for each tuple t in M(Q') do
8:         for each node d in d_{i2}^I.children do
9:             if d ⋈ t is a summary match of Q' ∪ e_i then
10:                Insert d ⋈ t into M(Q' ∪ e_i)
11: Return (Q' ∪ e_i)
```

B.2 Finding Valid Child States

Given a query signature graph Q^* with n vertices v_i, $i = 1, ..., n$, a summary match of Q^* over G^I is denoted as $J(d_1^I, ..., d_n^I)$. According to Lines 5-9 of Algorithm 1, we need to materialize all child states of J, i.e. $d_1^I.children \times ... \times d_n^I.children$, and then check each one to determine whether it is a summary match (or match) of Q^*. Assume that the fanout of a node in VS-tree is B. There are B^n child states of J. Obviously, this method is inefficient. Essentially, finding valid child states is to perform multi-way join processing over $d_1^I.children$ ($i = 1, ..., n$).

Instead, we propose a DFS strategy to find all valid child states of J. Initially, we set $Q' = \phi$, which denotes the structure of Q^* that has been visited so far. We start a DFS over G^* beginning from some vertex v_i that $|N(v_i, d_i^I.children)|$ is minimal among all vertices in G^*, where $N(v_i, d_i^I.children)$ denotes all child nodes d of d_i^I and v_i&$d = v_i$. We insert v_i into query Q'. Now, the matches of Q', i.e., $J(Q')$, are updated as $N(v_i, d_i^I.children)$.

DEFINITION B.3. *An edge $e = \overrightarrow{v_1v_2}$ in Q^* is called* adjacent *to Q' if and only if $(e \notin Q') \wedge (v_1 \in Q' \vee v_2 \in Q')$.*

Given an adjacent edge $e = \overrightarrow{v_1v_2}$ to Q', e is called a backward edge *if and only if $(v_1 \in Q' \wedge v_2 \in Q')$. Otherwise, e is called a* forward edge.

For each edge e_i adjacent to Q', if e_i is a backward edge, we employ Backward function in Algorithm 2 to find matches of $Q' \cup e_i$, i.e., $J(Q' \cup e_i)$. Otherwise, we employ Forward function to find $J(Q' \cup e_i)$. Essentially, Forward function is a nested loop join process, but Backward function is a selection process. Therefore, we always process backward edges ahead of forward edges. The whole process is iterated until $Q' = Q^*$. Finally, we report all valid child states of J, i.e., $d_1^I.children \bowtie ... \bowtie d_n^I.children$.

B.3 Putting It All Together–gStore

In this subsection, we recall the whole framework of our method, a graph-based RDF *store*, called *gStore*. Basically, there are two steps in gStore, including one offline and the other online.

In offline processing, we first represent an RDF dataset by an RDF graph G and store it as a disk-based adjacency list table T. According to the encoding method in Section 4, we encode G into a data signature graph G^*. Finally, we build a VS*-tree over G^* by invoking insertion operation (discussed in Section 6.1) sequentially. At the end of the offline process, there are two data structures: a disk-based adjacency list table T and a VS*-tree over G^*.

At run time, we represent a SPARQL query by a query graph Q, and encode it into a query signature graph Q^*. Then, we employ the optimized query algorithm over VS*-tree (i.e, Algorithm 3 that will be discussed shortly) to find matches of Q^* over G^*, denote as CL. For each match in CL, we check whether it is a match of Q over G. Finally, all matches of Q over G (denoted as RS) are returned to users.

Algorithm 1 has presented a simple method to find CL. However, this method is not optimized. We discuss an optimized algorithm (Algorithm 3) that combines some optimizations in Sections B.1 and B.2.

Specifically, given a query signature graph Q^*, we first employ the method in Section B.1 to find the I-th level (of VS*-tree) that has the maximal estimated pruning power. Then, for each vertex v_i in Q^*, we employ inclusion queries of S-tree down to G^I, i.e, the I-th level of VS*-tree. The matching nodes of v_i in G^I are denoted as $M(v_i, G^I)$. We employ Algorithm 2 to find summary matches of Q^* over G^I, i.e., $M(v_1, G^I) \bowtie ... \bowtie M(v_n, G^I)$, which are pushed into queue H. In each subsequent step, we always pop one summary match M from H. Then, we find valid child states of M by invoking Algorithm 2. If valid child states have reached leaf entries of VS*-tree, we insert them into CL. Otherwise, they are pushed back to H. The whole process is iterated until $H = \phi$. Finally, we report CL.

Algorithm 3 Optimized Query Algorithm Over VS*-tree, VS*-query

Require: Input: a query signature graph Q^* with n vertices v_i, $i = 1, ..., n$, and a VS*-tree.
Output: CL: all matches of Q^* over G^*.

1: Employ the method in Section B.1 to find the I-th level (of VS*-tree) that has the maximal estimated pruning power with regard to Q^*.
2: **for** each vertex v_i in Q^* **do**
3: Employ the inclusion method of S-tree over VS*-tree down to G^I to find matching nodes of v_i in G^I, denoted as $M(v_i, G^I)$.
4: Find all summary matches of Q^* over G^I by calling Algorithm 2 from $M(v_1, G^I) \bowtie ..... \bowtie M(v_n, G^I)$, which are pushed into queue H.
5: **while** $H \neq \phi$ **do**
6: Pop one summary match from H, denoted as J.
7: Find all valid child states of J by calling Algorithm 2.
8: **if** these valid child states reaches leaf entries **then**
9: Insert them into CL
10: **if** these valid child states do not reach the leaf entries **then**
11: Push them into queue H
12: Report CL

C. ENCODING AND S-TREE

Figure 10 shows how to assign a vertex signature to a vertex in a RDF graph. A sample of S-tree is given in Figure 11. In order to build VS-tree, we need to introduce super edges between intermediate nodes. Figure 12 shows how to obtain super edge signatures.

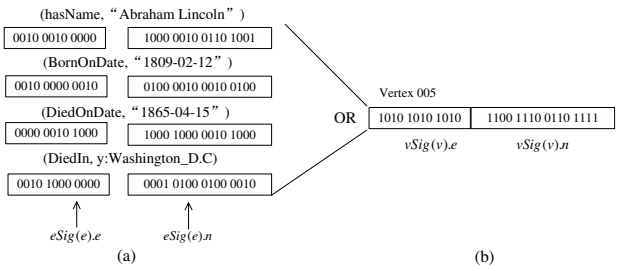


Figure 10: The Encoding Technique

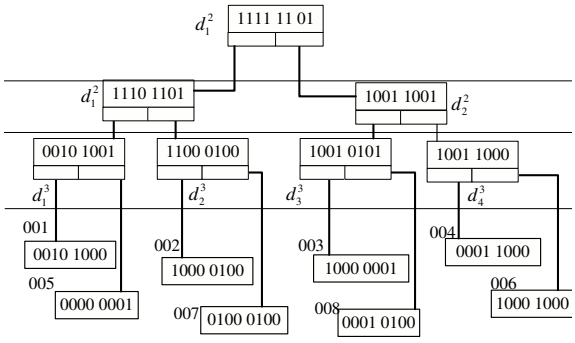


Figure 11: S-tree

D. PARAMETER SETTING

As discussed earlier, we introduce some parameters in our coding methods and indexing structures. In this subsection, we discuss how to set up these parameters to optimize query processing.

D.1 M and m

Given a vertex u in the RDF graph, we encode each edge label ($eLabel$) adjacent to u into a bitstring $eSig(e).e$ with length M, and set m out of M bits to be '1'. We obtain $vSig(u).e$ by performing bitwise OR over all $eSig(e).e$. Analogous to signature files, there

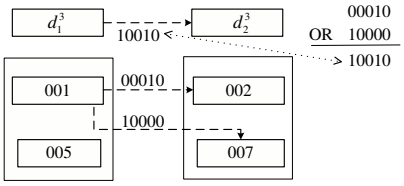


Figure 12: Building Super Edges

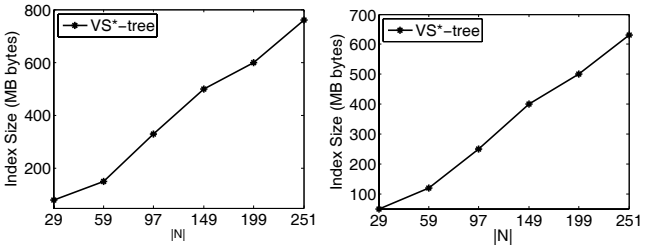


Figure 13: VS*-tree Index Size VS. N

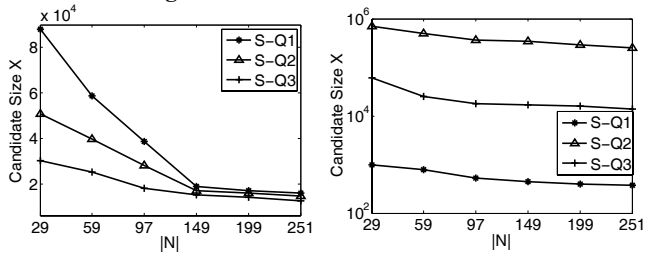


Figure 14: Candidate Size X VS. $|N|$

may exist the "false drop" problem [8]. For example, given a vertex v in query graph Q, all edge labels adjacent to v are denoted as $AdjEdges(v, Q)$. We also use $AdjEdges(u, G)$ to denote all edge labels adjacent to u in G. If $AdjEdges(v, Q) \not\subset AdjEdges(u, G) \wedge v\&u = v$, we say that a *false drop* has occurred. $v\&u = v$ means that u is a candidate match v. However, $AdjEdges(v, Q) \not\subset AdjEdges(u, G)$ means that u cannot match to v. Obviously, the key issue is how to reduce the number of false drops.

According to a theoretical study [8], the probability of false drops can be quantified by the following equation.

$$P_{false\_drop} = (1 - e^{-\frac{|AdjEdges(v,Q)|*m}{M}})^{m*|AdjEdges(u,G)|} \quad (3)$$

where $|AdjEdges(v, Q)|$ is v's degree in Q, $|AdjEdges\ (u, G)|$ is u's degree in G, M is the length of bitstring, and m out of M bits are set to be '1' in hash functions.

Given an RDF graph and query logs, it is straightforward to estimate the average values for $|AdjEdges(v, Q)|$ and $|AdjEdges(u, G)|$. When P_{false_drop} is fixed, we can employ Equation 3 to set up m and M. In Yago, the average value for $|Adj\ Edges(u, G)|$ is 10, and the average value for $|AdjEdges(q, Q)|$ is 3, while $|AdjEdges\ (u, G)| = 20$ and $|AdjEdges(q, Q)| = 3$ in DBLP. We set up $m = 2$ and $M = 97$ in both Yago and DBLP. In this case, according to Equation 3, $P_{false\_drop} < 1.0 \times 10^{-10}$.

D.2 N and n

Actually, we have the same false drop problems in comparing $vSig(q).n$ with $vSig(v).n$. Different from setting m and M, it is quite difficult to quantify the probability of false drops when comparing $vSig(q).n$ and $vSig(v).n$. Therefore, we adopt the following method, using the "n-gram" technique. It has been experimentally determined that n=3 works well [10].

It is clear that the larger N is, the fewer conflicts exist among the vertex signatures. On the other hand, large N will lead to large space cost of vertex signatures. Thus, we need to find a good trade-off for N. We use three star queries to evaluate the pruning power of the encoding technique. Given a star query S, we encode its central vertex v into a vertex signature $vSig(v)$. We use X to denote the number of vertex signatures $vSig(u)$, where $vSig(v)\&vSig(u) = vSig(v)$ in G^*. Obviously, X decreases with the increase of N as shown in Figure 14. However, the decreasing trend slows down when $N > 149$ in Yago and $N > 97$ in DBLP.

On the other hand, larger N leads to larger space cost of VS*-tree. In our experiment, in order to avoid I/O cost of VS*-tree, we require that the whole VS*-tree can be cached in memory. Figure 13 shows the size of VS*-tree with varying N. In our experiment, the maximal available memory size assigned to VS*-tree is 500 M bytes.

According to the observations in Figures 13 and 14, we can set $N = 149$ in Yago and $N = 97$ in DBLP.

E. MAINTENANCE

As mentioned earlier, most existing RDF stores cannot support update effectively. In this section, we discuss the maintenance issues in gStore. Obviously, the updates over the adjacency list table are very straightforward. The main challenge is the maintenance of G^* and VS*-tree to support updates over RDF dataset.

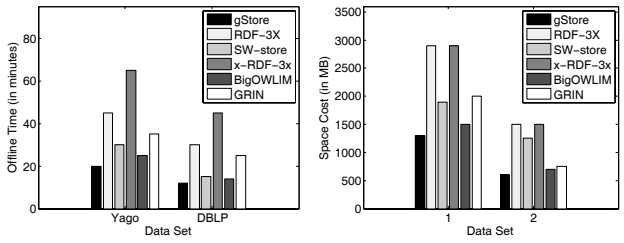
E.1 Insertion

Assume that a new triple $\langle s, p, o\rangle$ is inserted into RDF dataset. s must be an entity or a class vertex in RDF graph G. Thus, s must correspond to one vertex in G^*. If s has existed in G^* before insertion, we delete vertices s and its all adjacent edges from G^*. We also employ the deletion method (discussed in Section 6.1.3) to delete s from VS*-tree. Then, we re-encode vertex s, and re-insert s and its adjacent edges into G^*. Furthermore, we employ insertion method (discussed in Section 6.1.1) of VS*-tree to insert s into VS*-tree. If o is also an entity or a class vertex, we have the analogous method.

E.2 Deletion

Assume that a new triple $\langle s, p, o\rangle$ is deleted from the RDF dataset. s must be an entity or a class vertex in RDF graph G. Thus, s must correspond to one vertex in G^*. We first delete s and all its adjacent edges from G^*, and employ the deletion method (discussed in Section 6.1.3) to delete s from VS*-tree. If after deleting $\langle s, p, o\rangle$, there is no edge adjacent to s in the RDF graph, we can stop here. Otherwise, we re-code s, and insert s into G^* and VS*-tree. If o is also an entity or a class vertex, we have the analogous method.

F. MORE EXPERIMENT RESULTS

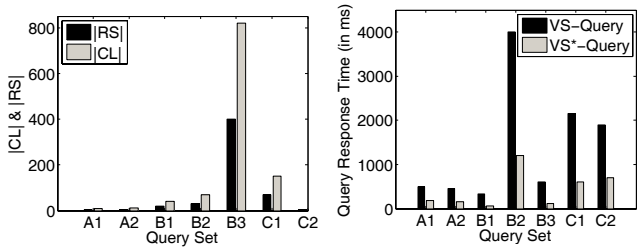


(a) Offline Processing Time (b) DB Size

Figure 15: Evaluating Offline Performance

F.1 Effect of Pruning Power

Theorem 4.1 shows that CL (matches of Q^* over G^*) is a subset of RS (matches of Q over G). Figure 16(a) shows both $|CL|$ and



(a) $|RS|$ VS. $|CL|$ (b) Query Response Time

Figure 16: Yago Dataset

$|RS|$ of queries over Yago dataset. We can find that $|CL| < 3 \times |RS|$, which indicates the low cost of the verification process in our method.

F.2 VS-query Versus VS*-query

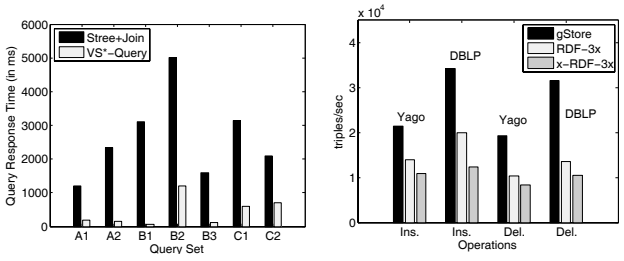
We compare VS-query with VS*-query in both Yago and DBLP datasets. Figure 16(b) shows that VS*-query is much faster than VS-query. The reason behind that is the following: For each summary match, we always need to materialize all child states in VS-query, which is quite expensive. Furthermore, VS*-query can choose the level (in VS*-tree) that leads to the minimal number of summary matches.

F.3 S-tree+Join Versus VS*-query

As mentioned in Section 5, in order to find matches of Q^* over G^*, a straightforward method can work as follows: for each vertex v_i in Q^*, we can employ S-tree to find $R_i = \{u_{i_1}, ..., u_{i_n}\}$, where $u_{i_j}\&v_i = v_i$ and $u_{i_j} \in G^*$. Then, according to the structure of Q^*, we join these lists R_i to find matches of Q^* over G^*. A key problem is that $|R_i|$ may be very large. Consequently, it is quite expensive to join R_i. According to our experiments in Yago, $|R_i| > 1000$ in many queries. Different from S-tree+Join method, most false positives are filtered out at the higher levels in VS*-tree. Therefore, VS*-query is much faster than S-tree+Join method, as shown in Figure 17(a).

F.4 Online Updates

RDF-3X had been extended for updates by deferred-indexing approach [14]. The updates are first recorded into differential indexes. Periodically, differential indexes are merged into main indexes. x-RDF-3x employs the similar update strategy except for introducing "timestamp" of each triple [15]. Note that, the current available codes of RDF-3x and x-RDF-3x do not provide update capabilities. Therefore, we implement the update methods according to [14] and [15], respectively. Figure 17(b) shows that our method is much faster than RDF-3x and x-RDF-3x. Furthermore, RDF-3x is faster than x-RDF-3x, since x-RDF-3x pays more overhead for introducing timestamps.



(a) Evaluating Stree+Join Method Over Yago (b) Evaluating Online Updates

Figure 17: More Experiments