Noname manuscript No.
(will be inserted by the editor)

gStore: A Graph-based SPARQL Query Engine

Lei Zou · M. Tamer Özsu · Lei Chen · Xuchuan Shen · Ruizhe Huang · Dongyan Zhao

the date of receipt and acceptance should be inserted later

Abstract We address efficient processing of SPARQL queries over RDF datasets. The proposed techniques, incorporated into the gStore system, handle, in a uniform and scalable manner, SPARQL queries with wildcards and aggregate operators over dynamic RDF datasets. Our approach is graph-based. We store RDF data as a large graph, and also represent a SPARQL query as a query graph. Thus the query answering problem is converted into a subgraph matching problem. To achieve efficient and scalable query processing, we develop an index, together with effective pruning rules and efficient search algorithms. We propose techniques that use this infrastructure to answer aggregation queries. We also propose an effective maintenance algorithm to handle online updates over RDF repositories. Extensive experiments confirm the efficiency and effectiveness of our solutions.

Extended version of paper "gStore: Answering SPARQL Queries via Subgraph Matching" that was presented at 2011 VLDB Conference.

Lei Zou, Xuchuan Shen, Ruizhe Huang, Dongyan Zhao
Institute of Computer Science and Technology,
Peking University, Beijing, China
Tel.: +86-10-82529643
E-mail: {zoulei,shenxuchuan,huangruizhe,zhaody}@pku.edu.cn

M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada
Tel.: +1-519-888-4043
E-mail: Tamer.Ozsu@uwaterloo.ca

Lei Chen
Department of Computer Science and Engineering,
Hong Kong University of Science and Technology,
Hong Kong, China
Tel.: +852-23586980
E-mail: leichen@cse.ust.hk

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. Its use in various applications is increasing.

A RDF data set is a collection of (subject, property, object) triples denoted as $\langle s, p, o \rangle$. A running example is given in Figure 1a. In order to query RDF repositories, SPARQL query language [23] has been proposed by W3C. An example query that retrieves the names of individuals who were born on February 12, 1809 and who died on April 15, 1865 can be specified by the following SPARQL query (Q_1):

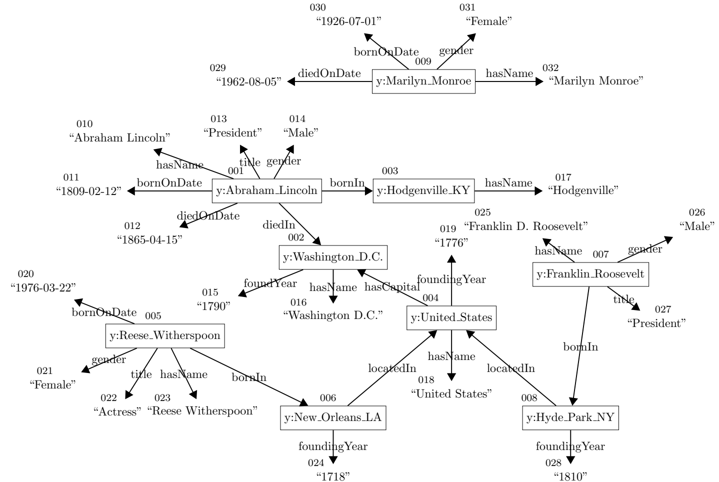
```
SELECT ?name WHERE
{?m <hasName> ?name.
 ?m <bornOnDate> ''1809-02-12''.
 ?m <diedOnDate> ''1865-04-15''.}
```

Although RDF data management has been studied over the past decade, most early solutions do not scale to large RDF repositories and cannot answer complex queries efficiently. For example, early systems such as Jena [31], Yars2 [14] and Sesame 2.0 [6], do not work well over large RDF datasets. More recent works (e.g., [2,20,33]) as well as systems, such as RDF-3x [19], x-RDF-3x [22], Hexastore [30] and SW-store [1], are designed to address scalability over large data sets. However, none of these address scalability **along with** the following real requirements of RDF applications:

– *SPARQL queries with wildcards.* Similar to SQL and XPath counterparts, the wildcard SPARQL queries enable users to specify more flexible query criteria in real-life applications where users may not have full knowledge about a query object. For example, we may know that a person was born in 1976 in a city

Subject	Property	Object
y:Abraham_Lincoln	hasName	"Abraham Lincoln"
y:Abraham_Lincoln	bornOnDate	"1809-02-12"
y:Abraham_Lincoln	diedOnDate	"1865-04-15"
y:Abraham_Lincoln	bornIn	y:Hodgenville_KY
y:Abraham_Lincoln	diedIn	y:Washington_DC
y:Abraham_Lincoln	title	"President"
y:Abraham_Lincoln	gender	"Male"
y:Washington_DC	hasName	"Washington D.C."
y:Washington_DC	foundingYear	"1790"
y:Hodgenville_KY	hasName	"Hodgenville"
y:United_States	hasName	"United States"
y:United_States	hasCapital	y:Washington_DC
y:United_States	foundingYear	"1776"
y:Reese_Witherspoon	bornOnDate	"1976-03-22"
y:Reese_Witherspoon	bornIn	y:New_Orleans_LA
y:Reese_Witherspoon	hasName	"Reese Witherspoon"
y:Reese_Witherspoon	gender	"Female"
y:Reese_Witherspoon	title	"Actress"
y:New_Orleans_LA	foundingYear	"1718"
y:New_Orleans_LA	locatedIn	y:United_States
y:Franklin_Roosevelt	hasName	"Franklin D. Roosevelt"
y:Franklin_Roosevelt	bornIn	y:Hyde_Park_NY
y:Franklin_Roosevelt	title	"President"
y:Franklin_Roosevelt	gender	"Male"
y:Hyde_Park_NY	foundingYear	"1810"
y:Hyde_Park_NY	locatedIn	y:United_States
y:Marilyn_Monroe	gender	"Female"
y:Marilyn_Monroe	hasName	"Marilyn Monroe"
y:Marilyn_Monroe	bornOnDate	"1926-07-01"
y:Marilyn_Monroe	diedOnDate	"1962-08-05"

(a) RDF Triples (Prefix: y=http://en.wikipedia.org/wiki/)



(b) RDF Graph G

Fig. 1: RDF Graph

that was founded in 1718, but we may not know the exact birth date. In this case, we have to perform a query with wildcards, as shown below (Q_2):

```
SELECT ?name WHERE
{?m <bornIn> ?city . ?m <hasName> ?name .
  ?m <bornOnDate> ?bd .
  ?city <foundingYear> ''1718''.
  FILTER(regex(str(?bd),''1976''))}
```

– *Dynamic RDF repositories.* RDF repositories are not static, and are updated regularly. For example, Yago and DBpedia datasets are continually expanding to 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.
– *Aggregate SPARQL queries.* Few existing works and SPARQL engines consider aggregate queries despite their real-life importance. A typical aggregate SPARQL query that groups all individuals by their titles, genders, and the founding year of their birth places, and reports the number of individuals in each group is shown below (Q_3):

```
SELECT ?t ?g ?y COUNT(?m) WHERE
{?m <bornIn> ?c. ?m <title> ?t.
  ?m <gender> ?g. ?c <foundingYear> ?y.}
GROUP BY ?t ?g ?y
```

In this paper we describe gStore, which is a graph-based triple store that can answer the above discussed SPARQL queries over dynamic RDF data repositories. In this context, answering a query is transformed into a subgraph matching problem. Specifically, we 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. We also represent a given SPARQL query by a *query graph*, Q. Subgraph matching of the query graph Q over the RDF graph G provides the answer to the query.

For example, Figure 1b shows an RDF graph G corresponding to RDF triples in Figure 1a. We formally define an RDF graph in Definition 1. Note that, the numbers above the boxes in Figure 1b are not vertex labels, but vertex IDs that we introduce to simplify the description. The RDF graph does not have to be connected. A SPARQL query can also be represented as a directed labeled query graph Q (Definition 2). Figure 2 shows the query graph corresponding to the SPARQL query Q_2. Usually, query graph Q is a connected graph. Otherwise, we can regard each connected component of Q as a separate query and perform them one by one.

We develop novel indexing and graph matching techniques rather than using existing ones. This is because the characteristics of an RDF graph are considerably different from graphs typically considered in much of the graph database research. First, the size of an RDF graph (i.e., the number of vertices and edges) is larger

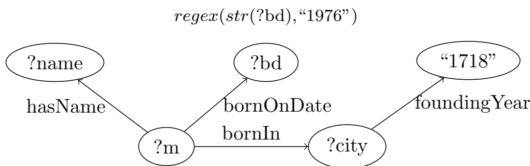


Fig. 2: Query Graph of Q_2

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 [25,32] 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 [19]. A *star query* refers to the query graph in the shape of a star, formed by one central vertex and its neighbors.

Contributions of this paper are the following:

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. An index (VS*-tree) is developed 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 SPARQL queries efficiently.
4. We introduce an auxiliary structure (called T-index), which is a structured organization of materialized views, to speed up aggregate SPARQL queries.
5. We demonstrate experimentally that the performance of our approach is superior to existing systems.

The rest of this paper is organized as follows. We discuss the related work and preliminaries in Sections 2 and 3, respectively. We give an overview of our solution in Section 4. We discuss the storage and encoding method in Section 5. We then present the VS*-tree index in Section 6 and an algorithm for SPARQL query processing in Section 7. In order to support aggregate queries efficiently, we develop T-index in Section 8 and aggregate query processing algorithm in Section 9. We discuss the maintenance of indexes (VS*-tree and T-index) as RDF data get updated in Section 10. We study our methods by experiments in Section 11. Section 12 concludes this paper. Some of the additional material supporting the main findings reported in the paper are included in an Online Supplement.

2 Related Work

Three approaches have been proposed to store and query RDF data: one giant triples table, clustered property tables, and vertically partitioned tables.

One giant triples table. The systems in this category store RDF triples in a single three-column table where columns correspond to subject, property, and object (as in Figure 1a) enabling them to manipulate all RDF triples in a uniform manner. However, this requires performing a large number of self-joins over this table to answer a SPARQL query. Some efforts have been made to address this issue, such as, RDF-3x [19,20] and Hexastore [30], which build several clustered B^+-trees for all permutations of s, p and o columns.

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

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.

Existing RDF storage systems, such as Jena [31], Yars2 [14] and Sesame 2.0 [6], do not work well in large RDF datasets. SW-store [1], RDF-3x [19], x-RDF-3x [22] and Hexastore [30] are designed to address scalability, however, they only support exact SPARQL queries, since they replace all literals (in RDF triples) by ids using a mapping dictionary.

Furthermore, most of existing methods do not efficiently handle online updates of the underlying RDF repositories. For example, in clustered property table-based methods (such as Jena [31]), if there are updates to the properties in RDF triples, it is necessary to re-cluster and re-build the property tables. In SW-store [1], it is potentially expensive to insert data, since

each update requires writing to many columns. In order to address this issue, it uses "overflow table + batch write", meaning that online updates are recorded to overflow tables that SW-store periodically scans to materialize the updates. Obviously, this kind of maintenance method cannot work well for applications such as online social networks that require real time access.

More recent x-RDF-3x [22] proposes an efficient online maintenance algorithm, but does not support wildcard or aggregate SPARQL queries. There exist some works that discuss the possibility of storing RDF data as a graph (e.g., [5,30]), but these approaches do not address scalability. Some are based on main memory implementations [26], while others utilize graph partitioning to reduce self-joins of triple tables [33]. While graph partitioning is a reasonable technique to parallelize execution, updates to the graph may require re-partitioning unless incremental partitioning methods are developed (which are not in these works).

Few SPARQL query engines consider aggregate queries, and to the best of our knowledge only two proposals exist in literature [16,24]. Given an aggregate SPARQL query Q, a straightforward method [16] is to transform Q into a SPARQL query Q' without aggregation predicates, find the solution to Q' by existing query engines, then partition the solution set into one or more groups based on rows that share the specified values, and finally, compute the aggregate values for each group. Although it is easy for existing RDF engines to implement aggregate functions this way, the approach is problematic, since it misses opportunities for query optimization. Furthermore, it has been pointed out [24] that this method may produce incorrect answers.

Seid and Mehrotra [24] study the semantics of group-by and aggregation in RDF graph and how to extend SPARQL to express grouping and aggregation queries. They do not address the physical implementation or query optimization techniques.

Finally, the RDF data tend not to be very structured. For example, each subject of the same type do not need to have the same properties. This facilitates "pay-as-you-go" data integration, but prohibits the application of classical relational approaches to speed up aggregate query processing. For example, materialized views [13], which are commonly used to optimize query execution, may not be used easily. In relational systems, if there is a materialized view V_1 over dimensions (A, B, C), an aggregate query over dimensions (A, B) can be answered by only scanning view V_1 rather than scanning the original table. However, this is not always possible in RDF. For example, consider Q_3 that groups all individuals by their titles, gender, and founding year of their birth places and reports the number of individuals in each group. The answer to this query, $R(Q_3)$, is given in Figure 3a (we show how to compute this answer in Section 9).

Now consider another query (say Q_4) that groups all individuals by their titles and gender and reports the number of individuals in each group. The answer to this query is given in Figure 3b. Although the group-by dimensions in Q_4 is a subset of those in Q_3, it is not possible to get the aggregate result set $R(Q_4)$ by scanning $R(Q_3)$. The main reason is the nature of RDF data and the fact that RDF data tend not be structured, and there may be subjects of the same type that do not have the same properties. Therefore, some subjects that exist in a "smaller" materialized view may not occur in a "larger" view.

title	gender	foundingYear	COUNT
President	Male	1718	1
Actress	Female	1976	1

(a) Answer to Query Q_3

title	gender	COUNT
President	Male	2
Actress	Female	1

(b) Answer to Query Q_4

Fig. 3: Difficulty of Using Materialized Views

3 Preliminaries

An RDF data set is a collection of (**s**ubject, **p**roperty, **o**bject) triples $\langle s, p, o \rangle$, where *subject* is an entity or a class, and *property* denotes one attribute associated with 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). 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 do not distinguish between an "entity" and a "class" since we have the same operations over them. RDF data can be modeled as an RDF graph, which is formally defined as follows (frequently used symbols are shown in Table 1):

Definition 1 A *RDF graph* is a four-tuple $G = \langle V, L_V, E, L_E \rangle$, where

1. $V = V_c \cup V_e \cup V_l$ 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.

Table 1: Frequently-used Notations

Notation	Description	Notation	Description
G	A RDF graph (Definition 1)	Q	A SPARQL query graph (Definition 2)
v	A vertex v in a SPARQL query graph (Definition 3)	u	a vertex in RDF graph G (Definition 3)
$eSig(e)$	an edge Signature (Definition 5)	$vSig(u)$	a vertex signature (Definition 6)
RS	The answer set of the SPARQL query matches	CL	The candidate set of the SPARQL query matches
G^*	A data signature graph (Definition 7)	Q^*	A query signature graph
$A(u)$	A transaction of vertex u (Definition 16)	O	A node in T-index (Definition 16)
$O.L$	The corresponding vertex list of node O	$MS(O)$	The corresponding aggregate set of node O

2. L_V is a collection of vertex labels. The label of a vertex $u \in V_l$ is its literal value, and the label of a vertex $u \in V_c \cup V_e$ is its corresponding URI.
3. $E = \{\overrightarrow{u_1, u_2}\}$ is a collection of directed edges that connect the corresponding subjects and objects.
4. L_E is a collection of edge labels. Given an edge $e \in E$, its edge label is its corresponding property.

An edge $\overrightarrow{u_1, u_2}$ is an *attribute property* edge if $u_2 \in V_l$; otherwise, it is a *link* edge. □

Figure 1b 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. Some triples in Q have *parameters*. In Q_2 (in Section 1), "?m" and "?bd" are parameters, and "?bd" has a wildcard filter: FILTER(regx(str(?bd),"1976")). Figure 2 shows the query graph that corresponds to Q_2.

Definition 2 A *query graph* is a five-tuple $Q = \langle V^Q, L_V^Q, E^Q, L_E^Q, FL \rangle$, where

1. $V^Q = V_c^Q \cup V_e^Q \cup V_l^Q \cup V_p^Q$ is a collection of vertices that correspond to all subjects and objects in a SPARQL query, where V_p^Q is a collection of parameter vertices, and V_c^Q and V_e^Q and V_l^Q are collections of class vertices, entity vertices, and literal vertices in the query graph Q, respectively.
2. L_V^Q is a collection of vertex labels in Q. The label of a vertex $v \in V_p^Q$ is ϕ; that of a vertex $v \in V_l^Q$ is its literal value; and that of a vertex $v \in V_c^Q \cup V_e^Q$ is its corresponding URI.
3. E^Q is a collection of edges that correspond to properties in a SPARQL query. L_E^Q is the edge labels in E^Q. An edge label can be a property or an edge parameter.
4. FL are constraint filters, such as a wildcard constraint. □

Note that, in this paper, we do not consider SPARQL queries that involve type reasoning/inferencing. Thus, the match of a query is defined as follows.

Definition 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, u_i should satisfy the filter constraint over parameter vertex v_i if any; otherwise, there is no constraint over u_i;
4. 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. If the edge label in Q is p (i.e., property), the edge from u_i to u_j in G has the same label. If the edge label in Q is a parameter, the edge label should satisfy the corresponding filter constraint; otherwise, there is no constraint over the edge label from u_i to u_j in G. □

Given Q_2's query graph in Figure 2, vertices (005,006, 020,023,024) in RDF graph G of Figure 1b 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 4 An aggregate SPARQL query Q consists of three components:

1. *Query pattern* P_Q is a set of triple statements that form one query graph.
2. *Group-by dimensions* and *measure dimensions* are pre-defined object variables in query pattern P_Q.
3. (Optional) *HAVING condition* specifies the condition(s) that each result group must satisfy. □

Figure 4 demonstrates these three components. This example shows the case where all group-by dimensions correspond to attribute property edges (e.g., "?g","?t" and "?y" in Figure 4). This is not necessary, and in Section 9.3 we discuss more general cases.

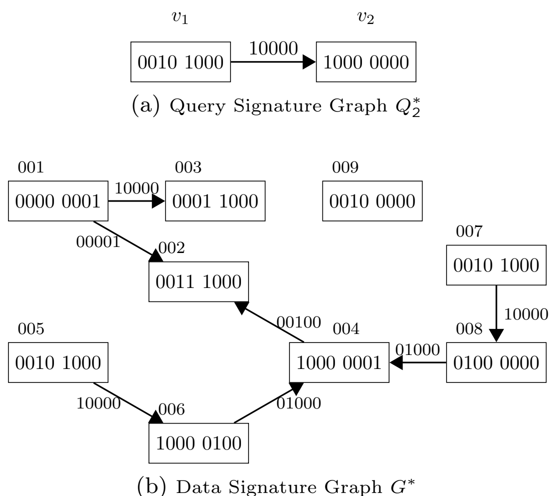
```
SELECT ?g ?t ?y COUNT(?m) WHERE
{?m <bornIn> ?c. ?m <title> ?t.
?m <gender> ?g.
?c <foundingYear>?y.}
GROUP BY ?g,?t,?y
HAVING COUNT(?m)>1
```

Measure dimension
Query pattern
Group-by dimension
HAVING condition (optional)

Fig. 4: Three Components in Aggregate Queries

4 Overview of gStore

gStore is a graph-based triple store system that can answer different kinds of SPARQL queries – exact queries, queries with wildcards and aggregate queries – over dynamic RDF data repositories. An RDF dataset is represented as an RDF graph G and stored as an adjacency list table (Figure 7). Then, each entity and class vertex is encoded into a bitstring (called *vertex signature*). The encoding technique is discussed in Section 5. According to RDF graph's structure, these vertex signatures are linked to form a *data signature graph* G^*, in which each vertex corresponds to a class or an entity vertex in the RDF graph (Figure 5). Specifically, G^* is induced by all entity and class vertices in G together with the edges whose endpoints are either entity or class vertices. Figure 5b shows the data signature graph G^* that corresponds to RDF graph G in Figure 1b. An incoming SPARQL query is also represented as a *query graph* Q that is similarly encoded into a *query signature graph* Q^*. The encoding of query Q_2 (which we will use as a running example) into a query signature graph Q_2^* is shown in Figure 5a.



(a) Query Signature Graph Q_2^*

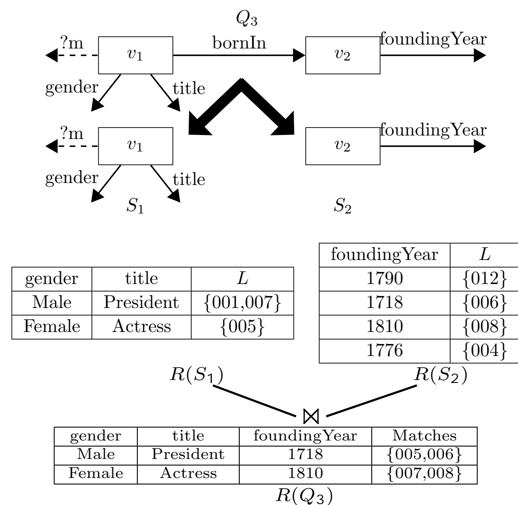
(b) Data Signature Graph G^*

Fig. 5: Signature Graphs

Finding matches of Q^* over G^* is known to be NP-hard since it is analogous to subgraph isomorphism. Therefore, we use a filter-and-evaluate strategy to reduce the search space over which we do matching. We first use a false-positive pruning strategy to find a set of candidate subgraphs (denoted as CL), and then we validate these using the adjacency list to find answers (denoted as RS). Reducing the search space has been considered in other works as well (e.g.,[25,32]).

According to this framework, two issues need to be addressed. First, the encoding technique should guarantee that $RS \subseteq CL$. Second, an efficient subgraph matching algorithm is required to find matches of Q^* over G^*. To address the first issue, we develop an encoding technique (Section 5) that maps each vertex in G^* to a signature. For the second issue, we design a novel index structure called VS*-tree (Section 6). VS*-tree is a summary graph of G^* used to efficiently process queries using a pruning strategy to reduce the search space for finding matches of Q^* over G^* (Section 7).

VS*-tree is also used in answering aggregate SPARQL queries (Section 9). We first decompose an aggregate query Q into *star aggregate queries* S_i ($i = 1, ..., n$), where each star aggregate query is formed by one vertex (called *center*) and its adjacent properties (i.e., adjacent edges). For example, query Q_3 is decomposed into two star aggregate queries S_1 and S_2 whose graph patterns are shown in Figure 6. We make use of materialized views to efficiently process star aggregate queries without performing joins. For this purpose, we introduce T-index (Section 8), which is a trie where each node O has a materialized set of tuples $MS(O)$. A star aggregate query can be answered by grouping materialized sets associated with nodes in T-index. Once the results $R(S_i)$ of star aggregate queries S_i ($i = 1, ..., n$) are obtained, we employ VS*-tree to join $R(S_i)$ and find all relevant nodes for each star center. Then, based on these relevant nodes, we can find the final result of aggregate queries.



gender	title	L
Male	President	{001,007}
Female	Actress	{005}

foundingYear	L
1790	{012}
1718	{006}
1810	{008}
1776	{004}

gender	title	foundingYear	Matches
Male	President	1718	{005,006}
Female	Actress	1810	{007,008}

Fig. 6: Aggregate Queries

gStore considers RDF data management in a dynamic environment, i.e., as the underlying RDF data get updated, VS*-tree and T-index are adjusted accord-

ingly. Therefore, we also address the index maintenance issues of VS*-tree and T-index (Section 10).

5 Storage Scheme and Encoding Technique

We develop a graph-based storage scheme for RDF data. Specifically, we store an RDF graph G using a disk-based adjacency list. Each (class or entity) vertex u is represented by an adjacency list $[uID, uLabel, adjList]$, where uID is the vertex ID, $uLabel$ is the corresponding URI, and $adjList$ is the list of its *outgoing* edges and the corresponding neighbor vertices. Formally, $adjList(u) = \{(e_i.eLabel, e_i.nLabel)\}$, where e_i is an edge adjacent to u, $eLabel$ is e_i's edge label that corresponds to some property, and $nLabel$ is the vertex label of u's neighbor connected via e_i. When clear, we omit "e_i." prefix from the specification of $adjList$. Figure 7 shows part of the adjacency list table for the RDF graph in Figure 1b.

uID	$uLabel$	$adjList$
001	y:Abraham_Lincoln	(hasName, \Abraham Lincoln"), (bornOnDate, \1809-02-12"), (diedOnDate, \1865-04-15"), (diedIn, y:Washington_DC), (gender, \Male"), (title, \President"), (bornIn, y:Hodgenville_KY)
002	y:Washington_DC	(hasName, \Washington D.C."), (foundingYear, \1790")
...	...	...

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

Fig. 7: Disk-based Adjacency List Table

According to Definition 3, 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. 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. Consequently, the match between Q and G can be verified by simply checking the match between corresponding encoded bitstrings.

Each row in the table corresponds to an entity vertex or a class vertex. Given a vertex, we encode each of its adjacent edges $e(eLabel, nLabel)$ into a bitstring. This bitstring is called *edge signature* (i.e., $eSig(e)$).

Definition 5 The *edge signature* of an edge adjacent to vertex u, $e(eLabel, nLabel)$, 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., $nLabel$). □

$eSig(e).e$ and $eSig(e).n$ are generated as follows. 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, in our implementation, we employ m different string hash functions H_i ($i = 1, ..., m$), such as BKDR and AP hash functions [7]. 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.

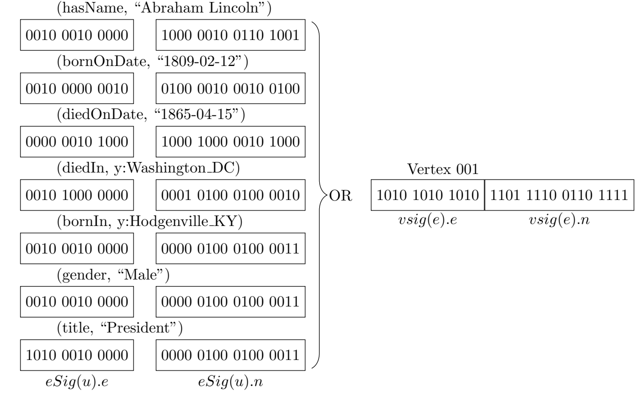


Fig. 8: The Encoding Technique

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 [10], 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 use a string hash function H for each n-gram g to obtain $H(g)$. Finally, we set the ($H(g)$ MOD N)-th bit in $eSig(e).n$ to be '1'. We discuss the settings of parameters M, m, N and n in Section 11.2. Figure 8 shows a running example of edge signatures. For example, given 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 6 Given a class or entity vertex u in the RDF graph, the *vertex signature* $vSig(u)$ is formed by performing bitwise OR operations over all its adjacent edge signatures. Formally, $vSig(u)$ is defined as follows:

$$vSig(u) = eSig(e_1)| \dots |eSig(e_n)$$

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

Considering vertex 001 in Figures 1b and 7, there are seven adjacent edges. We can encode each adjacent edge by its edge signature, as shown in Figure 8, which also shows the signature of vertex 001.

In computing the vertex signature we use textual value of the neighbor node, not its vertex signature. For

example, in computing the signature of y:Abraham_Lincoln that has in its adjacency list (diedIn, y:Washington_DC), we simply encode string "y:Washington_DC" and use this encoding rather than the vertex signature of node y:Washington_DC. This avoids recursion in the computation of vertex signatures.

Definition 7 Given an RDF graph G, its corresponding data signature graph G^* is induced by all entity and class vertices in G together with link edges (the edges whose endpoints are either entity or class vertices). Each vertex u in G^* has its corresponding vertex signature $vSig(u)$ (Definition 6) as its label. Given an edge $\overrightarrow{u_1, u_2}$ in G^*, its edge label is also a signature, denoted as $Sig(\overrightarrow{u_1 u_2})$, to denote the property between u_1 and u_2. □

We adopt the same hash function in Definition 5 to define $Sig(\overrightarrow{u_1, u_2})$. Specifically, we set m out of M bits in $Sig(\overrightarrow{u_1, u_2})$ to be '1' by some string hash function. Figure 5 shows an example of data signature graph G^*.

We also encode the query graph Q using the same method. Specifically, given an entity or class vertex v in Q, we encode each adjacent edge pair $e(eLabel, nLabel)$ into a bitstring $eSig(e)$ according to Definition 5. 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 parameter vertex and there is a wildcard constraint (e.g., regex(str(?bd),"1976")), we only consider the substring without "wildcard" in the label. For example, in Figure 2, we can only encode substring "1976" for vertex ?bd. 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 $\overrightarrow{v_1, v_2}$ in Q^* is associated with an edge signature $Sig(\overrightarrow{v_1, v_2})$. Figure 5 shows Q^* that corresponds to query Q_5.

Definition 8 Consider a data signature graph G^* and a query signature graph Q^* that has n vertices $\{v_1, \ldots, v_n\}$. A set of n distinct vertices $\{u_1, \ldots, u_n\}$ 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^*; and $Sig(\overrightarrow{v_i, v_j})\&Sig(\overrightarrow{u_i, u_j}) = Sig(\overrightarrow{v_i, v_j})$. □

Note that, each vertex u (and v) in data (and query) signature graph G^* (and Q^*) has one vertex signature $vSig(u)$ (and $vSig(v)$). For simplicity, 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 1 *$RS \subseteq CL$ holds.*

Proof See Part B of Online Supplements.

6 VS*-tree

In this section, we describe VS*-tree, which is an index structure over G^* that can be used to answer SPARQL queries as described in the next section. As discussed earlier, the key problem to be addressed is how to find matches of Q^* (query signature graph) over G^* (data signature graph) efficiently using Definition 8. A straightforward method can work as follows: first, for each vertex $v_i \in Q^*$, we find a list $R_i = \{u_{i_1}, u_{i_2}, ..., u_{i_n}\}$, where $v_i \& u_{i_j} = v_i$ (& is a bitwise AND operation). Then, we perform a multiway join over these lists R_i to find matches of Q^* over G^* (finding CL). The first step (finding R_i) is a classical *inclusion query*.

An inclusion query is a subset query that, given a set of objects with set-valued attributes, finds all objects containing certain attribute values. In our case, presence of elements in sets is captured in signatures. Thus, we have a set of signatures $\{s_i\}$ (representing a set of objects with set-valued attributes) and a query signature q [27]. Then, an inclusion query finds all signatures $\{s_j\} \subseteq \{s_i\}$, where $q \& s_j = q$.

In order to reduce the search space for the inclusion query, S-tree [8], a height-balanced tree similar to B$^+$-tree, has been proposed to organize all signatures $\{s_j\}$. Each intermediate node is formed by ORing all child signatures in S-tree. Therefore, S-tree can be employed to support the first step efficiently, i.e., finding R_i. An example of S-tree is given in Figure 9.

However, S-tree cannot support the second step (i.e., a multiway join), which is NP-hard. Although many subgraph matching methods have been proposed (e.g., [25,32]), they are not scalable to very large graphs. Therefore, we develop VS*-tree (*v*ertex *s*ignature *tree*) to index a large data signature graph G^* that also supports the second step. VS*-tree is a *multi-resolution summary graph* based on S-tree that can be used to reduce the search space of subgraph query processing.

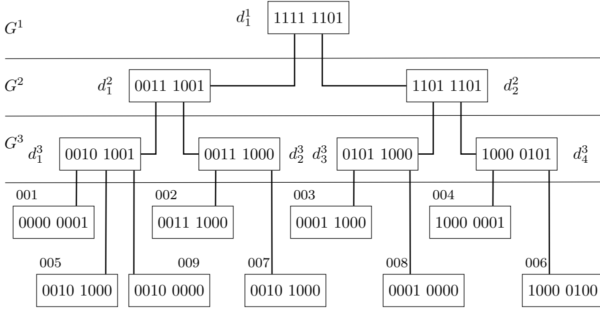


Fig. 9: S-tree

Definition 9 **VS*-tree** is a height balanced tree with the following properties:

1. Each path from the root to any leaf node has the same length h, which is the height of the VS*-tree.
2. Each leaf node corresponds to a vertex in G^* and has a pointer to it.
3. Each non-leaf node has pointers to its children.
4. The level numbers of VS*-tree increase downward with the root at level 1.
5. Each node d_i^I at level I is assigned a signature $d_i^I.Sig$. If node d_i^I is a leaf node, $d_i^I.Sig$ is the corresponding vertex signature in G^*. Otherwise, $d_i^I.Sig$ is obtained by "OR"ing signatures of d_i^I's children (i.e., $d_i^I.Sig = OR(d_1^{I+1}.Sig, \ldots, d_n^{I+1}.Sig)$) where d_j^{I+1} are d_i^I's children and $j = 1, \ldots, n$.
6. Each node other than the root has at least b children.
7. Each node has at most B children, $\frac{B+1}{2} \geq b$.
8. Given two nodes d_i^I and d_j^I, there is a super-edge $\overrightarrow{d_i^I, d_j^I}$ if and only if there is an edge (can be a super-edge) from at least one of d_i^I's children to one of d_j^I's children. The edge label $Sig(\overrightarrow{d_i^I, d_j^I})$ is created by "OR"ing signatures of all edge labels from d_i^I's children to d_j^I's children.
9. 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. □

According to Definition 9, the leaf nodes of VS*-tree correspond to vertices in G^*. An S-tree is built over these leaf nodes. Furthermore, each pair of leaf nodes (u_1, u_2) is connected by a directed "super-edge" $\overrightarrow{u_i, u_j}$ if there is a directed edge from u_1 to u_2 in G^*, and an edge signature $Sig(\overrightarrow{u_1, u_2})$ is assigned to it according to Definition 7. Figure 10 illustrates the process; for example, a super-edge is introduced from (008) to (004) (shown as a dashed line) since such an edge exists in G^*. In this figure we assume that the hash function for edge labels are the following: bornIn → 10000, diedIn → 00001, hasCapital → 00100, and locatedIn → 01000. Finally, given two non-leaf nodes at the same level d_i^k and d_j^k (where d_i^k denotes a node at the k-th level), a super-edge $\overrightarrow{d_i^k, d_j^k}$ is introduced if and only if there is an edge from any of d_i's children to any of d_j's children. The edge label of $\overrightarrow{d_i^k, d_j^k}$ is obtained by performing bitwise OR over all the edge labels from d_i's children to d_j's children. In Figure 10, since there is a super-edge from 008 to 004, we introduce a super-edge from d_3^3 to d_4^3 (i.e., $\overrightarrow{d_i^k, d_j^k}$) with signature 01000. We also introduce a self-edge for a non-leaf node d_i^k, if and only if there is an edge from one of its children to another one of its children. Thus, a self-loop super-edge over d_4^3 is also introduced since there is a (super-)edge from 006 to 004, both of which are children of d_4^3.

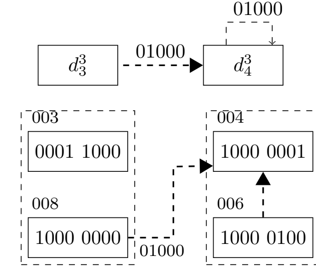


Fig. 10: Building Super-edges

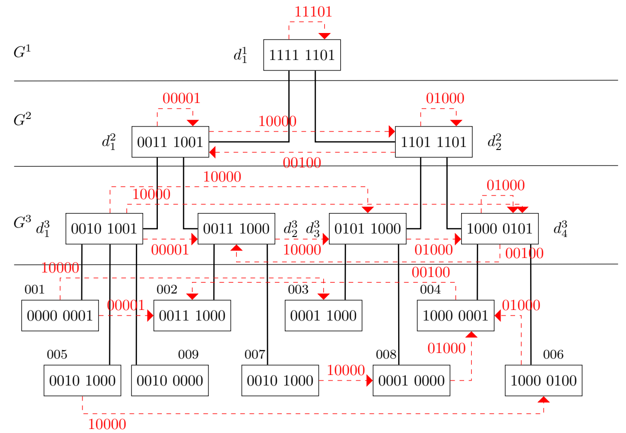


Fig. 11: VS*-tree

Figure 11 shows the VS*-tree over G^* of Figure 5. As defined in Definition 9, we use $d_i^I.Sig$ to denote the signature associated with node d_i^I. For simplicity, we also use d_i^I to denote $d_i^I.Sig$ when the context is clear.

Definition 10 Consider a query signature graph Q^* with n vertices v_i $(i = 1, ..., n)$ and a summary graph G^I at the I-th level of VS*-tree. A set of nodes $\{d_i^I\}$ $(i = 1, ..., n)$ at G^I is called a *summary match* of Q^* over G^I, if and only if the following conditions hold:

1. $vSig(v_i) \& d_i^I.Sig = vSig(v_i)$, $i = 1, ..., n$;
2. Given edge $\overrightarrow{v_1, v_2} \in Q^*$, there exists a super-edge $\overrightarrow{d_1^I d_2^I}$ in G^I and $Sig(\overrightarrow{v_1, v_2}) \& Sig(\overrightarrow{d_1^I, d_2^I}) = Sig(\overrightarrow{v_1, v_2})$. □

Note that, a summary match is not an injective function from $\{v_i\}$ to $\{d_i^I\}$, namely, d_i^I can be identical to d_j^I. For example, given a query signature graph Q^* (in Figure 5) and a summary graph G^3 of VS*-tree (in Figure 11), we can find one summary match $\{(d_1^3, d_4^3)\}$. Summary matches can be used to reduce the search space for subgraph search over G^* as we discuss next.

As we discuss in Section 7, the query algorithm uses level-by-level matching of vertex signatures of the query graph Q^* to the nodes in VS*-tree. A problem that may affect the performance of the query algorithm is that the vertex encoding strategy may lead to some vertex signatures having too many "1"s. Given a vertex u, we perform bitwise OR operations over all its adjacent edge signatures to obtain vertex signature $vSig(u)$ (see Definition 6). Therefore, for a vertex with a high degree, $vSig()$ may be all (or mostly) 1's, meaning that these vertices can match any query vertex signature. This will affect the pruning power of VS*-tree.

In order to address this issue, we perform the following optimization. Given a vertex u, if the number of 1's in $vSig(u)$ is larger than some threshold δ, we partition all of u's neighbors into n groups $g_1, ..., g_n$ and each group g_i corresponds to one instance of vertex u, denoted as $u_{[i]}$. According to Definition 6, we can compute vertex signature for these instances, i.e., $vSig(u_{[i]})$. We guarantee that the number of 1's in $vSig(u_{[i]})$ $(i = 1, ..., n)$ is no larger than δ. Given a vertex u, if the number of 1's in $vSig(u)$ is no larger than δ, u has only one instance $u_{[1]}$. Then, we use these vertex instance signatures of u $(vSig(u_{[i]}), i = 1, ..., n)$ instead of vertex signature $(vSig(u))$ in building VS*-tree. Note that, these vertex instances have the same vertex ID of u that corresponds to the same vertex in RDF graph G. Given two instances $vSig(u_{[i]})$ and $vSig(u'_{[j]})$ at the leaf level of the revised VS*-tree, we introduce an edge between $u_{[i]}$ and $u'_{[j]}$ if and only if there is an edge between their corresponding vertices u and u'.

For example, vertex 001 has seven neighbors in RDF graph G. We can decompose them into two groups g_1 and g_2 and encode the two groups as in Figure 12a. Each group corresponds to one instance, denoted as $001_{[1]}$ and $001_{[2]}$. Assume that other vertices have only one instance. Since there is an edge from 001 to 003, we introduce edges from $001_{[1]}$ to $003_{[1]}$ and from $001_{[2]}$ to $003_{[1]}$, as shown in Figure 12b.

Given a vertex v in query graph, if v can match vertex u in RDF graph G, v's neighbor vertices may match neighbors of different instances of u. Therefore, we need to revise encoding strategy for query graph Q. Assume that v in the query graph has m neighbors. We introduce m instances of v, i.e., $v_{[j]}$, $j = 1, ..., m$, and each instance has only one neighbor. According to Definition 6, we can compute the vertex signature for these instances, i.e., $vSig(v_{[j]})$. For example, v_1 in Q_2^* (Figure 5) has three neighbors, thus, we introduce three instances $v_{1[1]}$, $v_{1[2]}$, and $v_{1[3]}$ that corresponds to three neighbors, respectively, Figure 12b.

7 SPARQL Query Processing

Given a SPARQL query Q, we first encode it into a query signature graph Q^*, according to the method in Section 5. Then, we find matches of Q^* over G^*. Finally, we verify if each match of Q^* over G^* is also a match of Q over G following Definition 3. Therefore, the key issue is how to efficiently find matches of Q^* over G^* using the VS*-tree.

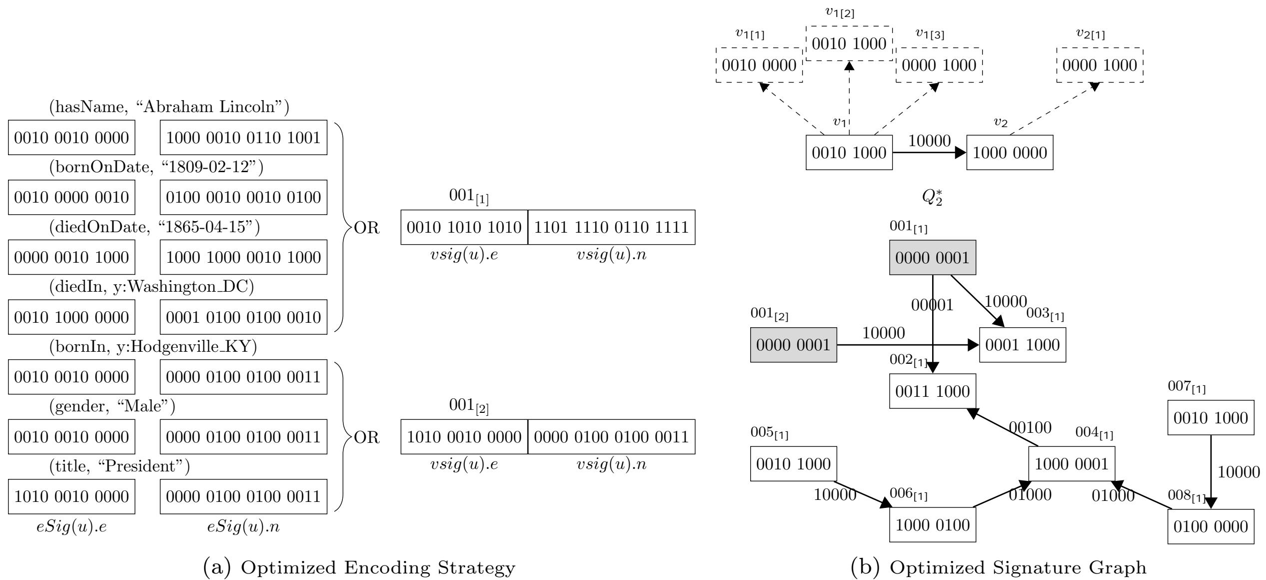
We employ a top-down search strategy over the VS*-tree to find matches. According to Theorem 2, the search space at level $I + 1$ of the VS*-tree is bounded by the summary matches at level I (level numbers increase downward with the root at level 1). This allows us to reduce the total search space.

Theorem 2 *Given a query signature graph Q^* with n vertices $\{v_1, \ldots, v_n\}$, a data signature graph G^* and the VS*-tree built over G^*:*

1. *Assume that n vertices $\{u_1, \ldots, u_n\}$ forms a match (Definition 8) of Q^* over G^*. Given a summary graph G^I in VS*-tree, let u_i's ancestor in G^I be node d_i^I. $(d_1^I, ..., d_n^I)$ must form a summary match (Definition 10) of Q^* over G^I.*
2. *If there exists no summary match of Q^* over G^I, there exists no match of Q^* over G^*.*

Proof See Part B of Online Supplements.

The basic query processing algorithm (BVS*-Query) is given in Algorithm 1. We illustrate it using a running example Q_2^* (of Figure 5). Figure 13 shows the process (pruned search space is shown as shaded). First, we find summary matches of Q_2^* over G^1 in VS*-tree and insert them into a queue H. In this case, the summary match is $\{(d_1^1, d_1^1)\}$, which goes into queue H. We pop one summary match from H and expand it



(a) Optimized Encoding Strategy (b) Optimized Signature Graph

Fig. 12: Optimizing VS*-tree

to its *child states* (as given in Definition 11). Given the summary match (d_1^1, d_1^1), its child states are formed by $d_1^1.children \times d_1^1.children = \{d_1^2, d_2^2\} \times \{d_1^2, d_2^2\} = \{(d_1^2, d_1^2), (d_1^2, d_2^2), (d_2^2, d_1^2), (d_2^2, d_2^2)\}$. The set of child states that are summary matches of Q^* are called *valid child states*, and they are inserted into queue H. In this example, only (d_1^2, d_2^2) is a summary match of Q^*, thus, we insert it into H. We continue to iteratively pop one summary match from H and repeat this process until the leaf nodes (i.e., vertices in G^*) are reached. Finally, we find matches of Q^* over leaf entries of VS*-tree, namely, the matches of Q^* over G^*.

Algorithm 1 Basic Query Algorithm Over VS*-tree (BVS*-Query)

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, and insert into queue H
3: **while** $(|H| > 0)$ **do**
4: Pop one summary match from H, denoted as SM
5: **for** each child state S of SM **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: Insert it into queue H
10: **return** CL.

Definition 11 Given a query signature graph Q^* with n vertices $\{v_1, \ldots, v_n\}$, and n nodes $\{d_1^I, ..., d_n^I\}$ in VS*-tree that form a summary match of Q^*, n nodes $\{d_1^{I'}, ..., d_n^{I'}\}$ form 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 $\{d_1^I, ..., d_n^I\}$. □

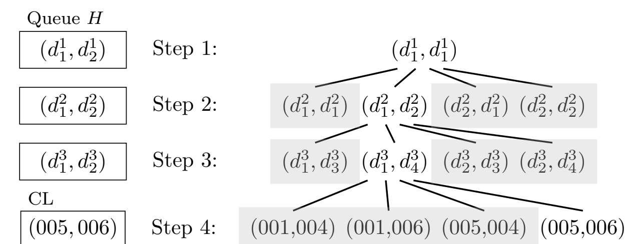


Fig. 13: BVS*-Query Algorithm Process

Algorithm 1 always begins by finding summary matches from the root of the VS*-tree, which can lead to a large number of intermediate summary matches. In order to speed up query processing, we do not materialize all summary matches in non-leaf levels. Instead, we apply a semijoin [4] like pruning strategy, i.e., pruning some nodes (in the VS*-tree) that are not possible in any summary match, and incorporate it into VS*-query algorithm.

Given a vertex v_i in query graph Q^* and a summary graph G^I in VS*-tree, we try to prune nodes d^I ($\in G^I$)

that cannot match v_i in any summary match of Q^* over G^I. At the leaf-level of the index, this shrinks the candidate list $C(v_i)$. Let us recall query Q_2^* in Figure 5. Vertices v_1 and v_2 have three $\{002, 005, 007\}$ and two candidates $\{004, 006\}$, respectively, according to their vertex signatures. Therefore, the whole join space is $3 \times 2 = 6$. Let us consider summary graph G^3. If we use S-tree for pruning, we get two candidates (d_1^3 and d_2^3) for v_1 and one candidate (d_4^3) for v_2. There is one edge from v_1 to v_2 whose label signature is 10000. d_2^3 has one edge to candidates of v_2 with an edge signature of 10000. Obviously $00010\&100000 \neq 10000$, and, therefore, d_2^3 cannot be in the candidate list of v_1. If a node cannot match v_1, all descendent nodes of d can be pruned safely. It means that node d_2^3 and its descendent nodes are pruned from candidates of v_1 resulting in a single candidate $\{005\}$ for v_1. Therefore, the join space is reduced to $1 \times 2 = 2$.

Algorithm 2 An Optimized Query Algorithm Over VS*-tree (VS*-Query Algorithm)

Input: a query signature graph Q^*
Input: a data signature graph G^*
Input: a VS*-tree.
Output: CL: All matches of Q^* over G^*.

```
1: for each vertex v_i ∈ Q* do
2:    C(v_i) = d_1^1, where d_1^1 is the root of VS*-tree {C(v_i)
      are candidate vertices in G* to match v_i in Q*}
3: for each level summary graph G^I of VS*-tree I = 2, ..., h
   do {h is the height of VS*-tree}
4:    for each C(v_i), i = 1, ..., n do
5:       set C'(v_i) = φ
6:       for each child node d^I of each element in C(v_i) do
7:          for each instance v_i[j] of vertex v_i, j = 1, ..., m
            do
8:             if Sig(d^I)&sig(v_i[j]) = sig(v_i[j]) then
9:                push d^I into C(v_i[j])
10:      Set C(v_i) = ∩_{j=1,...,m} C(v_i[j])
11:   for each C(v_i), i = 1, ..., n do
12:      for each each node d in C(v_i) do
13:         if ∃v_j | →(v_i, v_j) ∈ Q*, ∀d' | →(d, d') ∈
            G^I, vSig(v_j)&vSig(d') ≠ vSig(v_j) then
14:            Remove v_i from C(v_i)
15:         if ∃v_j | →(v_i, v_j) ∈ Q*, ∀d' | →(d, d') ∈
            G^I, Sig(→(v_i, v_j))&Sig(→(d, d')) ≠ Sig(→(v_i, v_j)) then
16:            Remove v_i from C(v_i)
17: Call Algorithm 3 to find matches of Q* over C(v_1) × ... ×
    C(v_n), denoted as CL.
18: for each candidate match in CL do
19:    Check whether it is match of SPARQL query Q over
       RDF graph G. If so, insert it into RS.
20: return RS.
```

The basic BVS-Query algorithm can be improved by reducing the candidates for each query vertex in Q^* instead of finding summary matches in non-leaf levels. The improved algorithm, called VS*-Query, is given

Algorithm 3 Find Matches of Q^* over G^*

Input: a query signature graph Q^* with n vertices $v_i, i = 1, ..., n$
Input: G^*
Input: $C(v_i)$ that are candidate vertices that may match v_i.
Output: $M(Q^*)$: all matches of Q^* over G^*

```
1: Set Q' = φ
2: Select some vertex v_i, where C(v_i) is minimal among all
   vertices in Q*.
3: Q' = Q' ∪ v_i and M(Q') = C(v_i).
4: while Q' != Q* do
5:    for each backward edge e_i = →(v_i1, v_i2) that is adjacent
      to Q' do
6:       M(Q' ∪ e_i)=Backward(e_i, M(Q'))
7:       Q' = Q' ∪ e_i
8:    for each forward edge e_i = →(v_i1, v_i2) that is adjacent to
      Q' do
9:       M(Q' ∪ e_i)=Forward(e_i, M(Q'))
10:      Q' = Q' ∪ e_i
11: Set M(Q*)=M(Q')
12: return M(Q*)
Backward(e_i = →(v_i1, v_i2), M(Q'))
1: for each tuple t in M(Q') do
2:    If t cannot form a 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 = →(v_i1, v_i2), 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 C(v_i2) do
4:          if t ⋈ d is a 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 C(v_i1) do
9:          if d ⋈ t is a match of Q' ∪ e_i then
10:            Insert d ⋈ t into M(Q' ∪ e_i)
11: return M(Q' ∪ e_i)
```

in Algorithm 2. For each vertex v_i in query signature graph Q^*, we find the candidate list of vertices that can match v_i, denoted as $C(v_i)$. Specifically, based on Definition 8, given a vertex v_i in Q^*, a node d in ($\in G^I$) VS*-tree cannot match v_i, if and only if one of the following conditions hold:

1. $vSig(v_i)\&vSig(d) \neq vSig(v_i)$; or
2. $\exists v_j | \overrightarrow{v_i, v_j} \in Q^*, \forall d' | \overrightarrow{d, d'} \in G^I, vSig(v_j)\&vSig(d') \neq vSig(v_j)$.
3. $\exists v_j | \overrightarrow{v_i, v_j} \in Q^*, \forall d' | \overrightarrow{d, d'} \in G^I, Sig(\overrightarrow{v_i, v_j})\&Sig(\overrightarrow{d, d'}) \neq Sig(\overrightarrow{v_i, v_j})$.

Furthermore, if a node d ($\in G^I$) in VS*-tree cannot match v_i ($i = 1, \ldots, n$), its descendant nodes cannot match v_i. Consequently, the descendant nodes can be pruned safely.

Definition 12 Given a subgraph Q' of Q^*, an edge $e = \overrightarrow{v_1, v_2}$ in Q^* is called *adjacent* to Q' if and only if $(e \notin Q') \wedge (v_1 \in Q' \vee v_2 \in Q')$. □

Definition 13 Given an adjacent edge $e = \overrightarrow{v_1, v_2}$ 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.* □

Lines 6–10 of Algorithm 2 exploit the optimization we introduced (see end of Section 6) in VS*-tree where we divide vertices whose signatures contain too many "1"s, which reduces their discriminatory power. Specifically, first, for each query vertex v_i in Q^*, we set $C(v_i) = \{d_1^1\}$. Consider the level I summary graph G^I. For each query vertex v_i in Q^*, we consider each instance $v_{i[j]}$ of v_i. For each child node d^I of each element in $C(v_i)$, we determine whether $Sig(d^I)$ & $Sig(v_{i[j]}) = Sig(v_{i[j]})$. If so, we push d^I into $C(v_{i[j]})$. When we finish considering all instances of v_i, we update $C(v_i) = \bigcap_{j=1,...,m} C(v_{i[j]})$.

After finding candidates $C(v_i)$ for each query vertex u_i, we perform multiway join $C(v_1) \bowtie ... \bowtie C(v_n)$ to find matches of Q^* (Algorithm 3). Specifically, we apply a depth-first search strategy to find matches of Q^* over G^*, denoted as $M(Q^*)$. Initially, we set $Q' = \phi$, which denotes the structure of Q^* that has been visited so far. We start a DFS over G^* beginning with a vertex v_i where $C(v_i)$ is minimal among all vertices in Q^*. We insert v_i into query Q'. Now, the matches of Q', i.e., $M(Q')$, are updated as $M(Q') = C(v_i)$. For each edge e_i adjacent to Q', if e_i is a backward edge, we employ Backward function in Algorithm 3 to find matches of $Q' \cup e_i$, i.e., $M(Q' \cup e_i)$. Otherwise, we employ Forward function to find $M(Q' \cup e_i)$. 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 $Q' = Q^*$, and then $M(Q^*)$ is returned.

Finally, for each match of Q^* over G^*, we check whether it is a subgraph match of Q. If so, we insert it into answer set RS. According to these subgraph matches, it is straightforward to find the matching variable bindings for the SELECT variables in SPARQL. Experiments (see Part D of Online Supplements) demonstrate that VS* performs much better than BVS*.

8 T-index

In this section, we introduce T-index, which is a structured organization of materialized views. T-index is used to process aggregate SPARQL queries (Section 9).

For each entity vertex u in a RDF graph G, all *distinct* attribute properties (Definition 1) adjacent to u are collected to form a *transaction.* For example, the adjacent attribute properties of entity vertex 001 are "hasName, gender, bornOnDate, title, diedOnDate". Thus, we have a transaction $A(001)$ ="hasName, gender, bornOnDate, title, diedOnDate". All transactions are collected to form a transaction database DB, as shown in Figure 14a. Each entity vertex u in RDF graph G generates one transaction $A(u)$ in DB. In each transaction, properties (*dimensions*[1]) are ordered in their frequency descending order in DB, where property frequency is defined as follows.

Definition 14 The frequency of a property p in a transaction database DB is $Freq(p) = |\{A(u)|p \in A(u) \wedge A(u) \in DB\}|$. □

For example, the frequencies of "hasName", "Found Year", "gender", "bornOnDate", "title" and "diedOnDate" are 7, 4, 4, 3, 3 and 3, respectively. Therefore, "hasName" precedes "FoundYear", which precedes "gender" in the relevant transactions. This order is important since the construction of paths follows this order. Frequency ordering leads to fewer nodes being inserted since there is a higher probability that more prefixes will be shared among different transactions, and, therefore, minimizes the number of materialized views that need to be maintained. It also facilitates query processing, as discussed in Section 9. Note that, if multiple dimensions have the same frequency, their order is arbitrarily defined (the effect of this is experimentally studied).

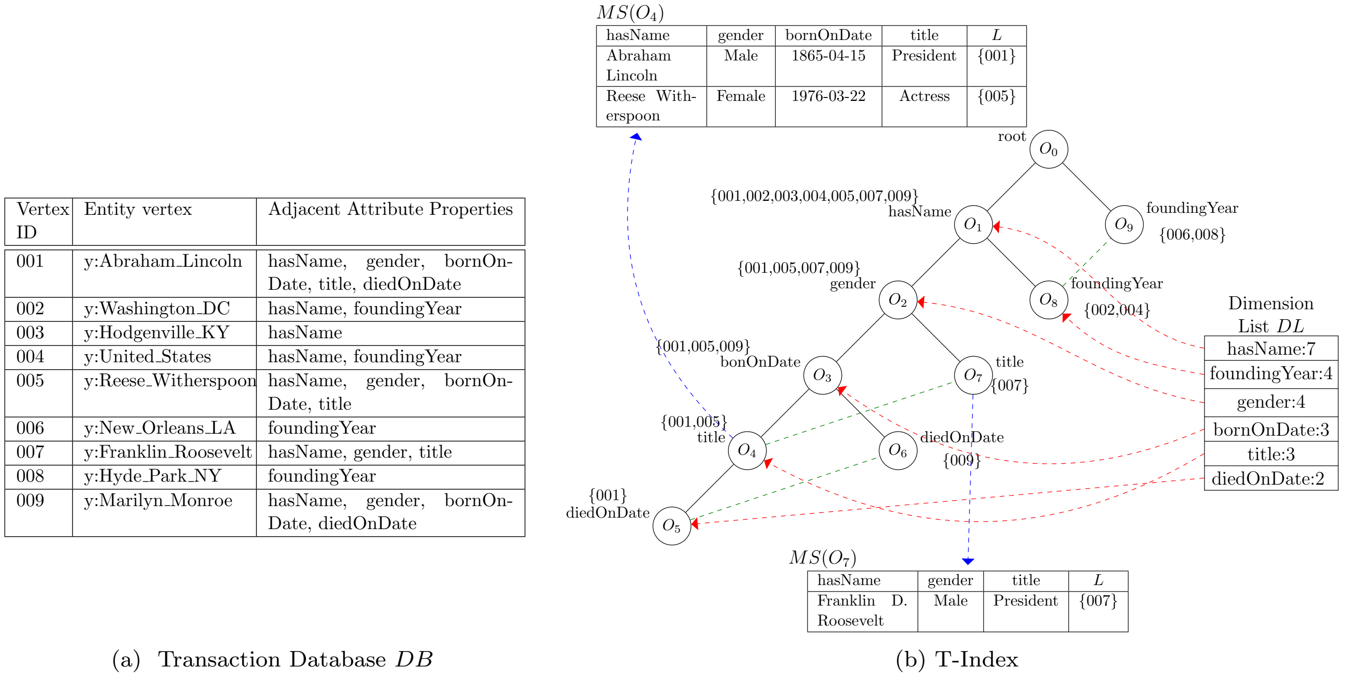
Definition 15 Given a transaction A, the length-n prefix of A is the first n dimensions in A. □

Definition 16 A T-index is a trie constructed as follows:

1. There is one root labeled as "root".
2. Each node O in T-index denotes a dimension.
3. Node O_j is a child of node O_i if and only if there exists at least one transaction A, where the path reaching node O_i is length-n prefix of A and the path reaching node O_j is length-$(n+1)$ prefix of A.
4. Each node O in T-index has a vertex list $O.L$ registering the IDs of all transactions A_i, where the path reaching O is a prefix of A_i. □

Figure 14b shows an example of T-index. When inserting $A(001)$ ="hasName, gender, bornOnDate, title, diedOnDate" into the T-index, the path "O_0-O_1-O_2-O_3-O_4-O_5" is followed. 001 is registered to $O_i.L$ where $i = 1, 2, 3, 4, 5$. Furthermore, since "hasName" is a prefix of seven transactions ("001,002,003,004,005,007,009"), the corresponding vertex list to node O_1 is $O_1.L = \{001, 002, 003, 004, 005, 007, 009\}$. Note that the storage

[1] The literature on aggregation and aggregate queries frequently refer to these attributes as *dimensions*. We follow the same convention.



Vertex ID	Entity vertex	Adjacent Attribute Properties
001	y:Abraham_Lincoln	hasName, gender, bornOnDate, title, diedOnDate
002	y:Washington_DC	hasName, foundingYear
003	y:Hodgenville_KY	hasName
004	y:United_States	hasName, foundingYear
005	y:Reese_Witherspoon	hasName, gender, bornOnDate, title
006	y:New_Orleans_LA	foundingYear
007	y:Franklin_Roosevelt	hasName, gender, title
008	y:Hyde_Park_NY	foundingYear
009	y:Marilyn_Monroe	hasName, gender, bornOnDate, diedOnDate

$MS(O_4)$

hasName	gender	bornOnDate	title	L
Abraham Lincoln	Male	1865-04-15	President	{001}
Reese Witherspoon	Female	1976-03-22	Actress	{005}

$MS(O_7)$

hasName	gender	title	L
Franklin D. Roosevelt	Male	President	{007}

(a) Transaction Database DB (b) T-Index

Fig. 14: T-index

of the vertex lists can be optimized by a variety of encoding techniques. We don't discuss this tangential issue any further.

In addition to T-index, there are two associated data structures: dimension list DL and materialized sets $MS(O)$. Dimension list DL records all dimensions in the transaction database. When introducing a node into T-index, according to the dimension of the node, we register the node to the corresponding dimension in DL, similar to building an inverted index. Consequently, the dimensions in DL are ordered in their frequency descending order.

Each node O has an aggregate set $MS(O)$. Let the dimensions along the path from the root to node O be $(l_1, l_2, ..., l_N)$. According to $O.L$, one can find all transactions represented by the portion of the path reaching this node. $MS(O)$ is a set of tuples that group these transactions based on shared values on dimensions $(l_1, l_2, ..., l_N)$. Each tuple t in $MS(O)$ has two parts: the dimensions $t.D$ and the vertex list $t.L$ that stores the vertex IDs in this aggregate tuple, as shown in Figure 14b. Consider node O_4 in Figure 14b. We partition all transactions represented by the path "O_0-O_1-O_2-O_3-O_4" into two groups based on group-by dimensions that share specified values along (hasName, gender, bornOnDate, title). These pre-computed sets speed up aggregate SPARQL query processing, as discussed in the next section.

Since T-index is a trie augmented with materialized views $MS(O)$ for each node O, we do not give its construction algorithm (for completeness, this is provided in Part C of Online Supplements). Two observations are important regarding T-index: (1) given a non-leaf node O in T-index that has n child nodes O_i, $(i = 1, ..., n)$, if the path reaching at least one O_i is a prefix of A, the path reaching node O is also a prefix of A; and (b) the structure of T-index does not depend on the order of inserting transactions into the structure.

We now discuss the materialization of $MS(O)$ of each node O in T-index. Obviously, for each node O in T-index, we can access all entity vertices in $O.L$ and their attribute properties to build $MS(O)$. However, some computation and I/O can be shared for computing $MS(O)$ of different nodes.

Given a node O with n child nodes O_i, $(i = 1, ..., n)$, if the path reaching at least one O_i is a prefix of one transaction A, the path reaching node O must also be a prefix of A. Thus, $O.L \supseteq \bigcup_i N_i.L$. Consequently, its aggregate set $MS(O)$ can be computed from the aggregate sets associated with its child nodes. Therefore, we propose a post-order traversal-based algorithm to materialize $MS(O)$ of the T-index in Algorithm 7. Assume that the properties along the path reaching node O are $(p_1, ..., p_m)$. Initially, $MS(O) = \phi$. For each child node O_i of O, we compute $MS'(O_i) = \prod_{(p_1, p_2, ..., p_n)} MS(O_i)$ (Lines 3-4 in Algorithm 7). Then, we compute $MS(O) =$

$\bigcup_i MS'(O_i)$ (Line 7). Furthermore, for each vertex v that is in $MS(O).L$ but not in $\bigcup_i MS(O_i).L$, we need to access the property values of vertex u on dimensions $p_1, ..., p_n$ in the RDF graph (Lines 6-12). Specifically, we define function $F(u) = \prod_{(p_1,...,p_n)} u$, which means projecting u's adjacent properties over $(p_1, ..., p_n)$ (Line 10). We insert $F(u)$ into $MS(O)$. If there exists some aggregate tuple t', where $t'.D = F(u)$, we register vertex ID of u in vertex list $t.L$ (Lines 8-9). Otherwise, we generate a new aggregate tuple t', where $t'.D = F'(u)$ and insert vertex ID of u into $t'.L$ (Lines 10-12).

Theorem 3 *Any entity vertex u in RDF graph G is accessed once in computing aggregate sets of trie-index by Algorithm 6.*

Proof See Part B of Online Supplementals.

We illustrate the construction of T-index using an example. First, a scan of DB (Figure 14a) derives a list of all dimensions in DB and their frequencies, and the dimension list DL is constructed. The root of T-index (O_0) is created and labeled as "root". Then, we insert all transactions of DB into T-index one-by-one.

1. The scan of the first transaction $A(001)$ leads to the construction of the first branch of the tree: (root, hasName,gender,bornOnDate,title, diedOnDate), inserting nodes O_1, O_2, O_3, O_4 and O_5. Initially, we set $O_i.L = \{001\}$, $i = 1, ..., 5$.
2. $A(002)$ shares a common prefix (hasName) with the existing path, and adds one new node O_8 (foundingYear) as a child of node O_1 (hasName). It also causes updating the corresponding vertex list of each node along the path.
3. The above process is iterated until all transactions are inserted into T-index.
4. Finally, for each node O_i in T-index, we build aggregate sets $MS(O_i)$ by post-order traversal over T-index. Specifically, we first compute $MS(O_5)$ by assessing entity vertices 001 and its dimension values. Then, we compute $MS(O_4)$ by merging the projection of $MS(O_5)$ over dimensions (hasName,gender, bornOnDate,title) and assessing entity vertex 005. The process is iterated until all aggregate sets are computed. $MS(O_4)$ and $MS(O_7)$ are given in Figure 14b as examples.

9 Aggregate Query Processing

As noted in Section 4, we decompose a GA query Q into several SA queries $\{S_i\}$, $i = 1, ..., n$, where each star center v_i is an entity vertex in Q. The result of each S_i ($R(S_i)$) is computed using the approach discussed in Section 9.1. We then join $R(S_i)$'s to compute the result of Q, i.e., $R(Q) = \bowtie_i R(S_i)$ as discussed in Section 9.2.

9.1 Star Aggregate Query Processing

Definition 17 A *Star Aggregate* (SA) query $(v, \{p_1, ... p_d\}, \{p_{d+1}, ...p_n\})$ consists of a central vertex v, a set of group-by dimensions $\{p_1, ...p_d\}$, and a set of measure dimensions $\{p_{d+1}, ...p_n\}$, where $\{p_1, ...p_d, ...p_n\}$ are all the attribute properties adjacent to v. □

Given a SA query $S = (v, \{p_1, ...p_d\}, \{p_{d+1}, ...p_n\})$, we answer S using T-index by Algorithm 4. Let $P = \{p_1, ..., p_d, ...p_n\}$. Given the set of properties P, we find their match $(O_1, ..., O_n)$, where O_i is a node in T-index and all nodes O_i $(i = 1, ..., n)$ are in the same path from the root, and the property associated with O_i equals p_i (Line 1 in Algorithm 4). We do not require that all nodes O_i (i=1,...,n) are adjacent to each other in the path. Thus, it is possible to have multiple matches for a given P. For each match $(O_1, ..., O_n)$, we use $\underline{O}$ to denote the farthest node from the root (Line 3 in Algorithm 4)[2]. The aggregate set associated with node $\underline{O}$ is denoted as $MS(\underline{O})$. We compute $MS'(\underline{O})$ by projecting $MS(\underline{O})$ over group-by dimensions $\{p_1, ..., p_d\}$ (i.e., $MS'(\underline{O}) = \prod_{(p_1,p_2,...,p_d)} MS(O)$). The $MS'(\underline{O})$ from all the matches are merged to form the final result to SA query S, i.e., $R(S)$ (Lines 6-7 in Algorithm 4).

Algorithm 4 SA Query Algorithm

Input: T-Index
Input: SA query$S(v, \{p_1, ..., p_d\}, \{p_{d+1}, ..., p_n\})$
Output: An aggregate result set MS for a SA query Q.
1: Locate all matches of properties $(p_1, ..., p_d, ..., p_n)$ in T-index
2: **for** each match m_i **do**
3: Let $\underline{O}_i$ denote the node in match m_i that is farthest from the root
4: Let $MS(\underline{O}_i)$ denote the aggregate set associated with node $\underline{O}_i$
5: $MS'(\underline{O}_i) = \prod_{(p_1,p_2,...,p_d)} MS(\underline{O}_i)$.
6: $MS = \bigcup_i MS'(\underline{O}_i)$
7: **return** MS

For example, given a SA query S_1 (v_1, {gender, title}, ϕ) in Figure 6, group-by dimensions are {gender, title} and measure dimension is ?m and we use "count" as an aggregate function (this is analogous to COUNT(*) in SQL). There are two matches (O_1, O_2, O_3, O_4) and (O_1, O_2, O_7) in T-index corresponding to two group-by dimensions. In the first match, O_4 is the farthest

[2] Note that, this is not necessarily O_n, since node identifiers are arbitrarily assigned to help with presentation.

node from the root. Since $MS(O_4)$ is an aggregate set over dimensions (hasName,gender, bornOnDate,title), we compute a temporary aggregate set $MS'(O_4)$ on dimensions (gender, title) by projecting $MS(O_4)$ over these two dimensions. In the second match, O_7 is the farthest node from the root. Since $MS(O_7)$ is an aggregate set over dimensions (hasName,gender,title), it is also projected over (gender,title) to get $MS(O'_7)$. Finally, we obtain $R(S_1)$ by merging $MS'(O_4)$ and $MS'(O_7)$. Figure 15 illustrates the process.

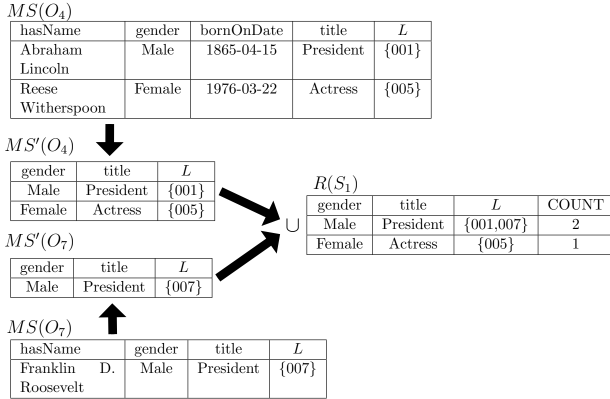


Fig. 15: Answering SA Query

Algorithm 5 General Aggregate (GA) Query Algorithm

Input: A GA query Q
Output: $R(Q)$ {Query Result}

1: Each entity vertex v_i (in Q), $i = 1, ..., n$, together with its adjacent attribute properties form a SA query S_i
2: **for** each SA query S_i **do**
3: Call Algorithm 4 to find $R(S_i)$, $i = 1, ..., n$.
4: $T_i = \bigcup_{t \in R(S_i)} t.L$
5: All entity vertices v_i together with link properties between them form the link structure J
6: Find all subgraph matches of J over RDF graph
7: $U = \{g_1\}$, where g_1 includes all subgraph matches
8: **for** each entity vertex v_i in J, $i = 1, ..., n$ **do**
9: set $U' = \phi$
10: **for** each group g in U **do**
11: **for** each aggregate tuple $t \in R(S_i)$ **do**
12: Select a group g' of matches $MS \in g$ $\{MS[i] \in t.L$ and $MS[i]$ refers to the i-th vertex in $MS\}$
13: Insert group g' into U'
14: Set $U = U'$
15: Assume that the measure dimension is associated with v_i
16: **for** each group g in U **do**
17: Find all matching vertices to v_i for all matches in g
18: Access the measure values of these matching vertices
19: Compute the aggregate value in measure dimension in this group, and insert it into $R(Q)$
20: **return** $R(Q)$

9.2 General Aggregate Query Processing

At this point, all $R(S_i)$ are computed. We now discuss how to compute the final result $R(Q) = \bowtie_i R(S_i)$.

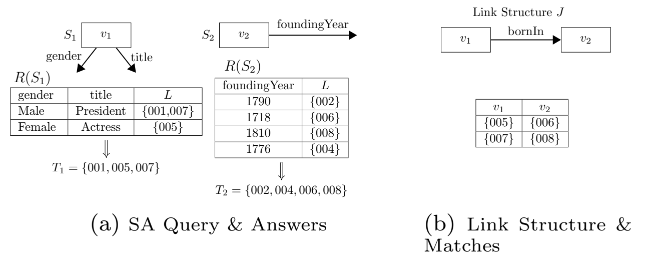
Definition 18 Let GA query Q consist of n SA queries. The *link structure* J of Q is a subgraph induced by all star centers v_i, $i = 1, ..., n$. Specifically, J is denoted as $J(V = \{v_i\}, E = \{e_j\}, \Sigma = \{eLabel(e_j)\})$, where vertex v_i $(1 \leq i \leq n)$ is a star center, e_j $(1 \leq j \leq m)$ is an edge whose endpoints are both star centers, and $eLabel(e_j)$ is the label (link property) of the edge e_j. □

Note that, J is a connected subgraph, since all entity vertices (in Q) are connected together by link properties. For each $R(S_i)$, we can find a vertex list L_i that includes all vertices in $R(S_i)$. Specifically, we get $T_i = \bigcup_{t \in R(S_i)} t.L$, where t is an aggregate tuple in $R(S_i)$. This means that all vertices in T_i are candidate matching vertices of v_i. To compute $R(Q)$, we need to find all subgraph matches of J over the RDF graph, where a subgraph match is defined as follows.

Definition 19 Given a link structure $J(V = \{v_i\}, E = \{e_j\}, \Sigma = \{eLabel(e_j)\})$ in Q and a subgraph $G'(V' = \{u_i\}, E' = \{e'_j\}, \Sigma' = \{eLabel(e'_j)\})$ in RDF graph G, where u_i is an entity vertex in G, e'_j is an edge whose endpoints are both in V', and $eLabel(e'_j)$ is the label (link property) of the edge e'_j, G' is called a subgraph match of J in RDF graph G if and only if:

1. $u_i \in T_i$
2. $e_j \in E \Leftrightarrow e'_j \in E'$ and $eLabel(e_j) = eLabel(e'_j)$ □

Consider query Q in Figure 6. The results of S_1 and S_2 and the T_i lists are shown in Figure 16a, while the link structure J is shown in Figure 16b.



(a) SA Query & Answers (b) Link Structure & Matches

Fig. 16: Query Decomposition

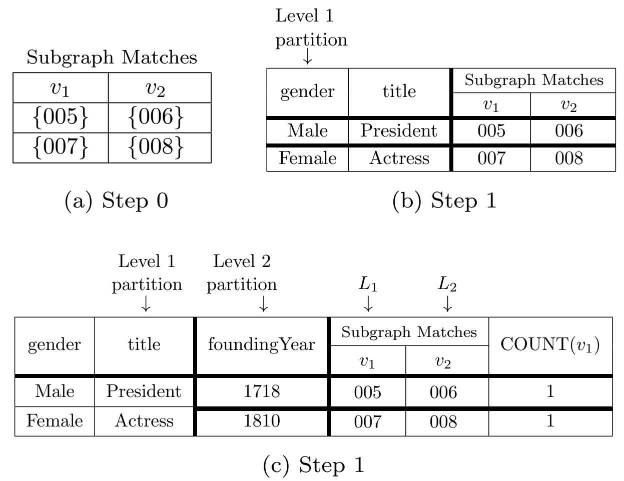
Obviously, we can perform multiway join $T_1 \bowtie ... \bowtie T_n$ to find matches of J over RDF graph G. In order to speed up online performance, T_i $(i = 1, ..., n)$ should be as small as possible. Note that, link structure J has a structure analogous to the query signature graph Q^*,

since each star center v_i in J corresponds to an entity vertex in SPARQL query graph Q and an edge in J is an edge between two entity vertices. Therefore, we can use the approach described in Section 7 to compute results. Specifically, given an aggregate query, we represent its query pattern graph as Q. Then, we encode Q into a signature query graph Q^*. For each vertex v_i in Q^*, we can find a candidate list $C(v_i)$ by employing Lines 1-15 in Algorithm 2. Then, we update $T_i = T_i \cap C(v_i)$. Finally, we utilize Algorithm 3 to find matches of J over RDF graph G.

For example, we find two subgraph matches of J over RDF graph G, where Figure 16b shows the flattened representation of these matches. Then, we need to partition these subgraph matches into one or more groups based on subgraph matches that share specified values in group-by dimensions, and create a new solution set $R(Q)$ that contains one tuple per aggregated group. Specifically, we try to get $U = \{g_j\}$, $j = 1, ..., m$, where all subgraph matches in group g_j share the same values over group-by dimensions.

A straightforward method is to find, for each match, the corresponding entity vertices and their group-by dimension values in the RDF graph, and partition these matches into different groups based on subgraph matches that share specified values in group-by dimensions. This method suffers from a large number of random I/Os. Furthermore, partitioning subgraph matches is an expensive task if there are a large number of subgraph matches of J over the RDF graph.

In order to improve performance, we use star aggregate query results $R(S_i)$ to partition subgraph matches, which helps reduce I/O accesses. Furthermore, scanning aggregate sets in T-index (in SA query algorithm) requires sequential access, which is much faster. More importantly, $R(S_i)$ has partitioned subgraph matches based on group-by dimensions associated with each vertex v_i in query Q. Therefore, we use $R(S_i)$ to find final partitions. Initially, we assume that all subgraph matches are in the same group g_1, and set $U = \{g_1\}$ (Line 7 in Algorithm 5). Then, we perform a multi-level partitioning over these subgraph matches. At the first level, we consider group-by dimensions in $R(S_1)$. For each group $g \in U$, we partition matches in g into some new groups g'_i, $i = 1, ..., m$, where each new group g'_i has matches that share the same values over group-by dimensions in $R(S_1)$. Obviously, $g = \bigcup_i g'_i$. Specifically, in order to partition matches in group g, we sequentially scan $R(S_1)$. For each aggregate tuple $t \in R(S_1)$, we find a new group g'_i of subgraph matches MS, such that $MS[i] \in t.L$ and $MS[i]$ refers to the ith vertex in subgraph match MS (Lines 11-12). We insert these new groups g'_i into U' (Line 13). We repeat the above process (Lines 10-14) for all groups g in U. Then, U' is the first-level partition. Obviously, $\bigcup_{g \in U}\{\text{all matches in group } g\} = \bigcup_{g' \in U'}\{\text{all matches in group } g'\}$. Iteratively, we consider other $R(S_i)$ for other level partitions (Lines 8-14). Finally, for each aggregate group, we compute the aggregate value in aggregated dimension, and insert it into final result $R(Q)$. Assume that the measure dimension is associated with vertex u_i in Q. For each group g, we find a list of distinct vertices matching u_i. We access the measure dimension values of these matching vertices and compute the aggregate value for this group.



Subgraph Matches

v_1	v_2
{005}	{006}
{007}	{008}

(a) Step 0

Level 1 partition ↓ (gender)

gender	title	Subgraph Matches	
		v_1	v_2
Male	President	005	006
Female	Actress	007	008

(b) Step 1

Level 1 partition ↓ (title); Level 2 partition ↓ (foundingYear); L_1 ↓ (v_1); L_2 ↓ (v_2)

gender	title	foundingYear	Subgraph Matches		COUNT(v_1)
			v_1	v_2	
Male	President	1718	005	006	1
Female	Actress	1810	007	008	1

(c) Step 1

Fig. 17: Partitioning Subgraph Matches

Let us recall the decomposition of Q_3 given in Figure 6. We discuss how to partition all matches of J into different groups based on group-by dimensions to find the final result $R(Q)$. Initially, all matches are in the same group g, i.e., $g = \{(005, 006), (007, 008)\}$ and $U_0 = \{g\}$, where (005, 006) is a flattened representation of a match. Based on $R(S_1)$, we can get the first-level partition U_1. Specifically, we partition matches of g into fine-grained groups. According to the first aggregate tuple t_1 in $R(S_1)$ (Figure 16a), we create a new group $g_1 = \{(005, 006)\}$, since $005 \in t_1.L$. Similarly, we create another group $g_2 = \{(007, 008)\}$. Consequently, the first level partition is $U_1 = \{g_1, g_2\}$, shown in Figure 17b. Then, based on $R(S_2)$, we can perform the second-level partition U_2. Specifically, we partition each group g_i ($i = 1, 2$) into more fine-grained groups. Figure 17c shows the second-level partition. Finally, we compute the aggregate value for each group in U_2.

9.3 Aggregation over Link Properties

So far, we assumed that group-by dimensions always correspond to attribute property edges. In this subsection, we discuss how to relax this assumption to allow

group-by dimensions that correspond to link property edges. Consider the following aggregate query Q_5.

```
SELECT ?d ?c COUNT(?m) WHERE
{?m <bornIn> ?c. ?m <bornOnDate> ?d.
 ?m <hasName> ?n.}
 GROUP BY ?d ?c
```

Note that, in Q_5, one of the group-by dimensions is $?c$, corresponding to "BornIn", which is a link property edge. Our solution to processing this type of aggregate query is the following.

For each entity vertex u in the RDF graph, we introduce a special dimension (i.e., attribute property) "hasURI", and set its dimension value to be URI of this entity. The introduction of "hasURI" changes the entries in the transaction database DB in Figure 14a by adding a new dimension "hasURI" to each transaction. Following the method in Section 8, we build the T-index and compute aggregate sets. Note that, although "hasURI" has the largest frequency, we set it to be the last element in dimension list DL such that it always corresponds to leaf nodes in T-index. We make this special arrangement for the following reason. Assume that "hasURI" corresponds to some non-leaf node O in the T-index. Consider any descendant node O' of O (including O itself). We know that aggregate set $MS(O')$ must have dimension "hasURI". Since each entity has a distinct URI, for each aggregate tuple $t \in MS(O')$, $t.L$ has only one entity. In that case, $MS(O')$ may have many rows, which increases the space cost. Therefore, we set "hasURI" to be the last one in DL to reduce the index size. We omit the details about building the T-index and computing the aggregate sets in the updated transaction database D after introducing "hasURI", since it is similar to the method described in Section 8.

Given an aggregate query Q, if Q has a triple $\langle ?\mathrm{m}_i$, p_i, $?\mathrm{o}_i\rangle$, where $?\mathrm{o}_i$ is a group-by dimension and p_i is a link property, we rewrite the aggregate query by inserting a new triple ($?\mathrm{o}_i$ hasURI $?\mathrm{o}'_i$) and replacing $?\mathrm{o}_i$ by $?\mathrm{o}'_i$ as the group-by dimension. For example, Q_4 has a triple (?m <bornIn> ?c), where "bornIn" is a link property and ?c is a group-by dimension. Thus, we rewrite Q_5 into Q'_5 as follows.

```
SELECT ?d ?c COUNT(?m) WHERE
{?m <BornIn> ?c. ?m <BornOnDate> ?d.
 ?m <hasName> ?n. ?c <hasURI> ?c'}
 GROUP BY ?d ?c'
```

Obviously, Q'_5 is a GA query (see Section 9.2) and all group-by dimensions are now attribute property edges, which can be answered by Algorithm 5.

10 Handling Data Updates

As mentioned earlier, most existing RDF triple stores cannot support updates effectively. In this section, we address this problem for gStore by discussing how the index structures are maintained. Obviously, the updates over the adjacency list table are very straightforward. The main challenge is the maintenance of G^*, VS*-tree and T-index. We discuss these issues in this section.

10.1 Updates to Signature Graph G^*

There are two cases to consider: inserting one triple and deleting one triple.

Inserting one triple. Assume that a new triple $\langle s, p, o\rangle$ is inserted into RDF dataset. If s existed in G^* before insertion, we delete vertex s and its all adjacent edges from G^*. We employ the method discussed in Section 10.2.3 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 the method discussed in Section 10.2.1 to insert s into VS*-tree. If o is also an entity or a class vertex, we follow an analogous method.

Deleting one triple. Assume that triple $\langle s, p, o\rangle$ is deleted from the RDF dataset. s must correspond to an entity vertex or a class vertex. If o is a literal, we only need to re-code s. The structure of G^* does not change. We also delete s from VS*-tree and insert the re-coded s into the tree. If o is an entity or class vertex, we first need to re-code s. Then, we delete the edge between s and o in G^*. We also delete s from VS*-tree and insert the re-coded s into the tree according to the updated G^*'s structure.

10.2 Updates to VS*-Tree

10.2.1 Insertion

The insertion process of a vertex u (in G^*) begins at the root of VS*-tree and iteratively chooses a child node until it reaches a suitable leaf node location where u is inserted. 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 the leaf summary graph have changed, this must be propagated upwards within the VS*-tree. The main challenge is the choice of a child node. The choice in the S-tree depends on the Hamming distance between the signatures of u and the nodes in the tree. In VS*-tree, we modify the insertion rule so that the location

depends on both node signatures and G^*'s structure. We prefer to select a node so that the updates of both vertex signatures and the number of newly introduced super edges (in VS*-tree) are minimized.

Specifically, given a vertex u and a non-leaf node d that has n children d_1,...,d_n, the distance between u and d_i $(i = 1, ..., n)$ is 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 (bit-string), 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, 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 edges adjacent to u in G^*. Experiments show that VS*-tree has significantly better pruning properties.

10.2.2 Split

Like other height balanced trees, insertion into a node that is already full will invoke a node split. 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 adopt a node splitting method similar to [27]. The fundamental issue is to determine the two new nodes from among the $B+1$ entities. For this, we need to consider each pair of $B+1$ entities. At each iteration, we select as the seed nodes two entities from among the $B+1$. We allocate the remaining entities to these two nodes according to Equation 1. We perform this for each pair of $B+1$ entities and keep the entity pair that leads to the minimal value of Max(α,β), where α and β are the number of 1's in the two nodes.

After a node splits, we have to update the signatures and the super edges associated with the two new nodes. The updates are 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.

10.2.3 Deletion

To delete a vertex u from VS*-tree, we find the leaf node d where u is stored, and delete u. This effects the nodes along the path from the root to d. 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, then d is deleted and its entries are reinserted into VS*-tree.

The VS*-tree nodes have two parameters B and b (see Definition 9). The root has at least two and at most B children, where other nodes have at least b and at most B children. Unlike B$^+$-tree, $b = \frac{B}{2}$, here, $1 \leq b \leq \frac{B}{2}$ [28]. $b = 1$ is the "free-at-empty" strategy, i.e., containing as few as one child [17], while $b = \frac{B}{2}$ is the "merge-at-half" strategy. We experimentally compare the two strategies in Section 11.

10.3 Updates to T-index

As mentioned earlier, all dimensions are ordered in descending frequency order in dimension list DL to improve SA query evaluation performance. However, RDF data updates may change the order of dimensions in DL requiring special care during index maintenance. Therefore, we consider updates of T-index in two cases based on whether or not the order of dimensions in DL changes. We give a high-level outline of our techniques, and provide detailed explanations and examples in Part C of Online Supplements.

10.3.1 Dimension List DL's order does not change

Consider the insertion of a new triple $\langle s, p, o \rangle$. If p is a link property, we do not need to update the T-index. Thus, we only consider the case when p is an attribute property, i.e., dimension.

If s does not occur in the existing RDF data, we introduce a new transaction into D. We insert the new transaction into one path of T-index following Definition 16, and update the affected materialized aggregate sets.

If s is already in the existing RDF graph, assume that s's existing dimensions are $\{p_1, ..., p_n\}$ and $Freq(p_i) > Freq(p) > Freq(p_{i+1})$ in dimension list DL. Again, in the case of equality, order is chosen arbitrarily. This means that the new inserted dimension p should be inserted between p_i and p_{i+1}. We locate two nodes O_i and O_n, where the path reaching node O_i (and O_n) has dimensions $(p_1, ..., p_i)$ (and $(p_1, ..., p_i, ...p_n)$[3]). We remove subject s from all materialized sets along the path between nodes O_{i+1} and O_n, where O_{i+1} is a child node of O_i. Then, we insert dimensions $(p, p_{i+1}, ..., p_n)$ into T-index from node O_i, and update the materialized sets along the path.

Consider the deletion of a triple $\langle s, p, o \rangle$, where p is an attribute property as discussed above. Assume

[3] Although dimension list is a set, when the order is important, we specify them as a list enclosed in ( ).

that s's existing dimensions are $\{p_1, ..., p_n\}$ and $p = p_i$. We proceed as above to delete s from all affected materialized sets.

10.3.2 Dimension List DL's order changes

Some triple deletions and insertions that affect dimension frequency will lead to changing the order of dimensions in DL. Assume that two dimensions p_i and p_j need to be swapped in DL due to inserting or deleting one triple $\langle s, p, o \rangle$. Obviously, $j = i + 1$, i.e., p_i and p_j are adjacent to each other in DL.

Updates are handled in two phases. First, we ignore the order change in DL and handle the updates using the method in Section 10.3.1. Second, we swap p_i and p_j in DL, and change the structure of T-index and the relevant materialized sets.

11 Experiments

We evaluate gStore over real RDF and synthetic datasets, and compare it with SW-store [1], RDF-3x [19], x-RDF-3x [22], a graph-based solution GRIN [29], and two commercial systems: BigOWLIM (www.ontotext.com/owlim/big/) and Virtuoso 6.3 (virtuoso.openlinksw.com/). The results of the main experiments are reported here; more results are given in Part D of Online Supplements.

11.1 Datasets

We use three real RDF and one benchmark datasets in our experiments.

1. Yago (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.
   For exact query evaluation, we use all SPARQL queries in [19] over Yago. For wildcard query evaluation, we rewrite all of these queries to include 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 six aggregate queries. All sample queries are given in Part A of Online Supplements.
2. In order to evaluate the scalability of gStore, we also use Yago2 [15] in our experiments. It has has more than 10 million entities and more than 180 million triples. The total size of Yago2 is larger than 23 GB.
3. DBLP (sw.deri.org/∼aharth/2004/07/dblp/) contains a large number of bibliographic descriptions. There are about 8 million triples consuming 0.8GB. We define six sample SPARQL queries, as shown in the Online Supplements.
4. LUBM is a synthetic benchmark [12] that adopts an ontology for the university domain, and can generate synthetic OWL data scalable to any arbitrary size. We vary the university number from 100 to 1000. The number of triples is 13 to 133 million. The total RDF file size is 1.5 to 15.1 GB.

11.2 Parameter Settings

Our coding methods and indexing structures use a number of parameters. In this subsection, we discuss how to set up these parameters to optimize query processing.

11.2.1 M and m

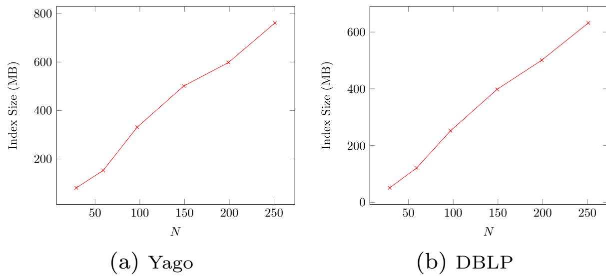
Given a vertex u in the RDF graph, we encode each edge label ($eLabel$) adjacent to u into a bitstring $eSig(e).e$ with length M, and set m out of M bits to be '1'. We obtain $vSig(u).e$ by performing bitwise OR over all $eSig(e).e$. Analogous to signature files, there may exist the "false drop" problem [9]. Let $AdjEdges(u, G)$ denote all edge labels adjacent to u in G and $AdjEdges(v, Q)$ defined similarly. If $AdjEdges(v, Q) \not\subset AdjEdges(u, G) \wedge vSig(v) \& vSig(u) = vSig(v)$, we say that a *false drop* has occurred. $vSig(v) \& vSig(u) = v$ 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 [18], the probability of false drops can be quantified by the following equation.

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

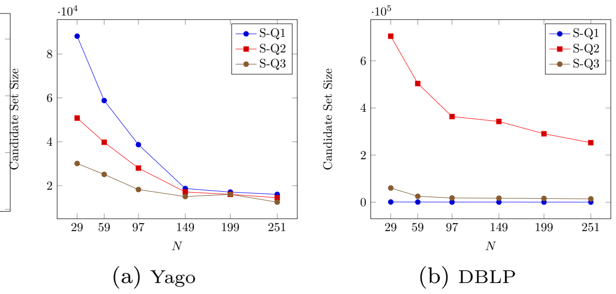
where $|AdjEdges(u, G)|$ is u's degree in G and $|AdjEdges(v, Q)|$ is v's degree in Q and M is the length of bitstring, and m out of M bits are set to be '1' in hash functions. According to the optimization method in Section 6, a high degree vertex will be decomposed into several instances and each instance's degree is no larger than δ. Thus, the false drop probability P_{false_drop} increases with $|AdjEdges(u, G)|$. Since $|AdjEdges(u, G)| \leq \delta$, $P_{false_drop} \leq (1 - e^{-\frac{\delta * m}{M}})^{m*|AdjEdges(v,Q)|}$. To improve query performance, false drop probability should be as small as possible, as the false drop will lead to unnecessary I/O. For example if we wish this probability to be no larger than 1.0×10^{-3}, the parameter setting can be tuned to ensure the following holds:

$$P_{false_drop} \leq (1 - e^{-\frac{\delta * m}{M}})^{m*|AdjEdges(v,Q)|} \leq 1.0 \times 10^{-3}$$



(a) Yago (b) DBLP

Fig. 18: VS*-tree Index Size vs. N



(a) Yago (b) DBLP

Fig. 19: Candidate Size vs. $|N|$

From the query logs, we determine that the average value of $|AdjEdges(v, Q)|$ is 3. Therefore, in Yago and DBLP datasets, we set $m = 2$,$M = 97$ and $\delta = 10$. In this case, $P_{false_drop} \leq 1.0 \times 10^{-3}$.

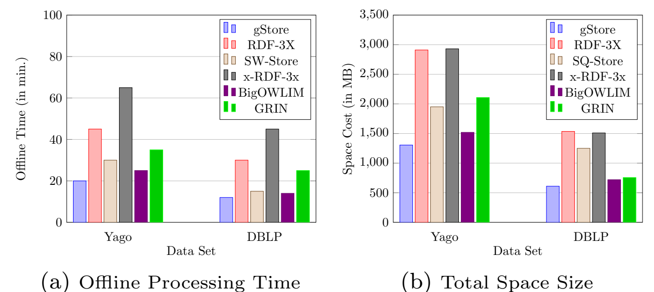
11.2.2 N and n

The false drop problem also exists in comparing $vSig(q).n$ with $vSig(v).n$, but it is quite difficult to quantify the probability of false drops in this case. Therefore, we adopt the following method, using the "n-gram" technique where $n = 3$ that has been experimentally shown to work well [11].

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 as N increases as shown in Figure 19. 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 experiments, we set $N = 149$ in Yago and $N = 97$ in DBLP, respectively, as larger N's values cannot lead to significant increase in pruning power.

11.3 Database Load Performance

We compare gStore with five competitors over both Yago and DBLP datasets in terms of database load time and index size. For a fair comparison, we adopt the settings in [19], i.e., each dataset is first converted into a factorized form: one file with RDF triples represented as integer triples, and one dictionary file to map from ids to literals. All methods utilize the same input files and load them into their own systems. 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 20a and 20b, respectively. Since our method adopts the bitstring as the basic index structures, compared with other approaches, gStore has the minimal total space size, as shown in Figure 20b. Figure 20a shows that our index building time is the fastest. In other methods, they need to build several exhaustive indexes, such as RDF-3x and x-RDF-3x. The index structure in GRIN is a memory-based data structure that is not optimized for I/O cost. In order to handle large graphs, GRIN has to employ virtual memory for index construction, which leads to I/O cost.



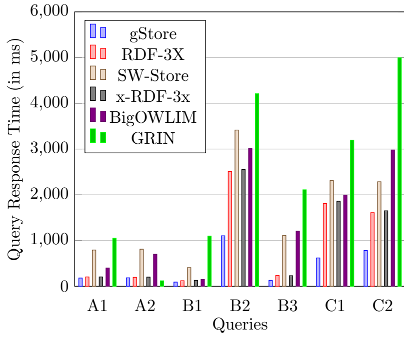
(a) Offline Processing Time (b) Total Space Size

Fig. 20: Evaluating Offline Performance

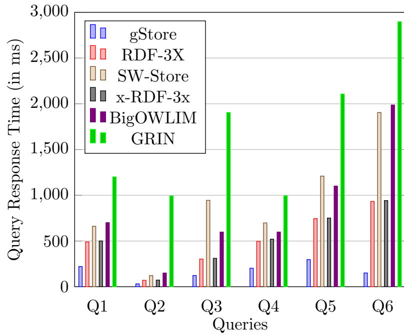
11.4 Query Performance

11.4.1 Exact Queries

We compare the performance of gStore with five competitors over both Yago and DBLP datasets. The sample queries are shown in Tables 6 and 7 in the Online Supplements. Figure 21 shows that gStore system is much faster than other methods. The key reason is that

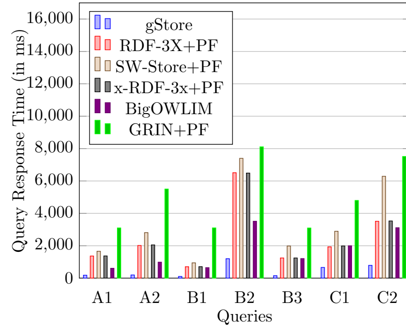


(a) Yago

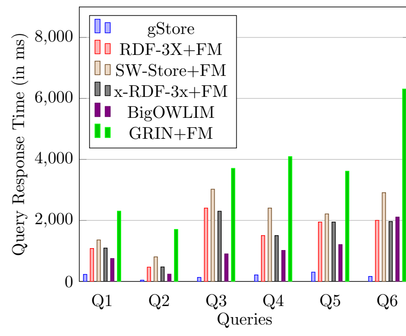


(b) DBLP

Fig. 21: Exact Query Response Time



(a) Yago



(b) DBLP

Fig. 22: Wildcard Query Response Time (FM: postfiltering)

candidate lists for each query vertex are reduced greatly by VS*-tree. We show the pruning power of VS*-tree in Figures 26 and 28b (in Online Supplements). The other ssytems do not consider the query graph structure for pruning.

11.4.2 Wildcard Queries

In order to enable comparison over wildcard queries, we adopt the post-filtering method in Section 11.1 in RDF-3x, SW-store, x-RDF-3x and GRIN. Since BigOWLIM has embedded full-text index, it can support wildcard queries. The sample queries are shown in Tables 9 and 10 in the Online Supplements. Figure 22 shows query response times of different methods. As would be expected, all of the tested systems suffer some amount of performance drop in executing wildcard queries. However gStore experiences the least amount of drop, because gStore can answer both exact and wildcard queries in a uniform manner, making its performance more robust. In all but one query, even with the performance drop, it executes wildcard queries in under one second. However, the query performance degrades dramatically for other systems, since they cannot support wildcard queries directly. The only way to process these queries is to first ignore wildcard constraints, and, for each returned result, check if it satisfies the wildcard constraints. The worst performance drop for each system are 929% for RDF-3x, 560% for SW-Store, 920% for x-RDF-3x, 330% for BigOWLIM, and 4486% for GRIN.

11.5 Aggregate Query Performance

We compared the performance of gStore with Virtuoso 6.3 (http://virtuoso.open linksw.com/) and the CAA algorithm [16] on both SA and GA queries. To the best of our knowledge, Virtuoso is the only commercial system that can fully support group-by and aggregation following SPARQL 1.1 specification, and CAA algorithm is the only proposal in literature that can handle aggregate queries that follows SPARQL 1.1 semantics. We use Yago dataset in this experiment. Six aggregate queries that are used in our experiments are given in Table 12 in the Online Supplements. Figure 23 shows that our method has the best online performance.

Generally speaking, the query performance depends on several factors, such as aggregation ratio $\frac{|R(Q')|}{|R(Q)|}$, the size of $|R(Q')|$, the size of query graph Q and the characteristics of group-by dimensions (whether or not they include link properties). In Figure 23, both SA1 and SA2 have one group-by dimension. SA1 (52 milliseconds) is much faster than SA2 (163 milliseconds). Furthermore, the speedup ratios (between gStore and

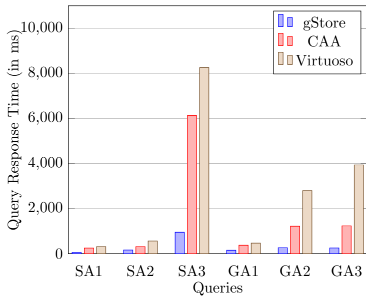


Fig. 23: Aggregation Queries in Yago

Virtuoso) for SA1 (6.1) is larger than that for SA2 (3.5). This is because $|R(Q')|$ is smaller in SA1 than that in SA2. The speedup ratio of SA3 (8.68) is the largest among all three SA queries in Yago, since SA3 has two edges in the query graph, which requires Virtuoso to perform expensive join operations. We find that SA3 is even slower than some GA queries. That is because "hasFamilyName" and "hasGivenName" have the largest frequency in Yago, which results in large $|R(Q')|$. The results also show that gStore-Aggregation method also performs better than other alternative approaches on processing GA queries. Our method uses VS*-tree to find matches of the link structure J, which uses structural pruning to reduce the search space.

11.6 Evaluating Dimension Orders in DL

We evaluate the affect of the order of dimensions DL using both Yago and LUBM datasets. Specifically, we build T-index under three different orders: frequency-descending order, frequency-ascending order and random order. The results show that frequency-descending order leads to the least number of nodes in T-index, the minimal total size and the minimal building time, since this order ensures that more prefixes can be shared among different transactions. Furthermore, the frequency-descending order also brings the fastest query response time, since it accesses the minimal number of nodes in T-index. Complete experimental results can be found in Part D of Online Supplements.

11.7 Evaluating Maintenance Cost

To evaluate the overhead of VS*-tree and T-index maintenance, we randomly selected 80% of the triples in each dataset to build VS*-tree and T-index. Then, for the remaining 20% of the triples, we performed a sequence of triple insertions using the method in Section 10. For

Table 2: Evaluating Updates to VS*-tree in Yago (number of updated triples/second)

	gStore	RDF-3x	x-RDF-3x
Insertion	2.2×10^4	1.4×10^4	1.2×10^4
Deletion (free-at-empty)	2.0×10^4	1.1×10^4	0.9×10^4
Deletion (merge-at-half)	1.5×10^4	1.1×10^4	0.9×10^4

evaluating deletion performance, we randomly deleted 20% of the triples from VS*-tree and T-index.

We report the number of triple insertions/deletions per second in gStore, and compare it with RDF-3x and x-RDF-3x in Table 2. RDF-3X had been extended for updates using the deferred-indexing approach [21] where the updates are first recorded into differential indexes, which are periodically merged into main indexes. x-RDF-3x employs the similar update strategy except for introducing "timestamp" of each triple [22]. Table 2 shows that gStore's update time is the fastest, bacause VS*-tree is a height-balance tree. The insertion/deletion operation has $\log|V(G)|$'s time complexity. RDF-3x and x-RDF-3x need to update six clustered B^+-trees. We also compare two deletion strategies ($b = 1$ and $b = \frac{B}{2}$) in Table 2. Experiments show that the "free-at-empty" ($b = 1$) leads to faster deletion time, since "merge-at-half" strategy ($b = \frac{B}{2}$) needs to re-insert the children of the half-empty nodes, while it is not necessary in the free-at-empty strategy.

The average running times to insert/delete one triple (without changing T-index's structure) into/from T-index and update the corresponding aggregate sets are 0.02 and 0.01 msec, respectively. The average running time to swap two adjacent dimensions is 3000 msec. The total running time to insert/delete 10,000 triples, which include both inserting and dimension swapping times, are 15.20 and 12.20 msec, respectively. The number of dimension swaps for insertion/deletion are 5 and 4, respectively. As these show, T-index's maintenance overhead is light. When we insert/delete a triple without changing T-index's structure, we only need to insert one tuple into some aggregate sets along one path. The operation is very cheap. If we need to swap two dimensions p_i and p_j, and there are n paths containing both p_i and p_j, there are at most $2 \times n$ aggregate sets that need to be updated. Thus, the time for swapping two dimensions is higher. However, the chances of swapping two dimensions during the insertion/deletion is small.

11.8 Scalability Experiments

We evaluate the scalability of gStore against RDF-3x and Virtuoso, using two large datasets: LUBM and Yago2.

We first report gStore's index building performance over various LUBM dataset sizes in Table 3.

For online query performance over LUBM, we use the set of 7 queries that have been defined and used in recent studies [3, 34]. This workload was designed because of the recognition that the original workload of 14 queries are very simple involving few triple patterns (many involve two triple patterns) and are similar to each other. Furthermore, a large number of them involve constants that artificially increase their selectivity – for example, triple patterns ?x undergraduateDegreeFrom http://www.University0.edu and ?x publicationAuthor http://www.Department0.University0.edu/AssistantProfessor0 identify a single object making their selectivity very high (with very few results). It has been argued that these queries are not representative of real workloads that contain implicit joins, and that they favor systems that use extensive indexing (such as RDF-3x). We ran experiments with both query sets[4], but we report the results obtained for the 7 queries that are given in Table 14. The conclusions reported below hold for the original 14 query set as well.

Table 4 shows comparative online query performance of gStore, RDF-3x and Virtuoso under small and very large LUBM configurations. gStore always performs (sometimes an order of magnitude) better than Virtuoso, with the latter sometimes failing to complete within the 30 minute window we allocated for executing queries. As expected, and as noted above, RDF-3x does better than gStore when the query contains a triple pattern has high selectivity because it refers to a constant. Even for these queries, gStore still performs very well with sub-second response time. For other queries (i.e., those that involve implicit joins), gStore's performance is superior to RDF-3x.

We also performed cross-query set evaluation to test the performance of systems according to the order of triple pattern evaluation. For example, query Q2 in the original LUBM workload (Table 15) is identical to Q1 we report (Table 14) except for the different triple pattern orders. For these queries, RDF-3x's performance varies by 29–119 times, while gStore performance varies by 3 times. Naturally, both systems would benefit from optimizing join orders, but gStore performance appears to be more stable.

We also compare the three systems over the Yago2 dataset (query sets in Tables 8 and 11). The results in Table 5 demonstrate that gStore is faster than Virtuoso by orders of magnitude in exact queries, and outperforms Virtuoso greatly in wildcard queries. This is because gStore utilizes the same (signature-based) pruning strategy for wildcard queries, while Virtuoso adopts the post-processing technique to handle them. Similar to LUBM results, the relative performance of gStore and RDF-3x depend on the selectivity of the triple patterns that include constants for which extensive indexing can be exploited. RDF-3x does not support wildcard queries, thus, we have not compared the two systems for those queries.

[4] We revised the original 14 to remove type reasoning that gStore does not currently support; the resulting queries return larger result sets since there is no filtering as a result of type reasoning. For completeness, these are included in Online Supplements as Table 15.

Table 3: Scalability of Index Building

Data Set	Raw Data			Offline Processing Time	Index Size (KB)
	RDF N3 File Size(KB)	Number of Triples	Number of Entities		
LUBM-100	1,499,131	13,402,474	2,178,865	12m23s	852,448
LUBM-200	3,007,385	26,691,773	4,340,588	32m11s	1,190,856
LUBM-500	7,551,065	66,720,02	10,846,157	1h5m9s	1,640,281
LUBM-1000	15,136,798	133,553,834	21,715,108	2h10m15s	3,071,205
Yago	3,126,721	19,012,854	4,339,591	20m35s	1,305,267
Yago2	23,216,336	183,183,314	10,557,352	8h55m21s	3,071,205

Table 4: Scalability of Query Performance on LUBM

Queries	Query Response Time (msec)					
	LUBM-100			LUBM-1000		
	gStore	RDF-3x	Virtuoso	gStore	RDF-3x	Virtuoso
Q1	1310	22634	2152	43832	202728	54460
Q2	236	4276	30560	1563	15008	> 30 mins
Q3	208	216	1981	1491	1737	7784
Q4*	153	44	1341	680	30	1357
Q5*	129	48	312	131	12	243
Q6*	227	20	843	828	49	842
Q7	1016	960	5257	8301	14072	36848

$*$ indicates the query contains at least one constant entity.

Table 5: Scalability of Query Performance on Yago2

	Query Response Time (msec)						
Exact Queries	A1*	A2*	B1*	B2	B3	C1	C2*
gStore	251	230	2157	131	198	875	865
RDF-3x	35	26	921	289	228	219077	80
Virtuoso	1544	3213	23447	2777	6240	151337	23275
Wildcard Queries	AW1*	AW2*	BW1*	BW2	BW3	CW1	CW2*
gStore	3226	6122	8644	268	3197	15183	6189
Virtuoso	3338	10109	33728	2388	21482	>30min	69031

$*$ indicates the query contains at least one constant entity.

12 Conclusions

In this paper, we described gStore, which is a graph-based triple store. Our focus is on the algorithms and data structures that were developed to answer SPARQL queries efficiently. The class of queries that gStore can handle include exact, wildcard, and aggregate queries. The performance experiments demonstrate that compared to two other state-of-the-art systems that we consider, gStore has more robust performance across all these query types. Other systems either do not support

some of these query types (e.g., none of the systems support wildcard queries and only Virtuoso supports aggregate queries) or they perform considerably worse (e.g., Virtuoso handles exact and aggregate queries, but is consistently worse than gStore). The techniques can handle dynamic RDF repositories that may be updated. gStore is a fully implemented and operational system.

There are many directions that we intend to follow. These include support for partitioned RDF repositories, parallel execution of SPARQL queries, and further query optimization techniques.

13 Acknowledgements

Lei Zou's work was supported by National Science Foundation of China (NSFC) under Grant 61370055 and by CCF-Tencent Open Research Fund. M. Tamer zsu's work was supported by Natural Sciences and Engineering Research Council (NSERC) of Canada under a Discovery Grant. Lei Chen's work was supported in part by the Hong Kong RGC Project M-HKUST602/12, National Grand Fundamental Research 973 Program of China under Grant 2012-CB316200, Microsoft Research Asia Grant, and a Google Faculty Award. Dongyan Zhao was supported by NSFC under Grant 61272344 and China 863 Project under Grant No. 2012AA011101.

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 *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 411–422, 2007.
3. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In *Proc. 19th Int. World Wide Web Conf.*, pages 41–50, 2010.
4. P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
5. V. Bönström, A. Hinze, and H. Schweppe. Storing RDF as a graph. In *Proc. 1st Latin American Web Congress*, pages 27–36, 2003.
6. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proc. 1st Int. Semantic Web Conf.*, pages 54–68, 2002.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2001.
8. U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *Proc. 9th Int. ACM SIGIR Conf. on Research and Dev. in Inf. Retr.*, pages 77–87, 1986.
9. 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.
10. 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.
11. L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *Proc. 12th Int. World Wide Web Conf.*, pages 90–101, 2003.
12. Y. Guo, Z. Pan, and J. Heflin. LUBM: a benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
13. A. Gupta, , and V. H. Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21st Int. Conf. on Very Large Data Bases*, pages 358–369, 1995.
14. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *Proc. 6th Int. Semantic Web Conf.*, pages 211–224, 2007.
15. J. Hoffart, F. M. Suchanek, K. Berberich, E. L. Kelham, G. de Melo, and G. Weikum. YAGO2: Exploring and querying world knowledge in time, space, context, and many languages. In *Proc. 20th Int. World Wide Web Conf.*, pages 229–232, 2011.
16. E. Hung, Y. Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *Proc. 21st Int. Conf. on Data Eng.*, pages 717–728, 2005.
17. T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *J. Comput. Syst. Sci.*, 47(1):45–76, 1993.
18. H. Kitagawa and Y. Ishikawa. False drop analysis of set retrieval with signature files. *IEICE Trans. on Inf. and Syst.*, E80-D(6):1–12, 1997.
19. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.
20. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 627–640, 2009.
21. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
22. T. Neumann and G. Weikum. X-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.*, 1(1):256–263, 2010.
23. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.
24. D. Y. Seid and S. Mehrotra. Grouping and aggregate queries over semantic web databases. In *Proc. Int. Conf. on Semantic Computing*, pages 775–782, 2007.
25. D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. 21st ACM Symp. on Principles of Database Systems*, pages 39–52, 2002.
26. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proc. 17th Int. World Wide Web Conf.*, pages 595–604, 2008.
27. E. Tousidou, P. Bozanis, and Y. Manolopoulos. Signature-based structures for objects with set-valued attributes. *Inf. Syst.*, 27(2):93–121, 2002.
28. E. Tousidou, A. Nanopoulos, and Y. Manolopoulos. Improved methods for signature-tree construction. *Comput. J.*, 43(4):301–314, 2000.
29. O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *Proc. 22nd National Conf. on Artificial Intelligence*, pages 1465–1470, 2007.

30. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, 2008.
31. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. 1st Int. Workshop on Semantic Web and Databases*, pages 131–150, 2003.
32. X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 335–346, 2004.
33. Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in RDF triple stores. In *Proc. 25th Int. Conf. on Data Eng.*, pages 1263–1266, 2009.
34. P. Yuan, P. Liu, H. Jin, W. Zhang, and L. Liu. TripleBit: A fast and compact system for large scale RDF data. *Proc. VLDB Endow.*, 6(7):517–528, 2013.

Online Supplements

A. Queries Used in Experiments

Table 6: Exact SPARQL Queries in Yago

A1	SELECT ?gn ?fn WHERE {?p y:hasGivenName ?gn.?p rdf:type <wordnet_scientist_110560637>.?p y:bornIn ?city.?city y:locatedIn <Switzerland>.?p y:hasAcademicAdvisor ?a.?a y:bornIn ?city2.?city2 y:locatedIn <Germany> }	a oriented fact query, i.e., finding scientists from Switzerland with a doctoral advisor from Germany
A2	SELECT ?name WHERE { ?a rdfs:label ?name .?a rdf:type wordnet_actor_109765278.?a y:livesIn ?city .?city y:locatedIn ?state .?state y:locatedIn "New York" .?a y:actedIn ?m1 .?m1 rdf:type "wordnet_movie_106613686".?m1 y:hasProductionLanguage "English language".?a y:directed ?m2 .?m2 rdf:type "wordnet_movie_106613686".?m2 y:hasProductionLanguage "English language" }	a oriented fact query, i.e., finding a person who is both an actor and a director of English movies.
B1	SELECT distinct ?name1 ?name2 WHERE { ?a1 rdfs:label ?name1. ?a2 rdfs:label ?name2.?a1 y:livesIn ?city1.?a2 y:livesIn ?city2.?city1 y:locatedIn "England".?city2 y:locatedIn "England".?a1 y:actedIn ?movie.?a2 y:actedIn ?movie.FILTER (?a1 != ?a2) }	a relationship oriented query, finding two actors from England playing together in the same movie.
B2	SELECT ?name1 ?name2 WHERE { ?p1 rdfs:label ?name1 .?p2 rdfs:label ?name2 .?p1 y:isMarriedTo ?p2 .?p1 y:bornIn ?city .?p2 y:bornIn ?city. }	a relationship oriented query, finding a couple who was born in the same city.
B3	SELECT distinct ?name1 ?name2 WHERE {?p1 y:hasFamilyName?name1.?p2 y:hasFamilyName ?name2.?p1 rdf:type "wordnet_scientist_110560637".?p2 rdf:type "wordnet_scientist_110560637". ?p1 y:hasWonPrize ?award. ?p2 y:hasWonPrize ?award.?p1 y:bornIn ?city. ?p2 y:bornIn ?city. FILTER (?p1 != ?p2) }	a relationship oriented query, finding two scientists who born in the same place and won the same prize.
C1	SELECT distinct ?name1 ?name2 WHERE { ?p1 y:hasFamilyName?name1.?p2 y:hasFamilyName ?name2.?p1 rdf:type "wordnet_scientist_110560637".?p2 rdf:type "wordnet_scientist_110560637".?p1 [] ?city.?p2 [] ?city.?city rdf:type <wordnet_site_108651247>.FILTER (?p1 != ?p2) }	a relationship oriented query with unknown predicates, finding two scientists related to the same city.
C2	SELECT distinct ?name WHERE { ?p rdfs:label ?name.?p [] ?c1.?p [] ?c2.?c1 rdf:type <wordnet_village_108672738>.?c2 rdf:type <wordnet_site_108651247>.?c1 rdfs:label "London".?c2 rdfs:label "Paris". }	a relationship oriented query with unknown predicates, finding a person who is related to both London and Pairs.

Table 7: Exact SPARQL Queries in DBLP

Q1	SELECT ?x1 WHERE {?x1 foaf:name "Jiawei Han".?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier "VLDB".?x2 rdf:type :Inproceedings}	A query with two star subqueries, finding Jiawei Han's VLDB papers.
Q2	SELECT ?x2 ?x3 WHERE {?x1 foaf:name "Jiawei Han".?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier "VLDB".?x2 rdf:type : Inproceedings.?x3 dc:creator ?x1.?x3 dc:identifier "ICDE".?x3 rdf:type : Inproceedings }	A query with three star subqueries, finding Jiawei Han's VLDB and ICDE papers.
Q3	SELECT ?x1 ?x2 WHERE { ?x1 foaf:name "Jiawei Han".?x2 :year "1999".?x2 dc:creator ?x1}	A query with two star subqueries, finding Jiawei Han's papers in 1999.
Q4	SELECT ?x1 WHERE {?x1 foaf:name "Jiawei Han".?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier "ICDE".?x2 :year "2001"}	A query with two star subqueries, finding Jiawei Han's ICDE papers in 2001.
Q5	SELECT ?x1 ?x2 WHERE {?x1 foaf:name "Jiawei Han".?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 booktitle "CIKM" }	A query with two star subqueries, finding Jiawei Han's CIKM papers.
Q6	SELECT ?x2x3 WHERE {?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier "ICDE".?x2 rdf:type : Inproceedings.?x3 dc:creator ?x1.?x3 dc:identifier "SIGMOD".}	A query with three stars, finding some persons who have both ICDE and SIGMOD papers.

Table 8: Exact SPARQL Queries in Yago2

A1	SELECT ?gn ?fn WHERE { ?p hasGivenName ?gn. ?p hasFamilyName ?fn . ?p type_star <wordnet_scientist_110560637>. ?p wasBornIn ?city . ?p hasAcademicAdvisor ?a . ?city isLocatedIn <Switzerland>. ?a *wasBornIn* ?city2 . ?city2 isLocatedIn_transitive <Germany> . }	a oriented fact query, i.e., finding scientists from Switzerland with a doctoral advisor from Germany
A2	SELECT ?n WHERE { ?a hasPreferredName ?n . ?a type wordnet_actor_109765278. ?a livesIn ?city . ?a actedIn ?m1 . ?a directed ?m2 . ?city isLocatedIn ?state . ?state isLocatedIn <Africa> . ?m1 type_star <wordnet_movie_106613686> . ?m2 type_star <wordnet_movie_106613686>. }	a oriented fact query, i.e., finding a person who is both an actor and a director of English movies.
B1	SELECT distinct ?n1 ?n2 WHERE { ?a1 hasPreferredName ?n1 . ?a2 hasPreferredName ?n2 . ?a1 wasBornIn ?city1. ?a2 wasBornIn ?city2 . ?city1 isLocatedIn_transitive <United_States> . ?city2 isLocatedIn_transitive <United_States> . ?a1 actedIn ?m . ?a2 actedIn ?m . FILTER (?a1 != ?a2) }	a relationship oriented query, finding two actors from England playing together in the same movie.
B2	SPARQL SELECT ?n1 ?n2 WHERE { ?a1 hasPreferredName ?n1 . ?a2 hasPreferredName ?n2 . ?a1 isMarriedTo ?a2 . ?a1 wasBornIn ?city . ?a2 wasBornIn ?city .}	a relationship oriented query, finding a couple who was born in the same city.
B3	SPARQL SELECT distinct ?n1 ?n2 WHERE { ?a1 hasFamilyName ?n1 .?a2 hasFamilyName ?n2 . ?a1 type_star <wordnet_scientist_110560637> . ?a2 type_star¿ <wordnet_scientist_110560637> . ?a1 hasWonPrize ?award . ?a2 hasWonPrize ?award . ?a1 wasBornIn ?city . ?a2 wasBornIn ?city . FILTER (?a1 != ?a2) }	a relationship oriented query, finding two scientists who born in the same place and won the same prize.
C1	SELECT distinct ?n1 ?n2 WHERE { ?a1 hasFamilyName ?n1 . ?a2 hasFamilyName ?n2 . ?a1 type_star <wordnet_scientist_110560637> . ?a2 type_star <wordnet_scientist_110560637> . ?a1 ?p1 ?city . ?a2 ?p2 ?city . ?city type_star <wordnet_site_108651247> . FILTER (?a1 != ?a2) }	a relationship oriented query with unknown predicates, finding two scientists related to the same city.
C2	SPARQL SELECT distinct ?n WHERE { ?p hasPreferredName ?n . ?p ?p1 ?city1 . ?p ?p2 ?city2 . ?city1 type_star <wordnet_site_108651247 . ?city1 type_star <wordnet_site_108651247 . ?city1 hasPreferredName"$London''.?city2hasPreferredName$ "Paris" . }	a relationship oriented query with unknown predicates, finding a person who is related to both London and Pairs.

Table 9: SPARQL Queries with Wildcards in Yago

A1	SELECT ?gn ?fn WHERE {?p y:hasGivenName ?gn.?p rdf:type ?a.?p y:bornIn ?city.?city y:locatedIn ?b.?p y:hasAcademicAdvisor ?a.?a y:bornIn ?city2.?city2 y:locatedIn ?c. FILTER regex(str(?a),"scientist").regex(str(?b),"Switzerland"). regex(str(?b),"Germany")}	a oriented fact query with wildcards, i.e., finding scientists from Switzerland with a doctoral advisor from Germany
A2	SELECT ?name WHERE { ?a rdfs:label ?name .?a rdf:type "wordnet_actor 109765278".?a y:livesIn ?city .?city y:locatedIn ?state .?state y:locatedIn "New York" .?a y:actedIn ?m1 .?m1 rdf:type "wordnet_movie_106613686".?m1 y:hasProductionLanguage ?a.?a y:directed ?m2 .?m2 rdf:type "wordnet_movie_106613686".?m2 y:hasProductionLanguage ?b.FILTER regex(str(?a),"English").regex(str(?b),"English")}	a oriented fact query with wildcards, i.e., finding a person who is both an actor and a director of English movies.
B1	SELECT distinct ?name1 ?name2 WHERE { ?a1 rdfs:label?name1.?a2 rdfs:label ?name2.?a1 y:livesIn ?city1.?a2 y:livesIn ?city2.?city1 y:locatedIn ?a.?city2 y:locatedIn ?b.?a1 y:actedIn ?movie.?a2 y:actedIn ?movie.FILTER (?a1 != ?a2.regex(str(?a),"England").regex(str(?b),"England")) }	a relationship oriented query with wildcards, finding two actors from England playing together in the same movie.
B2	SELECT ?name1 ?name2 WHERE { ?p1 rdfs:label ?name1 .?p2 rdfs:label ?name2 .?p1 y:isMarriedTo ?p2 .?p1 y:bornIn ?city .?p2 y:bornIn ?city.}	a relationship oriented query with wildcards, finding a couple who was born in the same city.
B3	SELECT distinct ?name1 ?name2 WHERE { ?p1 y:hasFamilyName?name1.?p2 y:hasFamilyName ?name2.?p1 rdf:type ?a.?p2 rdf:type ?b.?p1 y:hasWonPrize ?award.?p2 y:hasWonPrize ?award.?p1 y:bornIn ?city.?p2 y:bornIn ?city.FILTER (?p1 != ?p2,regex(str(?a),"scientist").regex(str(?b),"scientist")) }	a relationship oriented query with wildcards, finding two scientists who born in the same place and won the same prize.
C1	SELECT distinct ?name1 ?name2 WHERE { ?p1 y:hasFamilyName?name1.?p2 y:hasFamilyName ?name2.?p1 rdf:type ?a.?p2 rdf:type ?b.?p1 [] ?city.?p2 [] ?city.?city rdf:type ?c.FILTER (?p1 != ?p2,regex(str(?a),"scientist").regex(str(?b),"scientist"). regex(str(?c),"site")) }	a relationship oriented query with unknown predicates and wildcards, finding two scientists related to the same city.
C2	SELECT distinct ?name WHERE { ?p rdfs:label ?name.?p [] ?c1.?p [] ?c2.?c1 rdf:type <wordnet village 108672738>.?c2 rdf:type <wordnet site 108651247>.?c1 rdfs:label ?a.?c2 rdfs:label ?b.FILTER (regex(str(?a),"London").regex(str(?b),"Paris")) }	a relationship oriented query with unknown predicates and wildcards, finding a person who is related to both London and Pairs.

B. Proofs of Theorems

In order to make it easier to follow, we duplicate the theorem statements before giving their proofs.

Theorem 1 *$RS \subseteq CL$ holds.*

Proof According to Definition 3, given a query graph Q with n vertices $\{v_1, ..., v_n\}$, $\{u_1, ..., u_n\}$ is a subgraph match of Q over RDF graph G. Based on the match, we define a function F, where $F(v_i) = u_i$, where $i = 1, ..., n$. Since there are only entity and class vertices in data signature graph G^* and query signature graph Q^*, we only consider the entity and class vertices. We delete all literal vertices from query graph Q to get $\{v_1, ..., v_n\}$ to get $\{v'_1, ..., v'_m\}$. Obviously, v'_i $(i = 1, ..., m)$ is a vertex in query signature graph Q^*. We only need to prove that $(F(v'_1), ..., F(v'_m))$ is a match of $\{v'_1, ..., v'_m\}$ according to Definition 8. Since $u'_i (= F(v'_i)$, $i = 1, ..., m)$ is a matching vertex of v_i in the RDF graph G, all

Table 10: SPARQL Queries with Wildcards in DBLP

Q1	SELECT ?x1 WHERE {?x1 foaf:name ?a.?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier ?b.?x2 rdf:type :Inproceedings. }. FILTER (regex(str(?a),"Jiawei").regex(str(?b),"VLDB")) }	A wildcard query with two star subqueries, finding Jiawei Han's VLDB papers.
Q2	SELECT ?x2, x3 WHERE {?x1 foaf:name ?a.?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier ?b.?x2 rdf:type : Inproceedings.?x3 dc:creator ?x1.?x3 dc:identifier ?c.?x3 rdf:type : Inproceedings. FILTER (regex(str(?a),"Jiawei").regex(str(?b),"VLDB"). regex(str(?c),"ICDE"))}	A query with three star subqueries, finding Jiawei Han's VLDB and ICDE papers.
Q3	SELECT ?x1, ?x2 WHERE { ?x1 foaf:name ?a.?x2 :year ?b.?x2 dc:creator ?x1. FILTER (regex(str(?a),"Jiawei").regex(str(?b),"1999"))}	A query with two star subqueries, finding Jiawei Han's papers in 1999.
Q4	Q3SELECT ?x1 WHERE { ?x1 foaf:name ?a.?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier ?b.?x2 :year ?c. FILTER (regex(str(?a),"Jiawei"). regex(str(?b),"ICDE"). regex(str(?c),"2001")) }	A query with two star subqueries, finding Jiawei Han's ICDE papers in 2001.
Q5	SELECT ?x1 ?x2 WHERE { ?x1 foaf:name ?a.?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 booktitle ?b. FILTER (regex(str(?a),"Jiawei").regex(str(?b),"CIKM")) }	A query with two star subqueries, finding Jiawei Han's CIKM papers.
Q6	SELECT ?x2, x3 WHERE { ?x1 rdf:type foaf:Person.?x2 dc:creator ?x1.?x2 dc:identifier ?a.?x2 rdf:type : Inproceedings.?x3 dc:creator ?x1.?x3 dc:identifier ?b.?x3 :year ?c. FILTER (regex(str(?a),"ICDE").regex(str(?b),"SIGMOD"). regex(str(?c),2000) }	A query with three stars, finding some persons who have both ICDE and SIGMOD papers.

Table 11: SPARQL Queries with Wildcards in Yago2

AW1	SELECT ?gn ?fn WHERE { ?p hasGivenName ?gn. ?p hasFamilyName ?fn . ?p type_star ?w1. ?p wasBornIn ?city . ?p hasAcademicAdvisor ?a . ?city isLocatedIn ?w2. ?a *wasBornIn* ?city2 . ?city2 isLocatedIn_transitive ?w3 . FILTER (regex(?w1,"scientist") && regex(?w2, "Switzerland") && regex(?w3,"Germany")) }	a oriented fact query, i.e., finding scientists from Switzerland with a doctoral advisor from Germany
AW2	SELECT ?n WHERE { ?a hasPreferredName ?n . ?a type ?w1. ?a livesIn ?city . ?a actedIn ?m1 . ?a directed ?m2 . ?city isLocatedIn ?state . ?state isLocatedIn ?w2 . ?m1 type_star ?w3 . ?m2 type_star ?w3. FILTER (REGEX(?w1,"actor") && REGEX(?w2,"Africa") &®EX(?w3,"movie")) }	a oriented fact query, i.e., finding a person who is both an actor and a director of English movies.
BW1	SELECT distinct ?n1 ?n2 WHERE { ?a1 hasPreferredName ?n1 . ?a2 hasPreferredName ?n2 . ?a1 wasBornIn ?city1. ?a2 wasBornIn ?city2 . ?city1 isLocatedIn_transitive ?w1 . ?city2 isLocatedIn_transitive ?w2 . ?a1 actedIn ?m . ?a2 actedIn ?m . FILTER ((?a1 != ?a2) && REGEX(?w1,"United") && REGEX(?w2,"States")) }	a relationship oriented query, finding two actors from England playing together in the same movie.
BW2	SPARQL SELECT ?n1 ?n2 WHERE { ?a1 hasPreferredName ?n1 . ?a2 hasPreferredName ?n2 . ?a1 isMarriedTo ?a2 . ?a1 wasBornIn ?city . ?a2 wasBornIn ?city .}	a relationship oriented query, finding a couple who was born in the same city.
BW3	SPARQL SELECT distinct ?n1 ?n2 WHERE { ?a1 hasFamilyName ?n1 .?a2 hasFamilyName ?n2 . ?a1 type_star ?w1 . ?a2 type_star ?w2 . ?a1 hasWonPrize ?award . ?a2 hasWonPrize ?award . ?a1 wasBornIn ?city . ?a2 wasBornIn ?city . FILTER ((?a1 != ?a2) && REGEX(?w1,"scientist") && REGEX(?w2,"scientist")) }	a relationship oriented query, finding two scientists who born in the same place and won the same prize.
CW1	SELECT distinct ?n1 ?n2 WHERE { ?a1 hasFamilyName ?n1 . ?a2 hasFamilyName ?n2 . ?a1 type_star ?w1. ?a2 type_star ?w2. ?a1 ?p1 ?city . ?a2 ?p2 ?city . ?city type_star ?w3 . FILTER ((?a1 != ?a2) && REGEX(?w1,"scientist") && REGEX(?w2,"scientist") && REGEX(?w3,"site")) }	a relationship oriented query with unknown predicates, finding two scientists related to the same city.
CW2	SPARQL SELECT distinct ?n WHERE { ?p hasPreferredName ?n . ?p ?p1 ?city1 . ?p ?p2 ?city2 . ?city1 type_star ?w1 . ?city1 type_star ?w1. ?city1 hasPreferredName?$w2.?city2hasPreferredName$?w3. FILTER (REGEX(?w1,"site") && REGEX(?w2,"London") && REGEX(?w3,"Paris")) }	a relationship oriented query with unknown predicates, finding a person who is related to both London and Pairs.

Table 12: Aggregation SPARQL Queries in Yago

SA1	SELECT ?u COUNT(?m) WHERE {?m <hasUTCOffset> ?u } GROUP BY ?u	high aggregation ratio, but the cardinality of the ratio of the solution set to non-aggregation query is small. Group all places by their UTC offsets.
SA2	SELECT ?b COUNT(?m) WHERE {?m <bornOnDate> ?b } GROUP BY ?b	high aggregation ratio, and the cardinality of the ratio of the solution set to non-aggregation query is large. Group all individuals by their birth years.
SA3	SELECT ?f ?g COUNT(?m) WHERE {?m <hasFamilyName> ?f ?m <hasGivenName> ?g } GROUP BY ?f, ?g	low aggregation ratio, and the cardinality of the ratio of the solution set to non-aggregation query. Group all persons by their family names and given names.
GA1	SELECT ?b ?u AVG(?p) WHERE { ?m <livesIn> ?c. ?m <bornOnDate> ?b. ?c <hasUTCOffset > ?u } GROUP BY ?b, ?u	join two SA queries that have high aggregation ratio
GA2	SELECT ?f ?d COUNT(?b) WHERE {?m <hasFamilyName> ?f. ?m <created> ?b. ?b <createdOnDate> ?d. } GROUP BY ?f, ?d	join two SA queries one with high aggregation ratio, the other one with low aggregation ratio
GA3	SELECT ?f ?b COUNT(?m) WHERE {?m <hasFamilyName> ?f. ?n <hasFamilyName> ? b. ?m <hasAcademicAdvisor > ?b } GROUP BY ?f , ?b	join two SA queries, S_1, S_2, $\lvert S'_1 \rvert$ and $\lvert S'_2 \rvert$ (which are non-aggregation versions of the respective queries) are large, but $\lvert GA3' \rvert$ is small

neighbors of v_i also have the matching vertices in G. Thus, according to the encoding strategy in Definition 6, we know that $vSig(u'_i) \& vSig(v'_i) = vSig(v'_i)$. Thus,

Table 13: Aggregation SPARQL Queries in LUBM

	Sample Queries	
SA1	SELECT ?p COUNT(?m) WHERE {?m <rdf:type> ?p.} GROUP BY ?p	high aggregation ratio, and the cardinality of the solution set to non-aggregation query is large. Group all entities by their types.
SA2	SELECT ?t COUNT(?m) WHERE {?m <ub:name> ?n. ?m <ub:researchInterest> ?t.} GROUP BY ?t	low aggregation ratio, but the cardinality of the solution set to non-aggregation query is large. Group all persons by their research interests.
SA3	SELECT ?p ?a COUNT(?m) WHERE {?m <rdf:type> ?p. ?m <ub:publicationAuthor> ?a.} GROUP BY ?f ?g	A group-by dimension is a link property (?a) Group by all publications by their types and the authors.
GA1	SELECT ?s ?d COUNT(?x) WHERE { ?x <rdf:type> ?s. ?y <rdf:type> ?d. ?x <ub:memberOf> ?y } GROUP BY ?s ?d	join two SA queries that have high aggregation ratios.
GA2	SELECT ?t ?z COUNT(?x) WHERE {?x <rdf:type> ?t. ?x <ub:worksFor> ?y. ?y <ub:subOrganizationOf> ?z.} GROUP BY ?t ? z	join two SA queries. One has high aggregation ratio and the other one has low aggregation ratio
GA3	SELECT ?x ?y COUNT(?z) WHERE {?x <rdf:type> ub:Student. ?y <rdf:type> ub:Faculty. ?z <rdf:type> ub:Course. ?x <ub:advisor> ?y. ?y <ub:teacherOf> ?z. ?x <ub:takesCourse> ?z.} GROUP BY ?x ?y	The join structure in GA3 is a triangle.

Table 14: SPARQL Queries in LUBM

Q1	SELECT ?x ?y ?z WHERE { ?z ub:subOrganizationOf ?y . ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?x rdf:type ub:GraduateStudent . ?x ub:undergraduateDegreeFrom ?y . }	This query has large input and low selectivity. Furthermore, it is a triangular pattern of relationships between the objects.
Q2	SELECT ?x WHERE { ?x rdf:type ub:Course . ?x ub:name ?y . }	This query has large input and low selectivity.
Q3	SELECT ?x ?y ?z WHERE { ?x rdf:type ub:Undergraduate-Student. ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?z ub:subOrganizationOf ?y . ?x ub:undergraduateDegreeFrom ?y . }	The query is the same with S-Q1 except that the student is an undergraduate student. The output size is 0.
Q4	SELECT ?x WHERE { ?x ub:worksFor <http://www.-Department0.University0.edu> . ?x rdf:type ub:FullProfessor . ?x ub:name ?y1 . ?x ub:emailAddress ?y2 . ?x ub:telephone ?y3 . }	This query has a star-style query and it has a constant entity.
Q5	SELECT ?x WHERE { ?x ub:subOrganizationOf <http://www.Department0.University0.edu . ?x rdf:type ub:Research- Group}	This query has a star-style query and it has a constant entity.
Q6	SELECT ?x ?y WHERE { ?y ub:subOrganizationOf <http://www.University0.edu> . ?y rdf:type ub:Department . ?x ub:worksFor ?y . ?x rdf:type ub:FullProfessor . }	This query join two stars and one star has a constant entity.
Q7	SELECT ?x ?y ?z WHERE { ?y ub:teacherOf ?z . ?y rdf:type ub:FullProfessor . ?z rdf:type ub:Course . ?x ub:advisor ?y . ?x rdf:type ub:UndergraduateStudent . ?x ub:takesCourse ?z }	This is a complex query, since it contains a lot of join steps

Table 15: Original SPARQL Queries in LUBM (Type inferencing removed)

Q1	SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent. ?x ub:takesCourse http://www.Department0.University0.edu/GraduateCourse0 }	This query has large input and high selectivity. Finding all students who take some course.
Q2	SELECT ?x ?y WHERE { ?x rdf:type ub:GraduateStudent. ?x ub:undergraduateDegreeFrom ?y. ?x ub:memberOf ?Z. ?y rdf:type ub:University. ?Z rdf:type ub:Department. ?Z ub:subOrganizationOf ?y. }	This query has large input and high selectivity. Furthermore, it is a triangular pattern of relationships between the objects. Finding all students who take some courses that are hosted by their department.
Q3	SELECT ?x WHERE { ?x rdf:type ub:Publication. ?x ub:publicationAuthor http://www.Department0.University0.edu/AssistantProfessor0 }	This query has large input and high selectivity. Finding all papers that are published by some assistant professor.
Q4	SELECT ?x ?y1 ?y3 WHERE { ?x ub:worksFor http://www.Department0.University0.edu. ?x ub:name ?y1. ?x ub:emailAddress ?y2. ?x ub:telephone ?y3. }	This query has small input and high selectivity. Finding the emails and telephone numbers of employee who worked for some department.
Q5	SELECT ?x WHERE { ?x ub:memberOf http://www.Department0.University0.edu. }	It is a simple query. Finding all members of some department. Very high selectivity
Q6	SELECT ?x WHERE { ?x rdf:type ub:UndergraduateStudent. }	This query has a large number of output results. Finding all undergraduate students.
Q7	SELECT ?x ?y WHERE {?x rdf:type ub:UndergraduateStudent. ?x ub:takesCourse ?y. ?y rdf:type ub:Course. }	This query has a large number of output results. Finding all undergraduate students and the courses they take.
Q8	SELECT ?x ?y WHERE { ?x rdf:type ub:UndergraduateStudent. ?x ub:memberOf ?y. ?x ub:emailAddress ?Z. ?y rdf:type ub:Department. ?y ub:subOrganizationOf http://www.University0.edu }	This query is more complex than Query 7 by including one more property. Finding all undergraduate students of some university. High selectivity.
Q9	SELECT ?x ?y WHERE { ?x rdf:type ub:UndergraduateStudent. ?x ub:advisor ?y. ?x ub:takesCourse ?Z. ?y ub:teacherOf ?Z. ?Z rdf:type ub:Course }	This query has a triangle pattern. Finding all undergraduate students who take his advisor's course.
Q10	SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent. ?x ub:takesCourse http://www.Department0.University0.edu/GraduateCourse0 }	This is a simple query. Finding all students who take some course.
Q11	SELECT ?x ?y WHERE { ?x rdf:type ub:ResearchGroup. ?x ub:subOrganizationOf ?y. ?y ub:subOrganizationOf http://www.University0.edu. }	This is a star query. Finding all organizations of some university.
Q12	SELECT ?x ?y WHERE { ?x ub:headOf ?y. ?y rdf:type ub:Department. ?y ub:subOrganizationOf http://www.University0.edu. }	Input of this query is small. Finding the department heads of some university.
Q13	SELECT ?x WHERE { ?x ub:undergraduateDegreeFrom http://www.University0.edu }	This query has a large number of output results. Finding all students graduate from some university.
Q14	SELECT ?x WHERE { ?x rdf:type ub:UndergraduateStudent }	This query represents those with large input and low selectivity. Finding all undergraduate students

$(F(v'_1), ..., F(v'_m))$ forms a subgraph match of $\{v'_1, ..., v'_m\}$ in a data signature graph G^*. □

Theorem 2 *Given a query signature graph Q^* with n vertices $\{v_1, \ldots, v_n\}$, a data signature graph G^* and VS*-tree built over G^*:*

1. *Assume that n vertices $\{u_1, \ldots, u_n\}$ forms a match (Definition 8) of Q^* over G^*. Given a summary graph G^I in VS*-tree, let u_i's ancestor in G^I be node d_i^I. $(d_1^I, ..., d_n^I)$ must form a summary match (Definition 10) of Q^* over G^I.*
2. *If there exists no summary match of Q^* over G^I, there exists no match of Q^* over G^*.*

Proof (1) According to Definition 8, $vSig(v_i)\&\, vSig(u_i) = vSig(v_i)$. Since d_i^I is an ancestor of u_i, $vSig(u_i)\&vSig(d_i^I) = vSig(u_i)$. Therefore, we can conclude that $vSig(v_i)\&\, vSig(d_i^I) = vSig(v_i)$.

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^*; and $Sig(\overrightarrow{v_i, v_j})\&Sig(\overrightarrow{u_i, u_j}) = Sig(\overrightarrow{v_i, v_j})$. Since d_i^I and d_j^I are ancestors of u_i and u_j, respectively, according to VS*-tree's structure, we know that $Sig(\overrightarrow{u_i, u_j})\&Sig(\overrightarrow{d_i^I, d_j^I}) = Sig(\overrightarrow{u_i, u_j})$. Therefore, we can conclude that $Sig(\overrightarrow{v_i, v_j})\&Sig(\overrightarrow{d_i^I, d_j^I}) = Sig(\overrightarrow{v_i, v_j})$.

In summary, according to Definition 10, we know that $(d_1^I, ..., d_n^I)$ is a summary match of Q^* in G^I.

(2) The second claim in Theorem 2 can be proved according to the first claim by the contradiction. □

Theorem 3 *Any entity vertex u in RDF graph G is accessed once in computing aggregate sets of trie-index by Algorithm 6.*

Proof According to T-index's structure, each entity vertex u is only in one path of T-index. Assume that u corresponds to node O in T-index. The dimension values of u only need to be accessed when computing $MS(O)$. Aggregate sets of O's ancestor nodes can be computed from its children nodes without accessing the original data. □

C. T-index Construction and Maintenance

Construction

The construction algoirthm of T-index is given in Algorithm 6. Initially, a scan of DB derives a list of all dimensions, and the dimension list DL is created. We introduce a root node into T-index (Lines 1-2 in Algorithm 6). When we insert a transaction $A(u)$ into T-index, we find a node O_i, where the path reaching O_i is the maximal prefix of $A(u)$ (Line 4). The maximal prefix means that (1) the path reaching node O_i is a length-n prefix of $A(u)$ and (2) there exists no path that is a length-$(n+1)$ prefix of $A(u)$. Let $\{l_1, ..., l_{N_i}\}$ be the dimensions along the path from the root to node O_i and $\{p_i, ..., p_k\}$ be $A(u)$'s dimensions. If $\{l_1, ..., l_{N_i}\} \subset \{p_1, ..., p_k\}$, then we need to add new nodes to extend the path (Lines 5-6). Specifically, we introduce a new node O_j as a child of node O_i to denote the $(n+1)$-th dimension of $A(u)$. Iteratively, we introduce $(|A(u)| - n)$ new nodes (i.e., nodes corresponding to dimensions $\{p_1, ..., p_k\} \setminus \{l_1, ..., l_{N_i}\}$) to extend the path from node O_i. We register these newly introduced nodes into the corresponding dimensions in DL (Line 7). Furthermore, we update vertex lists of each node along the path (Line 9). We iterate and insert all transactions into T-index (Lines 3-10).

Algorithm 6 T-index and Materialized Aggregate Sets

Input: Transaction Database DB
Input: RDF graph G
Output: T-tree and aggregate sets associated with each node.

1: Scan DB to find all dimensions and build dimension list DL.
2: Introduce a root node into T-tree
3: **for** each transaction $A(u)$ in DB **do**
4: Find a node O_i in T-tree, where the path reaching O_i (i.e.,$\{N_0, ..., N_i\}$) is the maximal prefix of $A(u)$. Let $\{l_1, ..., l_{N_i}\}$ be dimensions along path $\{N_0, ..., N_i\}$.
5: **if** $\{l_1, ..., l_{N_i}\} \subset \{p_1, ..., p_k\}$, where $\{p_1, ..., p_k\}$ are $A(u)$'s dimensions **then**
6: Introduce $|A(u) - n|$ nodes $\{p_1, ..., p_k\} \setminus \{l_1, ..., l_{N_i}\}$ to extend the path from node O_i.
7: Register these new introduce nodes into the corresponding dimensions in DL
8: Record the ID of transaction $A(u)$ into vertex lists $O_i.L$, where $i = 0, ..., k$
9: **for** each child O_i of the root **do**
10: Call Function PostOrderVisit(O_i) (Algorithm 7)

Algorithm 7 Function: PostOrderVisit(O)

1: **for** each child node O_i of O **do**
2: Call Function PostOrderVisit(O_i).
3: **for** each child O_i of O **do**
4: compute $MS'(O_i) = \prod_{(p_1,p_2,...,p_n)} MS(O_i)$.
5: $MS(O) = \bigcup_i M'(O_i)$
6: **for** each vertex u in $O.L$ but not in $\bigcup_i N_i.L$ **do**
7: $F'(u) = \prod_{(p_1,...,p_n)} u$
8: **if** there exists some aggregate tuple t', where $t'.D = F'(u)$ **then**
9: Register vertex ID of u in vertex list $t.L$
10: **else**
11: Generate a new aggregate tuple t', where $t'.D = F'(u)$ and insert vertex ID of u into $t'.L$
12: Insert t' into $MS(O)$

Maintenance

Updates of the RDF data requires efficient maintenance of T-index. Obviously, updates can be modeled as a sequence of triple deletions and triple insertions. As mentioned earlier, all dimensions are ordered in their frequency descending order in dimension list DL to improve SA query evaluation performance. However, RDF data updates may change the order of dimensions in DL requiring special care during index maintenance. Therefore, we consider updates of T-index in two cases based on whether or not the order of dimensions in DL changes.

Dimension List DL's order does not change

Consider the insertion of a new triple $\langle s, p, o \rangle$. If p is a link property, we do not need to update the T-index. Thus, we only consider the case when p is an attribute property, i.e., dimension.

If s does not occur in the existing RDF data, we introduce a new transaction into D. We insert the new transaction into one path of T-index following Definition 16. Accordingly, we need to update the materialized aggregate sets $MS(O_i)$ along the path. The detailed steps are the same as Lines 4-9 in Algorithm 6.

If s is already in the existing RDF graph, assume that s's existing dimensions are $\{p_1, ..., p_n\}$ and $Freq(p_i) > Freq(p) > Freq(p_{i+1})$ in dimension list DL. Again, in the case of equality, order is chosen arbitrarily. This means that the new inserted dimension p should be inserted between p_i and p_{i+1}. We locate two nodes O_i and O_n, where the path reaching node O_i (and O_n) has dimensions $(p_1, ..., p_i)$ (and $(p_1, ..., p_i, ...p_n)$[5]). We remove subject s from all materialized sets along the path between nodes O_{i+1} and O_n, where O_{i+1} is a child node of O_i. Then, we insert dimensions $(p, p_{i+1}, ..., p_n)$ into T-index from node O_i, and update the materialized sets along the path.

Consider inserting a triple
⟨y:Franklin_D._Roosevelt, bornOnDate, "1882-01-30"⟩
into RDF triple table T shown in Figure 1a, where
y:Franklin_D._Roosevelt
corresponds to vertex 007.

Although inserting the triple changes the frequency of dimension "bornOnDate", it does not lead to changing the order in DL. 007's dimensions are (hasName, gender, title). According to dimension order, the inserted triple should be placed between "gender" and "title". Therefore, we delete 007 from path "$O_2 - O_7$" in the original T-index. Then, we insert 007 into path "$O_2 - O_3 - O_4$". Path "$O_2 - O_7$" is deleted, since the updated aggregate sets in O_7 are empty. Figure 24 shows the updated T-index after inserting the triple.

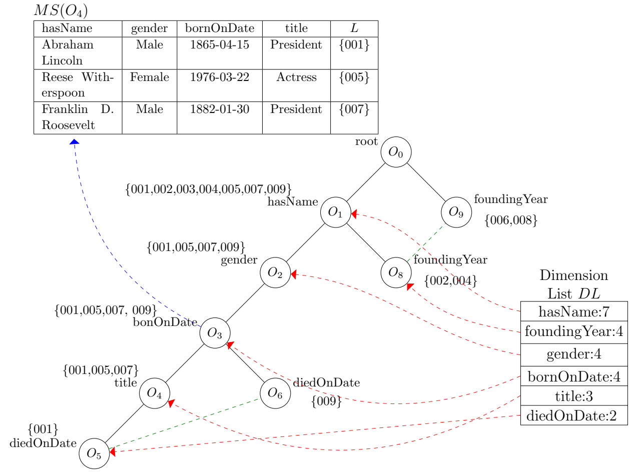


Fig. 24: Addition of Triple ⟨y:Franklin_D._Roosevelt, bornOnDate, "1882-01-30"⟩

Suppose now that we need to delete a triple $\langle s, p, o \rangle$, where p is an attribute property as discussed above. Assume that s's existing dimensions are $\{p_1, ..., p_n\}$ and $p = p_i$, i.e., p and p_i are the same dimension. We locate two nodes O_i and O_n, where the path reaching node O_i (and O_n) has dimensions $(p_1, ..., p_i)$ (and $(p_1, ..., p_i, ...p_n)$). We remove subject s from all materialized sets along the path between nodes O_{i+1} and O_n. Then, we insert dimensions $(p_{i+1}, ..., p_n)$ into T-index from node O_i, and update the materialized sets along the path. Again T-index itself does not need to be modified.

Dimension List DL's order changes

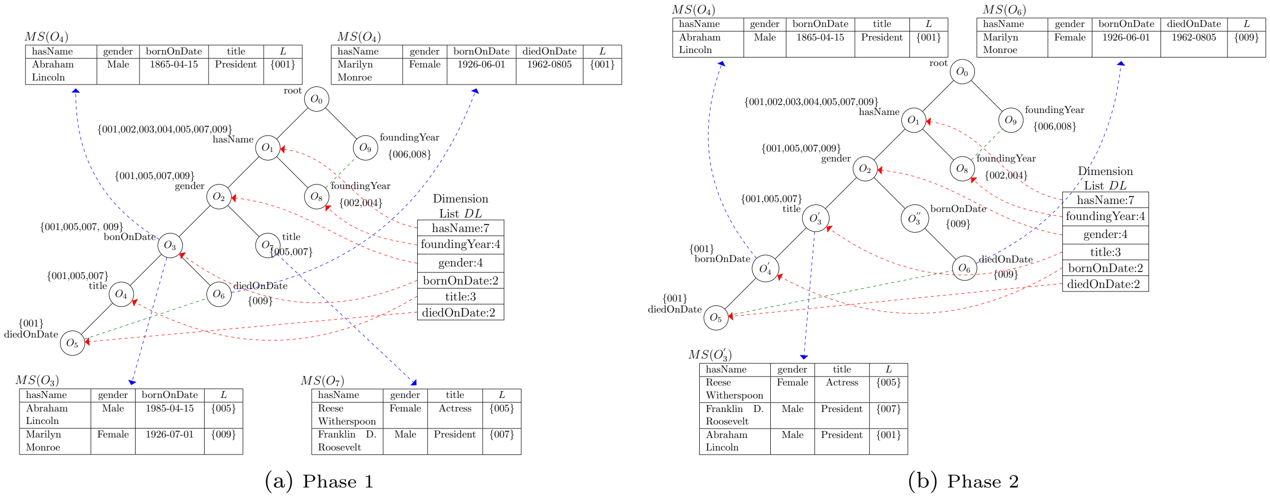
Some triple deletions and insertions that affect dimension frequency will lead to changing the order of dimensions in DL. Assume that two dimensions p_i and p_j need to be swapped in DL due to inserting or deleting one triple $\langle s, p, o \rangle$. Obviously, $j = i + 1$, i.e., p_i and p_j are adjacent to each other in DL [6].

Updates are handled in two phases. First, we ignore the order change in DL and handle the updates using the method in Section 13. Second, we swap p_i and p_j in DL, and change the structure of T-index and the relevant materialized sets.

We focus on the second phase. There are only three categories of paths that can be affected by swapping

[5] Although dimension list is a set, when the order is important, we specify them as a list enclosed in ( ).

[6] Assume that some dimensions p_i, $p_{i+1}, ..., p_{j-1}$ p_j have the same frequency. If we insert one triple $\langle s, p, o \rangle$, we need to swap adjacent dimensions several times.



(a) Phase 1

(b) Phase 2

Fig. 25: Deletion of triple ⟨y:Reese_Witherspoon, bornOnDate, "1976-03-22"⟩

p_i and p_j: (1) path has both dimensions p_i and p_j, i.e., path has dimensions $(p_1, .., p_i, p_j, ..., p_m)$; (2) path shares a prefix $(p_1, ..., p_i)$ with a path in the first category; and (3) path shares a prefix $(p_1, ..., p_{i-1})$ with a path in the first category and p_{i-1}'s child is p_j. We discuss below how to update the three categories of affected path.

We first locate all paths in the first category, i.e., find all paths that have both dimensions p_i and p_j. Consider one such path H with dimensions $(p_1, ..., p_i, p_j, ..., p_n)$. Nodes O_i and O_j have dimensions p_i and p_j, respectively. We rename O_i as O'_i and O_j as O'_j. Moreover, we set O'_i's corresponding dimension to be p_j and O'_j's corresponding dimension to be p_i. We compute aggregated set $MS(O'_i)$ by projecting $MS(O_j)$ over dimensions $(p_1, ..., p_{i-1}, p_j)$, i.e., $MS(O'_i) = \prod_{(p_1,...,p_{i-1},p_j)} MS(O_j)$ and set $MS(O'_j) = MS(O_j)$, since they have the same group by dimensions.

If node O_i has a branch (i.e. O_i has other child nodes in addition to O_j) in the original T-index, which is exactly the case belonging to the second category, we introduce a node O''_i as a child of O_{i-1}, where O_{i-1} is a parent of O_i in the original T-index, and set the dimension in O''_i as p_i. We move all child nodes of O_i except for O_j to be children of O''_i. We compute aggregate set $MS(O''_i) = MS(O_i) \setminus \prod_{(p_1,...,p_i)} MS(O_j)$. Specifically, for each aggregate tuple t in $MS(O_i)$, we check whether there exists an aggregate tuple $t' \in \prod_{(p_1,...,p_i)} MS(O_j)$, where $t.D = t'.D$. If there exists such an aggregate tuple, we generate a new aggregate tuple t'', where $t''.D = t.D$, and $t''.L = t.L \setminus t'.L$, and insert t'' into $MS(O''_i)$. Otherwise, we insert t into $MS(O''_i)$ directly.

If node O_{i-1} has a child O_m whose corresponding dimension is p_j, where O_{i-1} is a parent of O_i in the original T-index, which is a case belonging to the third category, we merge node O'_j and O_m. Specifically, we remove O_m and move O_m's child nodes to be O'_i's child nodes. Then, we compute $MS(O'_i) = MS(O'_i) \cup MS(O_m)$. Specifically, for each aggregate tuple t in $MS(O_m)$, we check if there exists an aggregate tuple t' in $MS(O'_i)$, where $t.D = t'.D$. If so, we set $t'.L = t'.L \cup t.L$. Otherwise, we insert t into $MS(O'_i)$ directly.

Consider deleting the triple ⟨y:Reese_Witherspoon, bornOnDate, "1976-03-22"⟩ from the RDF triple table T shown in Figure 1a. This changes the order of "title" and "bornOnDate" in dimension list DL, since the frequency of "bornOnDate" is changed to 2. We first assume that the order does not change, and we delete the triple using the method in Section 13. Figure 25a shows the updated T-index after the first phase.

Now, we swap "title" and "bornOnDate". Path H_1 ="$O_0 - O_1 - O_2 - O_3 - O_4 - O_5$" is a path in the first category, since O_3 and O_4's corresponding dimensions are "bornOnDate" and "title". Path H_2 ="$O_0 - O_1 - O_2 - O_3 - O_6$" is a path in the second category, since it shares prefix $(O_0 - O_1 - O_2 - O_3)$ with H_1. Path H_3="$O_0 - O_1 - O_2 - O_7$" is one path in the third category, since it shares prefix "$O_0 - O_1 - O_2$" with H_1 and O_2's child node's dimension is "title".

In the first step of phase 2, we first find the path "$O_0 - O_1 - O_2 - O_3 - O_4$", where the dimensions in O_3 and O_4 are "bornOnDate" and "title", respectively. Then, we rename O_4 as O'_4 with dimension "bornOnDate", and rename O_3 as O'_3 with dimension "title". $MS(O'_4) = MS(O_4)$ and $MS(O'_3) = \prod_{(hasName,gender,title)} MS(O_4)$. Since transaction 009 does not have dimension "title", we introduce a new node O''_3, and $MS(O''_3) = MS(O_3) - \prod_{(hasName,gender,bornOnDate)} MS(O_4)$. Finally,

we merge nodes O_7 with node O'_3, since they share the same prefixes. Figure 25b shows the final updated T-index.

D. Additional Experiments

Effect of Pruning Power

Theorem 1 shows that CL (matches of Q^* over G^*) is a subset of RS (matches of Q over G). Figure 26 shows both $|CL|$ and $|RS|$ of queries over Yago dataset. We find that $|CL| < 3 \times |RS|$, which indicates the low cost of the verification process in our method.

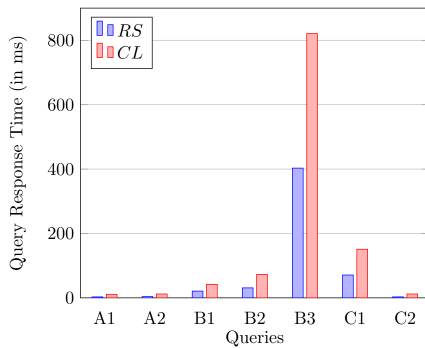


Fig. 26: $|RS|$ VS. $|CL|$ over Yago Dataset

BVS-Query Algorithm vs VS*-Query Algorithm*

We compare the two query algorithms, BVS*-Query (Algorithm 1) and VS*-query (Algorithm 2) over both Yago and DBLP datasets. Figure 27 shows that VS*-query algorithm is much faster than BVS*-Query algorithm. The reason behind that is the following: For each summary match, we always need to materialize all child states in BVS*-Query, which is quite expensive. However, VS*-query algorithm uses VS*-tree to reduce the search space instead of materializing all summary matches in all non-leaf levels.

S-tree+Join VS. gStore

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 $vSig(u_{i_j}) \& vSig(v_i) = vSig(v_i)$ 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, $|R_i|$ is shrunk according to Theorem

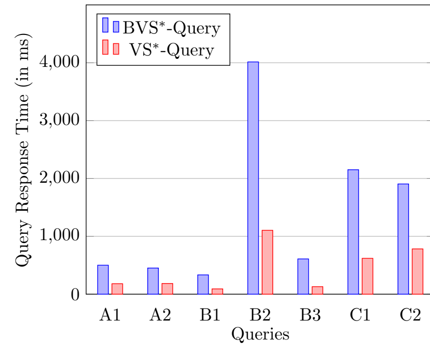


Fig. 27: BVS*-Query vs VS*-query

2. Therefore, VS*-query algorithm in our gStore system is much faster than S-tree+Join method, as shown in Figure 28a. Given a sample query in our experiment in Figure 28b, we show the candidate sizes for each vertex v_i in Q using S-tree and VS*-tree for pruning, respectively. Obviously, VS*-tree provides stronger pruning power. Thus, VS*-query algorithm is much faster than S-tree+Join.

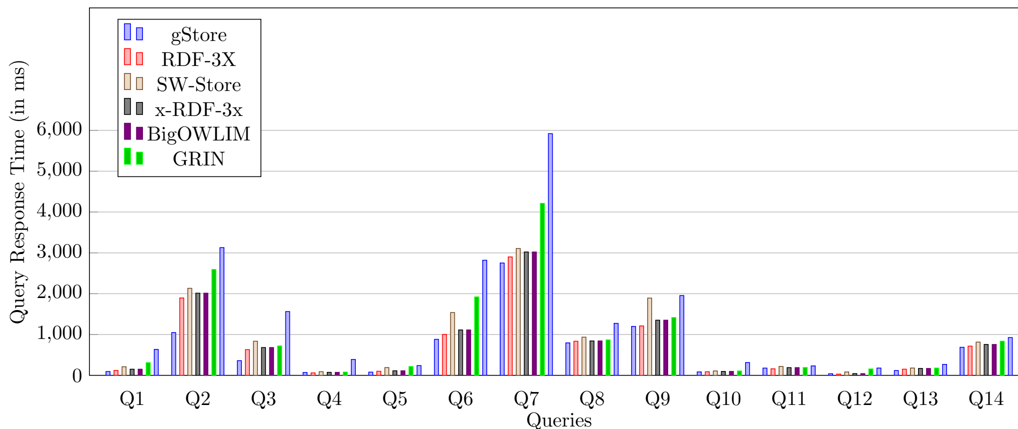


Fig. 29: Exact Queries over LUBM Dataset

Table 16: Yago Data – No. of 1's at Various Levels

δ	1st level	2nd level	3rd level
80	221	211	181
70	191	172	159
60	181	161	151
50	173	155	145
40	167	153	142
30	165	149	138

Table 17: DBLP Data – No. Of 1's at Various Levels

δ	1st level	2nd level	3rd level
80	183	139	103
70	161	120	91
60	153	115	82
50	140	109	75
40	138	105	73
30	135	102	68

time, since this order ensures that more prefixes can be shared among different transactions. Furthermore, we find that T-index in LUBM has a small number of nodes. This is because LUBM data is more structured than Yago data, since all RDF triples are generated following an ontology in LUBM. The structured data causes most entities to have almost equal dimensions; thus, they introduce few branches in T-index, i.e., the number of nodes is small.

Table 18: Evaluating T-index

Datasets	Dimension Order	Construction Time (sec)			Index size (MB)			Node #
		T-index	Aggregate Sets	Total Time	T-index (including vertex lists in each node	Aggregate Sets	Total Size	T-index
Yago	Frequency Decreasing	28	41	69	17	158	175	672
	Frequeny Increasing	35	58	93	19	210	229	810
	Random	32	52	80	18	180	198	750
LUBM	Frequency Decreasing	48	24	72	16	249	265	15
	Frequeny Increasing	53	36	89	20	287	307	22
	Random	50	31	81	18	271	289	16

We evaluate the online performance of different dimension orders in answering SA queries. Table 19 shows

Table 19: Evaluating Dimension Orders in *DL*

Dimension Order	Query Response Time (msec)						Number of Accessed Nodes					
	Yago			LUBM			Yago			LUBM		
	SA1	SA2	SA3	SA1	SA2	SA3	SA1	SA2	SA3	SA1	SA2	SA3
Frequency Decreasing	52	163	951	19	28	160	4	5	3	2	1	3
Frequency Increasing	150	236	1223	31	45	235	34	72	8	3	2	3
Random	67	200	1103	27	37	189	7	36	5	2	2	2

that SA query on T-index using descending frequency order has the fastest response time, since it accesses the minimal number of nodes in T-index.

Furthermore, our experiments show that the random choice of the order for dimensions with the same frequency does not affect online performance, because only a few dimensions have the same frequency, and T-index's size and node numbers are similar in different arbitrarily defined orders.

Additional Scalability Experiments

The full set of scalability experiments involving gStore, RDF-3x, and Virtuoso are given in Table 20. The first and last columns are identical to what is reported in Table 4 in Section 11.8; they are included here for completeness.

Table 20: Scalability of Query Performance on LUBM

Queries	Query Response Time (msec)											
	LUBM-100			LUBM-200			LUBM-500			LUBM-1000		
	gStore	RDF-3x	Virtuoso	gStore	RDF-3x	Virtuoso	gStore	RDF-3x	Virtuoso	gStore	RDF-3x	Virtuoso
Q1	1310	22634	2152	4146	44589	11653	24691	99231	28252	43832	202728	54460
Q2	236	4276	30560	393	7595	71807	857	7084	>30 mins	1563	15008	> 30 mins
Q3	208	216	1981	297	480	2175	688	860	4868	1491	1737	7784
Q4*	153	44	1341	162	52	1358	311	6	1357	680	30	1357
Q5*	129	48	312	163	15	312	246	4	328	131	12	243
Q6*	227	20	843	367	29	827	726	20	827	828	49	842
Q7	1016	960	5257	1786	2901	8861	4859	5513	19546	8301	14072	36848

Note that $*$ means that the query contains at least one constant entity.