gStore: a graph-based SPARQL query engine

Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang & Dongyan Zhao
gStore: a graph-based SPARQL query engine

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

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.

Keywords RDF · SPARQL · Graph database · Graph matching · Aggregate query

1 Introduction

The RDF (Resource Description Framework) data model was proposed for modeling Web objects as part of developing the semantic web. Its use in various applications is increasing.

A RDF dataset is a collection of (subject, property, object) triples denoted as \((s, p, o)\). A running example is given in Fig. 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 \((Q_1)\):

\[
\text{SELECT ?name WHERE }
\{ ?m <hasName> ?name.}
\text{?m <bornOnDate> ‘1809-02-12’.}
\text{?m <diedOnDate> ‘1865-04-15’}.
\]

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., [1, 19, 32]), as well as systems, such as RDF-3x [20], x-RDF-3x [22], Hexastore [30] and SW-store [2], are designed to address scalability over large datasets. 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 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 (Q2):

```
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 (Q3):

```
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 multiedge 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, Fig. 1b shows a RDF graph G corresponding to RDF triples in Fig. 1a. We formally define an RDF graph in Definition 1. Note that, the numbers above the boxes in Fig. 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 Q2. Usually, query graph Q is a connected graph. Oth-
The rest of this paper is organized as follows. We discuss the related work and preliminaries in Sects. 2 and 3, respectively. We give an overview of our solution in Sect. 4. We discuss the storage and encoding method in Sect. 5. We then present the VS*-tree index in Sect. 6 and an algorithm for SPARQL query processing in Sect. 7. In order to support aggregate queries efficiently, we develop T-index in Sect. 8 and aggregate query processing algorithm in Sect. 9. We discuss the maintenance of indexes (VS*-tree and T-index) as RDF data get updated in Sect. 10. We study our methods by experiments in Sect. 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 Fig. 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 [20,19] 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 [2]. 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 [2], RDF-3x [20], 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 [2], 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 [32]. 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 \( V_0 \) 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 Fig. 3a (we show how to compute this answer in Sect. 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 Fig. 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.

### 3 Preliminaries

An RDF dataset is a collection of \((subject, property, object)\) triples \((s, p, o)\), 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 Fig. 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 \( G = (V, L_V, E, L_E) \), where

1. \( V = V_c \cup V_e \cup V_l \) is a collection of vertices that correspond to all subjects and objects in RDF data, where \( V_c \), \( V_e \), and \( V_l \) are collections of class vertices, entity vertices, and literal vertices, respectively.
2. \( L_V \) is a collection of vertex labels. 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.

<table>
<thead>
<tr>
<th>title</th>
<th>gender</th>
<th>foundingYear</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>President</td>
<td>Male</td>
<td>1718</td>
<td>1</td>
</tr>
<tr>
<td>Actress</td>
<td>Female</td>
<td>1976</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Answer to Query \( Q_3 \)

<table>
<thead>
<tr>
<th>title</th>
<th>gender</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>President</td>
<td>Male</td>
<td>2</td>
</tr>
<tr>
<td>Actress</td>
<td>Female</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Answer to Query \( Q_4 \)

Fig. 3 Difficulty of using materialized views a answer to query \( Q_3 \) b answer to query \( Q_4 \)
Table 1 Frequently used notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>A RDF graph (Definition 1)</td>
</tr>
<tr>
<td>$Q$</td>
<td>A SPARQL query graph (Definition 2)</td>
</tr>
<tr>
<td>$v$</td>
<td>A vertex $v$ in a SPARQL query graph (Definition 3)</td>
</tr>
<tr>
<td>$u$</td>
<td>A vertex in RDF graph $G$ (Definition 3)</td>
</tr>
<tr>
<td>$e\text{Sig}(e)$</td>
<td>An edge Signature (Definition 4)</td>
</tr>
<tr>
<td>$v\text{Sig}(u)$</td>
<td>A vertex signature (Definition 5)</td>
</tr>
<tr>
<td>$RS$</td>
<td>The answer set of the SPARQL query matches</td>
</tr>
<tr>
<td>$CL$</td>
<td>The candidate set of the SPARQL query matches</td>
</tr>
<tr>
<td>$G^{*}$</td>
<td>A data signature graph (Definition 6)</td>
</tr>
<tr>
<td>$Q^{*}$</td>
<td>A query signature graph</td>
</tr>
<tr>
<td>$A(u)$</td>
<td>A transaction of vertex $u$ (Definition 16)</td>
</tr>
<tr>
<td>$O$</td>
<td>A node in T-index (Definition 16)</td>
</tr>
<tr>
<td>$O_L$</td>
<td>The corresponding vertex list of node $O$</td>
</tr>
<tr>
<td>$MS(O)$</td>
<td>The corresponding aggregate set of node $O$</td>
</tr>
</tbody>
</table>

3. $E = \{u_1, u_2\}$ 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 $u_1 \rightarrow u_2$ 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 Sect. 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 $Q = \langle V^Q, L^Q, E^Q, L_E^Q, FL \rangle$, where

1. $V^Q = V_p^Q \cup V_e^Q \cup V_l^Q \cup V_f^Q$ 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 $V_e^Q$ and $V_l^Q$ are collections of class vertices, entity vertices, and literal vertices in the query graph $Q$, respectively.
2. $L^Q_v$ is a collection of vertex labels in $Q$. The label of a vertex $v \in V_p^Q$ is $\phi$, that of a vertex $v \in V_l^Q$ is its literal value and that of a vertex $v \in V_e^Q \cup V_f^Q$ is its corresponding URI.
3. $E^Q$ is a collection of edges that correspond to properties in a SPARQL query. $L^Q_E$ 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.

SELECT $?g$ $?t$ $?y$ COUNT($m$) WHERE {
  ?m <gender> ?g .
  ?c <foundingYear> ?y .}
GROUP BY $?g$, $?t$, $?y$
HAVING COUNT($m$) > 1
HAVING condition (optional)

Fig. 4 Three components in aggregate queries

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 $\{v_1, \ldots, v_n\}$. 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. 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 Fig. 2, vertices (005,006, 020,023,024) in RDF graph $G$ of Fig. 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 predefined 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 Fig. 4). This is not necessary, and in Sect. 9.3, we discuss more general cases.
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 (Fig. 7). Then, each entity and class vertex is encoded into a bitstring (called vertex signature). The encoding technique is discussed in Sect. 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 (Fig. 5). Specifically, $G^*$ is induced by all entity and class vertices in $G$ to 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 Fig. 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 Fig. 5a.

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, 33]).

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 (Sect. 5) that maps each vertex in $G^*$ to a signature. For the second issue, we design a novel index structure called VS*-tree (Sect. 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 $G^*$ (Sect. 7).

VS*-tree is also used in answering aggregate SPARQL queries (Sect. 9). We first decompose an aggregate query $Q$ into star aggregate queries $S_i (i = 1, \ldots, n)$, 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 Fig. 6. We make use of materialized views to efficiently process star aggregate queries without performing joins. For this purpose, we introduce T-index (Sect. 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 $S_i (i = 1, \ldots, n)$ 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.

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 accordingly. Therefore, we also address the index maintenance issues of VS*-tree and T-index (Sect. 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, \( adjList(u) = \{(e_i, eLabel, e, nLabel)\} \), 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 \( "e\) prefix from the specification of \( adjList \). Figure 7 shows part of the adjacency list table for the RDF graph in Fig. 1b.

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: \( eSig(e).e \) and \( eSig(e).n \). 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 \( H_i (i = 1, \ldots , m) \), 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.

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 [11], 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: \{1809, 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 Sect. 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:

\[
vSig(u) = eSig(e_1)|\ldots|eSig(e_n)
\]

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 Figs. 1b and 7, there are seven adjacent edges. We can encode each adjacent edge by its edge signature, as shown in Fig. 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 \( u_1, u_2 \) in \( G^* \), its edge label is also a signature, denoted as \( Sig(u_1, u_2) \), to denote the property between \( u_1 \) and \( u_2 \).

We adopt the same hash function in Definition 5 to define \( Sig(u_1, u_2) \). Specifically, we set \( m \) out of \( M \) bits in...
$\text{Sig}(\vec{u}_1, \vec{u}_2)$ 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 $e(\text{Label}, n\text{Label})$ into a bitstring $e\text{Sig}(e)$ according to Definition 5. Note that, if the adjacent neighbor vertex of $v$ is a parameter vertex, we set $e\text{Sig}(e).n$ 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($\text{str}(?bd)$, “1976”)), we only consider the substring without “ wildcard” in the label. For example, in Fig. 2, we can only encode substring “1976” for vertex $bd$. The vertex signature $v\text{Sig}(v)$ 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 $v\text{Sig}(v)$, and each edge $(\vec{v}_1, \vec{v}_2)$ in $Q^*$ is associated with an edge signature $\text{Sig}(\vec{v}_1, \vec{v}_2)$. Figure 5 shows $Q^*$ that corresponds to query $Q$.

**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. $v\text{Sig}(v_i) \& v\text{Sig}(u_i) = v\text{Sig}(v_i), i = 1, \ldots, n$, 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 $\text{Sig}(\vec{v}_i, \vec{v}_j) \& \text{Sig}(\vec{u}_i, \vec{u}_j) = \text{Sig}(\vec{v}_i, \vec{v}_j)$. \hfill $\square$

Note that, each vertex $u$ (and $v$) in data (and query) signature graph $G^*$ (and $Q^*$) has one vertex signature $v\text{Sig}(u)$ (and $v\text{Sig}(v)$). For simplicity, we use $u$ (and $v$) to denote $v\text{Sig}(u)$ in $G^*$ (and $v\text{Sig}(v)$ 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. \hfill $\square$

### 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 $R_i = \{u_{i1}, u_{i2}, \ldots, u_{in}\}$, where $v_i \& u_{ij} = v_i$ (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 $q$ [28]. 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 $\{s_i\}$. 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 Fig. 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, 33]), they are not scalable to very large graphs. Therefore, we develop VS*-tree (vertex signature tree) to index a large data signature graph $G^*$ that also supports the second step. VS*-tree is a multiresolution summary graph based on S-tree that can be used to reduce the search space of subgraph query processing.

**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., $d_i^I.Sig = OR(d_{i+1}^I.Sig, \ldots, d_{n}^I.Sig)$) where $d_{j}^I$ 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 $Sig(d_i^I, d_j^I)$ 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” $\overrightarrow{u_1, u_2}$ if there is a directed edge from $u_1$ to $u_2$ in $G^*$, and an edge signature $Sig(u_1, u_2)$ 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 is the following: bornIn $\rightarrow$ 10000, diedIn $\rightarrow$ 00001, hasCapital $\rightarrow$ 00100, and locatedIn $\rightarrow$ 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 Fig. 10, since there is a super edge from 008 to 004, we introduce a super edge from $d_3^2$ to $d_3^2$ (i.e., $\overrightarrow{d_3^2, d_3^2}$) 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 $d_i^k$ is also introduced since there is a (super)edge from 006 to 004, both of which are children of $d_i^k$.

Figure 11 shows the VS$^*$-tree over $G^*$ of Fig. 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 $v_i$ ($i = 1, \ldots, n$) and a summary graph $G^I$ at the $I$-th level of VS$^*$-tree. A set of nodes $\{d_i^I\}$ ($i = 1, \ldots, n$) at

![Diagram](image-url)

**Fig. 10** Building super edges

**Fig. 11** VS$^*$-tree

$G^I$ is called a summary match of $Q^*$ over $G^I$, if and only if the following conditions hold:

1. $vSig(v_i) \& d_i^I.Sig = vSig(v_i)$, $i = 1, \ldots, n$;
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 $d_i^I$. For example, given a query signature graph $Q^*$ (in Fig. 5) and a summary graph $G^3$ of VS$^*$-tree (in Fig. 11), we can find one summary match $\{\overrightarrow{d_1^I, d_3^I}\}$. Summary matches can be used to reduce the search space for subgraph search over $G^*$ as we discuss next.

As we discuss in Sect. 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 $\delta$, we partition all of $u$’s neighbors into $n$ groups $g_1, \ldots, g_n$, 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 $vSig(u[i])$ $(i = 1, \ldots, n)$ is no larger than $\delta$. Given a vertex $u$, if the number of 1’s in $vSig(u)$ is no larger than $\delta$, $u$ has only one instance $u[1]$. Then, we use these vertex instance signatures of $u$ ($vSig(u[i])$, $i = 1, \ldots, n$) 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 $vSig(u'_i)$ at the leaf level of the revised VS*-tree, we introduce an edge between $u[i]$ and $u'_i$ 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 Fig. 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 Fig. 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., $v[j]$, $j = 1, \ldots, m$, 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 $Q'_2$ (Fig. 5) has three neighbors, and thus, we introduce three instances $v_1[1]$, $v_1[2]$, and $v_1[3]$ that corresponds to three neighbors, respectively, Fig. 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 Sect. 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 $\{u_1, \ldots, u_n\}$ form 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 $d_i^I$. $(d_1^I, \ldots, d_n^I)$ must form a summary match (Definition 10) of $Q^*$ over $G^I$. 

![Fig. 12 Optimizing VS*-tree](image-url)
2. If there exists no summary match of $Q^*$ over $G^1$, 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 Fig. 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 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 $d^1_1\cdot children \times d^1_1\cdot children = \{d^2_1, d^2_2\} \times \{d^1_2, d^1_2\} = \{(d^1_2, d^1_2), (d^2_1, d^1_2), (d^1_2, d^2_1), (d^2_2, d^2_2)\}$. The set of child states that are summary matches of $Q^*$ is called valid child states, and they are inserted into queue $H$. In this example, only $(d^1_2, d^1_2)$ is a summary match of $Q^*$, and 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 $G^*$.

Definition 11. Given a query signature graph $Q^*$ with $n$ vertices $\{v_1, \ldots, v_n\}$, and $n$ nodes $\{d^1_1, \ldots, d^1_n\}$ in VS*-tree that form a summary match of $Q^*$, $n$ nodes $\{d^1_1, \ldots, d^1_n\}$ form a child state of $\{d^1_i, \ldots, d^1_i\}$, if and only if $d^1_j$ is a child node of $d^1_i$, $i = 1, \ldots, n$. Furthermore, if $\{d^1_i, \ldots, d^1_n\}$ is also a summary match of $Q^*$, $\{d^1_i, \ldots, d^1_n\}$ is called a valid child state of $\{d^1_i, \ldots, d^1_n\}$.

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 $CL = \emptyset$
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 $CL$

![Fig. 13 BVS*-query algorithm process](image)

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 $G^*$ in VS*-tree, we try to prune nodes $d^j_1 (\in G^1)$ that cannot match $v_i$ in any summary match of $Q^*$ over $G^1$. At the leaf level of the index, this shrinks the candidate list $C(v_i)$. Let us recall query $Q^*_2$ in Fig. 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 $(d^3_1, d^3_2)$ for $v_1$ and one candidate $(d^3_2)$ for $v_2$. There is one edge from $v_1$ to $v_2$ whose label signature is 10000. $d^3_1$ has one edge to candidates of $v_2$ with an edge signature of 10000. Obviously, 00010&100000 $\neq$ 10000, and, therefore, $d^3_1$ cannot be in the candidate list of $v_1$. If a node cannot match $v_i$, all descendant nodes of $d$ can be pruned safely. It means that node $d^3_1$ and its descendant nodes are pruned from candidates of $v_1$ resulting in a single candidate $\{005\}$ for $v_1$. Therefore, the join space is reduced to $1 \times 2 = 2$.

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 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 $(\in G^1)$ VS*-tree cannot match $v_i$, if and only if one of the following conditions hold:

1. $vSig(v_i) \land vSig(d) \neq vSig(v_j)$; or
2. $\exists v_j | v_i, v'_j \in Q^*, \forall d^j_1, d^j_2 \in G^1, vSig(v_j) \land vSig(d^j_2) \neq vSig(v_j)$.
3. $\exists v_j | v_i, v'_j \in Q^*, \forall d^j_1, d^j_2 \in G^1, dSig(v_i, v'_j) \lor Sig(d^j_1, d^j_2) \neq Sig(v_i, v'_j)$.

Furthermore, if a node $d (\in G^1)$ 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.
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 \( v_i \in Q^* \)
do
2: \( C(v_i) = d_i \), where \( d_i \) is the root of VS*-tree \( \{C(v_i)\} \) are candidate vertices in \( G^* \) to match \( v_i \) in \( Q^* \)
endfor
3: for each level summary graph \( G^I \) of VS*-tree \( I = 2, \ldots, h \) do \( h \) is the height of VS*-tree
   4: for each \( C(v_i), i = 1, \ldots, n \) do
      5: set \( C'(v_i) = \phi \)
      6: for each child node \( d_i' \) of each element in \( C(v_i) \) do
         7: for each instance \( v_i[j] \) of vertex \( v_i, j = 1, \ldots, m \) do
            8: if \( \text{Sig}(d_i') \& \text{Sig}(v_i[j]) = \text{sig}(v_i[j]) \) then
               9: push \( d_i' \) into \( C(v_i[j]) \)
            endif
         endfor
      endfor
   endfor
      10: Set \( C(v_i) = \bigcap_{i=1,\ldots,m} C(v_i[j]) \)
   endfor
      11: for each \( C(v_i), i = 1, \ldots, n \) do
      12: for each node \( d \) in \( C(v_i) \) do
         13: if \( \exists v_j \in \{v_1, v_2\} \in \text{Q^*} \) s.t. \( d' \in G^I, v \text{Sig}(d') \& \text{Sig}(d'') \neq \text{vSig}(v_j[j]) \) then
               14: Remove \( v_j \) from \( C(v_i) \)
            endif
         endfor
      endfor
      16: Call Algorithm 3 to find matches of \( Q^* \) over \( C(v_1) \times \ldots \times C(v_n) \), denoted as \( CL \).
17: for each candidate match in \( CL \) do
18: Check whether it is match of SPARQL query \( Q \) over RDF graph \( G \). If so, insert it into \( RS \).
20: return \( RS \).

**Definition 12** Given a subgraph \( Q' \) of \( Q^* \), an edge \( e = (v_1, v_2) \) in \( Q^* \) is called adjacent to \( Q' \) if and only if \( (e \notin Q') \land (v_1 \in Q' \land v_2 \notin Q') \).

**Definition 13** Given an adjacent edge \( e = (v_1, v_2) \) to \( Q' \), if \( (v_1 \in Q' \land v_2 \notin Q') \), \( e \) is called a backward edge if and only if \( (v_1 \in Q' \land v_2 \notin Q') \). Otherwise, \( e \) is called a forward edge.

Lines 6–10 of Algorithm 2 exploit the optimization we introduced (see end of Sect. 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 \( C(v_i) = [d_i] \). 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 \( \text{Sig}(d_i') \& \text{Sig}(v_i[j]) = \text{sig}(v_i[j]) \). If so, we push \( d_i' \) into \( C(v_i[j]) \). When we finish considering all instances of \( v_i \), we update \( C(v_i) = \bigcap_{j=1,\ldots,m} C(v_i[j]) \).

After finding candidates \( C(v_i) \) for each query vertex \( v_i \), we perform multiway join \( C(v_1) \times \ldots \times C(v_n) \) 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 \( G^* \).
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 (Sect. 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, and diedOnDate”. Thus, we have a transaction \( A(001) = \text{“hasName, gender, bornOnDate, title, and diedOnDate”} \). All transactions are collected to form a transaction database \( DB \), as shown in Fig. 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 \( \text{Freq}(p) = |\{A(u) | p \in A(u) \land A(u) \in DB\}|. \)

For example, the frequencies of “hasName”, “FoundYear”, “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, minimize the number of materialized views that need to be maintained. It also facilitates query processing, as discussed in Sect. 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 \( OL \) registering the IDs of all transactions \( A_i \), where the path reaching \( O \) is a prefix of \( A_i \).

\(^1\) The literature on aggregation and aggregate queries frequently refer to these attributes as dimensions. We follow the same convention.

Figure 14b shows an example of T-index. When inserting \( A(001) = \text{“hasName, gender, bornOnDate, title, and diedOnDate”} \) into the T-index, the path \( “O_0 − O_1 − O_2 − O_3 − O_4 − O_5” \) is followed. 001 is registered to \( O_1.L \) 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 of the vertex lists can be optimized by a variety of encoding techniques. We do not 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 \( l_1, l_2, \ldots, l_N \). According to \( OL \), 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 \( l_1, l_2, \ldots, l_N \). 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 Fig. 14b. Consider node \( O_4 \) in Fig. 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, and title). These precomputed 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 \( O_i, (i = 1, \ldots, n) \), 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 \( OL \) 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 \( O_i, (i = 1, \ldots, n) \), 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, \( OL \supseteq \bigcup_i N_i.L \). 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 $(p_1, \ldots, p_m)$. Initially, $MS(O) = \phi$. For each child node $O_i$ of $O$, we compute $MS'(O_i) = \prod_{j=1}^{m} MMS(O_j)$ (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 $p_1, \ldots, p_n$ in the RDF graph (Lines 6–12). Specifically, we define function $F(u) = \prod_{j=1}^{m} u_j$, which means projecting $u$’s adjacent properties over $(p_1, \ldots, p_n)$ (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. \(\square\)

We illustrate the construction of T-index using an example. First, a scan of $DB$ (Fig. 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, and diedOnDate), inserting nodes $O_1, O_2, O_3, O_4$ and $O_5$. Initially, we set $O_1.L = \{001\}, i = 1, \ldots, 5$.

2. $A(002)$ shares a common prefix (hasName) with the existing path and adds one new node $O_6$ (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, and title) and assessing entity vertex 005. The process is iterated until all aggregate sets are computed. $MS(O_4)$ and $MS(O_5)$ are given in Fig. 14b as examples.

### 9 Aggregate query processing

As noted in Sect. 4, we decompose a GA query $Q$ into several SA queries $\{S_i\}, i = 1, \ldots, n$, where each star center $v_j$ is an entity vertex in $Q$. The result of each $S_i (R(S_i))$ is computed using the approach discussed in Sect. 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 Sect. 9.2.
9.1 Star aggregate query processing

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

Given a SA query \( S = (v, \{p_1, \ldots, p_d\}, \{p_{d+1}, \ldots, p_n\}) \), we answer \( S \) using T-index by Algorithm 4. Let \( P = \{p_1, \ldots, p_d, \ldots, p_n\} \). Given the set of properties \( P \), we find their match \( (O_1, \ldots, O_n) \), where \( O_1 \) is a node in T-index and all nodes \( O_i \) (\( i = 1, \ldots, n \)) 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 \( O_i \) \((i=1,\ldots,n)\) are adjacent to each other in the path. Thus, it is possible to have multiple matches for a given \( P \). For each match \( (O_1, \ldots, O_n) \), we use \( Q \) to denote the farthest node from the root (Line 3 in Algorithm 4). The aggregate set associated with node \( Q \) is denoted as \( MS(Q) \). We compute \( MS'(Q) \) by projecting \( MS(Q) \) over group-by dimensions \( \{p_1, \ldots, p_d\} \) (i.e., \( MS'(Q) = \prod_{p_1, p_2, \ldots, p_d} MS(Q) \)). The \( MS'(Q) \) 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 \( S(v, \{p_1, \ldots, p_d\}, \{p_{d+1}, \ldots, p_n\}) \)

**Output:** An aggregate result set \( MS \) for a SA query \( Q \).

1. Locate all matches of properties \( \{p_1, \ldots, p_d\} \) \( \{p_{d+1}, \ldots, p_n\} \) in T-index
2. For each match \( m_i \) do
3. \( O_i \) denote the node in match \( m_i \) that is farthest from the root
4. \( MS(O_i) \) denote the aggregate set associated with node \( O_i \)
5. \( MS'(O_i) = \prod_{p_1, p_2, \ldots, p_d} MS(O_i) \)
6. \( MS = \bigcup MS'(O_i) \)
7. return \( MS \)

For example, given a SA query \( S_1(v_1, \{\text{gender}, \text{title}\}, \phi) \) in Fig. 6, group-by dimensions are \( \{\text{gender}, \text{title}\} \) 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 \( (O_1, O_2, O_3) \) in T-index corresponding to group-by dimensions. In the first match, \( O_4 \) is the farthest node from the root. Since \( MS(O_4) \) is an aggregate set over dimensions (\( \text{hasName} \), \( \text{gender} \), \( \text{bornOnDate} \), and \( \text{title} \)), we compute a temporary aggregate set \( MS'(O_4) \) on dimensions (\( \text{gender} \) and \( \text{title} \)) 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 (\( \text{hasName} \), \( \text{gender} \), and \( \text{title} \)), it is also projected over (\( \text{gender} \) and \( \text{title} \)) to get \( MS(O_7) \).

Note that, this is not necessarily \( O_n \), since node identifiers are arbitrarily assigned to help with presentation.

Finally, we obtain \( R(S_1) \) by merging \( MS'(O_4) \) and \( MS'(O_7) \). Figure 15 illustrates the process.

9.2 General aggregate query processing

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

**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 \( v_i, i = 1, \ldots, n \). 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 \( T_i = \bigcup_{t \in R(S_i)} L_i \), 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 \( G'(V' = \{u_j\}) \), \( E' = \{e'_j\} \), \( \Sigma' = \{eLabel(e'_j)\} \) 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 \( eLabel(e_j) = eLabel(e'_j) \)
Algorithm 5 General Aggregate (GA) Query Algorithm

**Input:** A GA query $Q$

**Output:** $R(Q)$ (Query Result)

1. Each entity vertex $v_i$ in $Q$, $i = 1, \ldots, n$, 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 all matches of $S_i$, $i = 1, \ldots, n$.
4. $T_i = \bigcup_{v_i \in R(S_i)} v_i$
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 $G$
7. $U = \{g_1\}$, where $g_1$ includes all subgraph matches
8. for each entity vertex $v_i$ in $J$, $i = 1, \ldots, n$ 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 $MS \in g \{MS[i] \in t.L \text{ and } MS[i] \text{ refers to the } i\text{-th vertex in } MS\}$
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 $R(Q)$

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

Obviously, we can perform multiway join $T_1 \Join \ldots \Join T_n$ to find matches of $J$ over RDF graph $G$. In order to speed up online performance, $T_i$ ($i = 1, \ldots, n$) 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 Sect. 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 $T_1 = T_1 \cap C(v_i)$. 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 Fig. 16b shows the flattened representation of these matches. Then, we need to partition these subgraph matches into one or more groups based on the 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 $U = \{g_1\}, j = 1, \ldots, m$, 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 multilevel partitioning over these subgraph matches. At the first level, we consider group-by dimensions in $R(S_i)$. For each group $g \in U$, we partition matches in $g$ into some new groups $g_i', i = 1, \ldots, m$, where each new group $g_i'$ has matches that share the same values over group-by dimensions in $R(S_i)$. Obviously, $g = \bigcup_j g_i'$. Specifically, in order to partition matches in group $g$, we sequentially scan $R(S_i)$. For each aggregate tuple $t \in R(S_i)$, we find a new group $g_i'$ of subgraph matches $MS$, such that $MS[i] \in t.L$ and $MS[i]$ refer to the $i$th vertex in subgraph match $MS$ (Lines 11–12). We insert these new groups $g_i'$ into $U'$ (Line 13).
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 \( Q_5 \).

\[
\text{SELECT } \forall d \: \forall c \: \text{COUNT}(\forall m) \text{ WHERE } \{ \forall m \: \text{bornIn} \: ?c, \: \forall m \: \text{bornOnDate} \: \forall d, \: \forall m \: \text{hasName} \: ?n. \} \\
\text{GROUP BY } \forall d \: \forall c
\]

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 Fig. 14a by adding a new dimension “hasURI” to each transaction. Following the method in Sect. 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 Sect. 8.

Given an aggregate query \( Q \), if \( Q \) has a triple \( (?m_i, \: p_i, \: ?o_i) \), where \( ?o_i \) is a group-by dimension and \( p_i \) is a link property, we rewrite the aggregate query by inserting a new triple \( (?o_i \: \text{hasURI} \: ?o_i') \) and replacing \( ?o_i \) by \( ?o_i' \) as the group-by dimension. For example, \( Q_4 \) has a triple \( (?m \: \text{<bornIn>} \: ?c) \), where “bornIn” is a link property and \( ?c \) is a group-by dimension. Thus, we rewrite \( Q_4 \) into \( Q_5 \) as follows.

\[
\text{SELECT } \forall d \: \forall c \: \text{COUNT}(\forall m) \text{ WHERE } \{ \forall m \: \text{<bornIn}> \: ?c, \: \forall m \: \text{<bornOnDate}> \: \forall d, \: \forall m \: \text{<hasName}> \: ?n, \: \forall c \: \text{<hasURI}> \: ?c \} \\
\text{GROUP BY } \forall d \: \forall c
\]

Obviously, \( Q_5 \) is a GA query (see Sect. 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 $(s, p, o)$ is inserted into RDF dataset. If $s$ existed in $G^*$ before insertion, we delete vertex $s$ and all adjacent edges from $G^*$. We employ the method discussed in Sect. 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 Sect. 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 $(s, p, o)$ 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 upward 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 $d_1, \ldots, d_n$, the distance between $u$ and $d_i$ ($i = 1, \ldots, n$) is defined as follows:

$$Dist(u, d_i) = \frac{\delta(u, d_i)}{|u|} \times \frac{\beta(u, d_i)}{\max_j \frac{\beta(u, d_j)}}$$

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

As mentioned earlier, after inserting vertex $u$, the signature of that leaf node and super edges adjacent to it may be updated, the change must be propagated upward 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 [28]. 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 Eq. 1. We perform this for each pair of $B + 1$ entities and keep the entity pair that leads to the minimal value of $\max(\alpha, \beta)$, where $\alpha$ and $\beta$ 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 affects 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, $1 \leq b \leq \frac{B}{2}$ [27]. $b = 1$ is the “tree-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 Sect. 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 $\{p_1, \ldots, p_n\}$ 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 $(p_1, \ldots, p_i)$ (and $(p_1, \ldots, p_{i-1}, p_n)$). 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 $(p, p_{i+1}, \ldots, p_n)$ 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 that $s$’s existing dimensions are $\{p_1, \ldots, p_n\}$ 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 Sect. 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 [2], RDF-3x [20], 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 [20] 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 $Q^\prime$ 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 more than 10 million entities and more than 180 million triples. The total size of Yago2 is larger than 23GB.

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 13–133 million. The total RDF file size is 1.5–15.1 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.
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 $\text{AdjEdges}(u, G)$ denote all edge labels adjacent to $u$ in $G$ and $\text{AdjEdges}(v, Q)$ defined similarly. If $\text{AdjEdges}(v, Q) \not\subset \text{AdjEdges}(u, G) \land vSig(v) \land vSig(u) = vSig(v)$, we say that a false drop has occurred. $vSig(v) \land vSig(u) = v$ means that $u$ is a candidate match $v$. However, $\text{AdjEdges}(v, Q) \not\subset \text{AdjEdges}(u, G)$ 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.

$$P_{false_{drop}} = \left(1 - e^{-\frac{|\text{AdjEdges}(u, G)|}{M}}\right)^m |\text{AdjEdges}(v, Q)|$$

where $|\text{AdjEdges}(u, G)|$ is $u$’s degree in $G$ and $|\text{AdjEdges}(v, Q)|$ 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 Sect. 6, a high-degree vertex will be decomposed into several instances and each instance’s degree is no larger than $\delta$. Thus, the false drop probability $P_{false_{drop}}$ increases with $|\text{AdjEdges}(u, G)|$. Since $|\text{AdjEdges}(u, G)| \leq \delta$, $P_{false_{drop}} \leq \left(1 - e^{-\frac{m\delta}{M}}\right)^{|\text{AdjEdges}(v, Q)|}$. 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 $1.0 \times 10^{-3}$, the parameter setting can be tuned to ensure the following holds:

$$P_{false_{drop}} \leq \left(1 - e^{-\frac{m\delta}{M}}\right)^{|\text{AdjEdges}(v, Q)|} \leq 1.0 \times 10^{-3}$$

From the query logs, we determine that the average value of $|\text{AdjEdges}(v, Q)|$ is 3. Therefore, in Yago and DBLP datasets, we set $m = 2, M = 97$ and $\delta = 10$. In this case, $P_{false_{drop}} \leq 1.0 \times 10^{-3}$ (Fig. 18).

### 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 [10].

It is clear that the larger $N$ is, the fewer conflicts exist among the vertex signatures. On the other hand, large $N$ will lead to large space cost of vertex signatures. Thus, we need to find a good trade-off for $N$. We use three star queries to evaluate the pruning power of the encoding technique. Given a star query $S$, we encode its central vertex $v$ into a vertex signature $vSig(v)$. We use $X$ to denote the number of vertex signatures $vSig(u)$, where $vSig(v) \land vSig(u) = vSig(v)$ in $G^*$. Obviously, $X$ decreases as $N$ increases as shown in Fig. 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 [20], 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 Fig. 20a, b, 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 Fig. 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.

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 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 systems 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 Sect. 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 worst performance drop for each system is 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.openlinksw.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 \(|R(Q')|/|R(Q)|\), 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 Fig. 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 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 “hasFamily-Name” 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.

![Fig. 21](image1.png) Exact query response time

![Fig. 22](image2.png) Wildcard query response time (FM: post-filtering)

![Fig. 23](image3.png) Aggregation queries in Yago
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 Sect. 10. For 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 the deferred-indexing approach [21] where the updates 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, because 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 \((b = 1 \text{ and } b = \frac{b}{2})\) in Table 2. Experiments show that the “free-at-empty” strategy \((b = 1)\) 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 is 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 is very simple involving few triple patterns (many involve two triple patterns) and is similar to each other. Furthermore, a large number of them involve constants that artificially increase their selectivity—for example, triple patterns \(?x undergraduateDegreeFrom \text{http://www.University0.edu and } ?x publicationAuthor http://www.Department0.University0.edu/ AssistantProfessor0\) 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, but we report the results obtained for the 7 queries that are given in Table 14 in the Online Supplements. 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 min window we allocated for executing queries. As expected, and as noted above, RDF-3x does better than gStore when the query con-

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Evaluating updates to VS*-tree in Yago (number of updated triples/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gStore</td>
</tr>
<tr>
<td>Insertion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.2 \times 10^4</td>
</tr>
<tr>
<td>Deletion (free-at-empty)</td>
<td>2.0 \times 10^4</td>
</tr>
<tr>
<td>Deletion (merge-at-half)</td>
<td>1.5 \times 10^4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Raw data</th>
<th>Offline processing time</th>
<th>Index size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF</td>
<td>1,499,131</td>
<td>13,402,474</td>
<td>2,178,865</td>
</tr>
<tr>
<td>RDF-3x</td>
<td>26,691,773</td>
<td>4,340,588</td>
<td>12m23s</td>
</tr>
<tr>
<td>Virtuoso</td>
<td>66,720,02</td>
<td>10,846,157</td>
<td>1h59m9s</td>
</tr>
<tr>
<td>Yago</td>
<td>133,553,834</td>
<td>21,715,108</td>
<td>2h10m15s</td>
</tr>
<tr>
<td>Yago2</td>
<td>19,012,854</td>
<td>4,339,591</td>
<td>20m35s</td>
</tr>
<tr>
<td>LUBM-100</td>
<td>3,007,385</td>
<td>852,448</td>
<td></td>
</tr>
<tr>
<td>LUBM-200</td>
<td>7,551,065</td>
<td>1,190,856</td>
<td></td>
</tr>
<tr>
<td>LUBM-500</td>
<td>15,136,798</td>
<td>1,640,281</td>
<td></td>
</tr>
<tr>
<td>LUBM-1000</td>
<td>3,126,721</td>
<td>3,071,205</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Scalability of query performance on LUBM

<table>
<thead>
<tr>
<th>Queries</th>
<th>Query response time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUBM-100</td>
</tr>
<tr>
<td></td>
<td>gStore</td>
</tr>
<tr>
<td>Q1</td>
<td>1310</td>
</tr>
<tr>
<td>Q2</td>
<td>236</td>
</tr>
<tr>
<td>Q3</td>
<td>208</td>
</tr>
<tr>
<td>Q4*</td>
<td>153</td>
</tr>
<tr>
<td>Q5*</td>
<td>129</td>
</tr>
<tr>
<td>Q6*</td>
<td>227</td>
</tr>
<tr>
<td>Q7</td>
<td>1016</td>
</tr>
</tbody>
</table>

* Indicates the query contains at least one constant entity

Table 5: Scalability of query performance on Yago2

<table>
<thead>
<tr>
<th>Queries</th>
<th>Query response time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>gStore</td>
</tr>
<tr>
<td>A1*</td>
<td>251</td>
</tr>
<tr>
<td>A2*</td>
<td>230</td>
</tr>
<tr>
<td>B1*</td>
<td>2157</td>
</tr>
<tr>
<td>B2</td>
<td>131</td>
</tr>
<tr>
<td>B3</td>
<td>198</td>
</tr>
<tr>
<td>C1</td>
<td>875</td>
</tr>
<tr>
<td>C2*</td>
<td>865</td>
</tr>
<tr>
<td>Wildcard</td>
<td>gStore</td>
</tr>
<tr>
<td>AW1*</td>
<td>3226</td>
</tr>
<tr>
<td>AW2*</td>
<td>6122</td>
</tr>
<tr>
<td>BW1*</td>
<td>8644</td>
</tr>
<tr>
<td>BW2</td>
<td>268</td>
</tr>
<tr>
<td>BW3</td>
<td>3197</td>
</tr>
<tr>
<td>CW1</td>
<td>151337</td>
</tr>
<tr>
<td>CW2*</td>
<td>6189</td>
</tr>
</tbody>
</table>

* Indicates the query contains at least one constant entity

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 in the Online Supplements) is identical to Q1 we report (Table 14 in the Online Supplements) 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 in the Online Supplements). 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 depends on the selectivity of the triple patterns that include constants for which extensive indexing can be exploited.
RDF-3X does not support wildcard queries, and thus, we have not compared the two systems for those queries.

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.

Acknowledgments Lei Zou’s work was supported by National Science Foundation of China (NSFC) under Grant No. 61370055 and by CCF-Tencent Open Research Fund. M. Tamer Özsu’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 No. 2012-CB316200, Microsoft Research Asia Grant, and a National Grand Fundamental Research 973 Program of China under Grant No. 2012-CB316200, Microsoft Research Asia Grant, and a Google Faculty Award. Dongyan Zhao was supported by NSFC Grant No. 2012-CB316200, Microsoft Research Asia Grant, and a National Grand Fundamental Research 973 Program of China under Grant No. 2012-CB316200, Microsoft Research Asia Grant, and a Google Faculty Award. Dongyan Zhao was supported by NSFC under Grant No. 61727344 and China 863 Project under Grant No. 2012AA011101.

References