Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 5341797, 11 pages
http://dx.doi.org/10.1155/2016/5341797

Research Article

STLIS: A Scalable Two-Level Index Scheme for Big Data in IoT

Yonglin Leng,[1,2] Zhikui Chen,[1] and Yueming Hu[3]

[1]*School of Software Technology, Dalian University of Technology, Dalian 116620, China*
[2]*College of Information Science and Technology, Bohai University, Jinzhou 112100, China*
[3]*College of Natural Resources and Environment, South China Agricultural University, Guangzhou 510642, China*

Correspondence should be addressed to Zhikui Chen; zkchen@dlut.edu.cn

Received 21 August 2016; Accepted 20 October 2016

Academic Editor: Beniamino Di Martino

The rapid development of the Internet of Things causes the dramatic growth of data, which poses an important challenge on the storage and quick retrieval of big data. As an effective representation model, RDF receives the most attention. More and more storage and index schemes have been developed for RDF model. For the large-scale RDF data, most of them suffer from a large number of self-joins, high storage cost, and many intermediate results. In this paper, we propose a scalable two-level index scheme (STLIS) for RDF data. In the first level, we devise a compressed path template tree (CPTT) index based on S-tree to retrieve the candidate sets of full path. In the second level, we create a hierarchical edge index (HEI) and a node-predicate (NP) index to accelerate the match. Extensive experiments are executed on two representative RDF benchmarks and one real RDF dataset in IoT by comparison with three representative index schemes, that is, RDF-3X, Bitmat, and TripleBit. Results demonstrate that our proposed scheme can respond to the complex query in real time and save much storage space compared with RDF-3X and Bitmat.

1. Introduction

The rapid development of the Internet of Thing (IoT) [1] has generated immense sensory data from numerous sensors and devices. In a white paper [2], Cisco predicated that, by 2022, 50 billion things will be connected to the Internet. IDC predicated that 28 billion IoT devices will be installed by 2020 [3]. In facing of such a large number of sensors and devices, IoT data will maintain rapid growth, which poses an important challenge on the storage and quick retrieval of IoT data. RDF (resource description framework) [4], as an effective data representation model, has been applied in many fields including IoT [5]. Therefore, researching the storage and index of large-scale RDF data is very meaningful.

Many works have been engaged in RDF storage and query research. Triple table [6, 7], column storage [8, 9], and property table [10] adopt the alternative storage scheme to accelerate the retrieval of RDF data. But a large number of self-joins and NULL values and scalability constrain the development of these storage systems. RDF-3X [11], Hexastore [12], and SPOVC [13] store the multipermutation of S (subject), P (predicate), and O (object) to match different types of triple patterns. Bitmat [14] and RDFCube [15] use the three dimensions of bit-cube to represent S, P, and O, respectively. Though these methods have a high query efficiency, all of them are at the cost of the storage. Zou et al. [16] proposed a graph-based approach to store and query RDF data (gStore). In order to improve the efficiency of query, gStore creates VS-tree and VS$*$-tree above the RDF graph. TripleBit [17] employs compact technology to store RDF data and introduces two auxiliary index structures to reduce the intermediate results. The index tree and compact technology improve the filter quality and cut down the storage space. However, data filtering and retrieval of gStore and TripleBit are triple-based, which ignore the relation of triple patterns. In order to reduce the number of redundant intermediate results, RP-index (RDF path index) [18] is proposed to efficiently filter data. Wu et al. [19] also used path partitioning to realize the scalable SPARQL query.

To sum up the above discussion, the joins, storage cost, and intermediate results are the main problems in large-scale RDF data storage and index. The tree-based index can effectively filter the redundant data and the path can strengthen the correlation between triples. Therefore, in this

paper, we propose a scalable two-level storage and index scheme based on path partitioning (STLIS), which makes use of the advantages of index tree and the path. The first level of STLIS is used to filter the candidate sets of full path. And the second level is designed to accelerate the match. The main contributions are summarized as follows:

(i) We create a CPTT index based on S-tree to filter the irrelevant RDF data and get the candidate set of full paths. In CPTT index, each leaf node corresponds to a set of full paths, which have the same path tree template.

(ii) During the retrieval process, we design two compressed logical operations, *compressed_AND* and *compressed_OR*, to avoid the decoding operation, thereby improving the efficiency of STLIS.

(iii) We create a HEI index to query the constant predicate path, which can quickly locate the known edge and retrieve it along with the hierarchy of edge backwards or forwards. Through the correlation between triples, HEI dramatically reduce the scale of intermediate results. For the variable predicate path, we design a NP index to convert the variable predicate path into the constant predicate path.

(iv) A set of extensive experiments are executed on a benchmark dataset and real dataset. Results demonstrate that our proposed scheme can respond to the complex query in real time and save much storage space.

The rest of the paper is organized as follows. In Section 2, we review related work about the storage and index of RDF data. Then, in Section 3, we briefly introduce some notations that are closely related to our works, and then we present STLIS index scheme in Section 4. Section 5 provides detailed experimental evaluations compared with the state-of-the-art indexes and Section 6 concludes the paper.

2. Related Work

There are three major kinds of RDF storage and index methods: relational database, triple, and graph.

The management and control technology of relational database is very mature and effective. Many RDF storage systems, for instance, 3-store [6] and Sesame [7], store triples into the relational database. For a SPARQL query, each triple pattern needs to access triple table once, which produces many self-joins and increases query time. In order to improve the retrieval efficiency, many alternative storage schemes are designed to reduce the self-join. Jena2 [10] stores triples into property-class tables that cluster several properties together into a single table. Each property in property-class table is a column and the form is consistent with the relational table. By this way, we can find the triples relating to the same class in a table; therefore, the number of self-joins is reduced dramatically. However, if a SPARQL query involves multiple property-class tables, the merge joins are inevitable. In addition, not all subjects have the same properties in a property-class table, so the property-class table would contain many NULL values. SW-store [8, 9] partitions vertically the triple table into multiple tables according to the predicate. Each table has two columns and all the data in one table have the same predicate. It is easy to query the triple pattern with the given predicate, but this method does not scale well when the predicate is variable.

Triple index scheme is a combination and permutation of S, P, and O. Hexastore [12] creates six different B+tree indexes, SPO, SOP, PSO, POS, OPS, and OSP, and shares common indexes within these 6-way indexes to eliminate storage redundancy. RDF-3X [11] also stores triples into six clustered B+trees above. And, meanwhile, it creates 9 projection indexes: S, P, O, SP, PS, SO, OS, OP, and PO. Instead of storing triples, the projection indexes map search keys to the number of triples. Similar to RDF-3X, instead of 6-way indexes, SPOVC [13] uses 5-way indexes, namely, sIndex (subject), pIndex (predicate), oIndex (object), value, and class, to partition a large triple table into a smaller table. By this way, each triple pattern can be retrieved from a single partition. Bitmat [14] and RDFCube [15] are two kinds of combination indexes based on S, P, and O. Bitmat uses a bit-cube to index the RDF data. The 3 dimensions of bit-cube correspond to S, P, and O, respectively, and each cell corresponds to a unique RDF triple. If the cell is set to 1, the combination of coordinate values of S, P, and O is a given RDF triple; otherwise, the triple does not exist. Similar to Bitmat, RDFCube maps S, P, and O into the 3 dimensions of bit-cube by hash functions. The high efficiency of triple indexes is at the cost of the redundant storage. Though the compression technique is applied to these indexes, the redundancy mode cannot adapt to the data updating and query processing in confronting the large-scale RDF data.

The nature of RDF is a directed graph; therefore, the query can be converted into a subgraph matching problem. Index is an efficient method in the subgraph matching problem, and many indexes based on RDF graph are proposed. GRIN [20] uses a balanced binary tree to index the RDF graph, which groups information around "center" vertices within a given radius. Zou et al. [16] proposed VS-tree and VS$*$-tree index to process both exact and wildcard SPARQL queries by efficient subgraph matching. PIG (parameterizable index graph) [21] is a compact representation of the data graph. Each vertex in PIG corresponds to a group of data graph vertices which have a similar or equal structural "neighborhood." He et al. [22] proposed a bilevel indexing and query processing scheme (BLINKS) for top-k keywords search on graph, which stores summary information at the block level for the search of interblock and more detailed information in each block to accelerate search of intrablock. However, BLINKS only supports the search over node-labeled directed graphs. In order to reduce the number of redundant intermediate results, RP-index (RDF path index) is proposed in [18] to efficiently filter data. TripleBit [17] employs compact technology to store and access RDF data. And, in the query process, TripleBit introduces two index structures to minimize the cost of index selection. To sum up the above discussion, the path, compaction, and index tree are very effective technologies

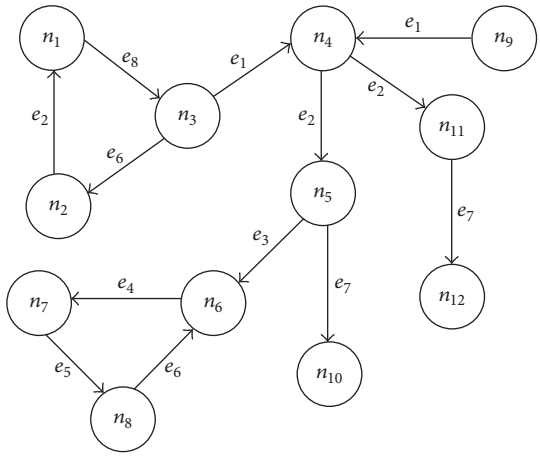


Figure 1: RDF graph.

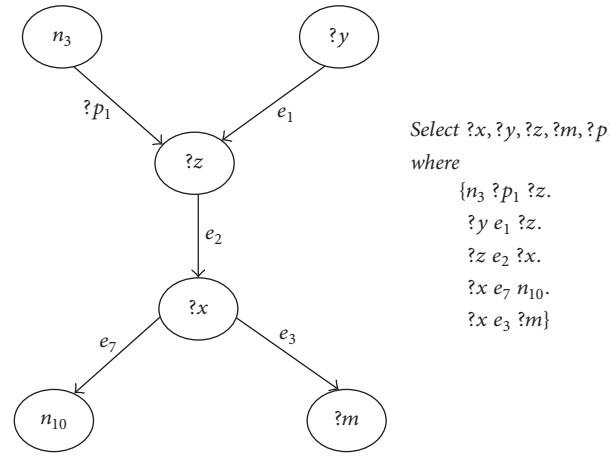


Figure 2: SPARQL query.

in improving the query efficiency and reducing the storage space.

3. Preliminaries

Before going into the details of our scheme, we briefly review some core contexts that are relevant to STLIS.

3.1. RDF Data. In this paper, we use $T = \{t \mid t \in S \times P \times O\}$ to represent a RDF dataset, where S, P, and O are the set of subjects, predicates, and objects, respectively. The corresponding directed label graph of T is denoted as $G = \{V, E, L, f\}$. $V = S \cup O$ is a set of vertices, and $E \subseteq V \times V$ is a set of directed edges from subjects to objects. The label function $f(V \cup E) \rightarrow L$ maps each vertex and edge to a label in L. In the directed label graph, there are two kinds of special vertices: the start vertex and the end vertex. The start vertex refers to the vertex with only outgoing edge or any vertex in a directed cycle in which the directed cycle has no incoming edge, and the vertex with only incoming edge or the start vertex of a directed cycle is the end vertex.

In Figure 1, we use simple notations to substitute URIs and literals. The vertices n_9 and any vertex in the directed cycle (n_1, n_3, n_2, n_1) are the start vertices. The vertices n_{10} and n_{12} and the vertices n_1 and n_6 in the directed cycles (n_1, n_3, n_2, n_1) and (n_6, n_7, n_8, n_6) are the end vertices.

3.2. SPARQL Query. SPARQL is a standard RDF query language [23], which consists of multiple triples and triple patterns. We call the triple including variables the triple pattern. Figure 2 gives an example of SPARQL, where the symbols x, y, z, m, and $p1$ starting with "?" are variables. SPARQL query can also be modeled as graph patterns. In this paper, we use $G_Q = \{V_Q, E_Q, L_Q, f_Q\}$ to describe a SPARQL graph pattern. SPARQL query processing is to find matching triples or subgraphs, in which the information of vertex or edge could substitute for the query variables.

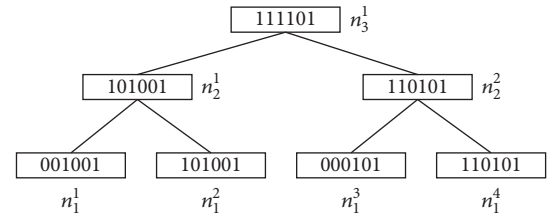
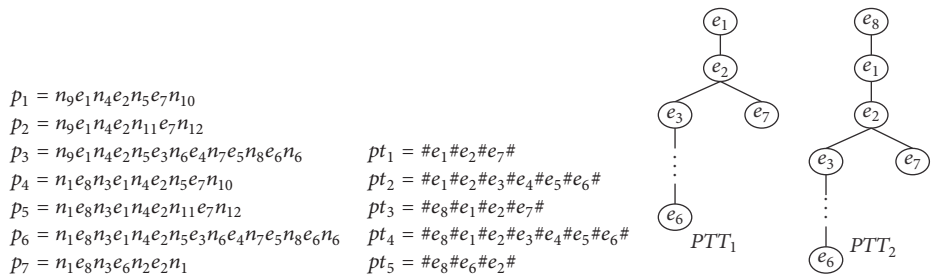


Figure 3: S-tree.

3.3. S-Tree. S-tree is a dynamic balanced multiway signature tree [24], which is similar to B+tree. Each node in S-tree corresponds to one or more signatures. The signature is denoted by a bit-string of fixed length. The bit-string is generated by applying an appropriate hash transformation on the object. The leaf nodes of S-tree are the retrieval objects, and the intermediate nodes are obtained by superimposing the signatures contained in their son nodes. Figure 3 represents a binary S-tree, in which each leaf node corresponds to one signature. n_2^1 is the result of $n_1^1 \mid n_1^2$, where "|" is the bitwise-OR operator. RDF data use Universal Resource Identifiers (URIs) to identify subjects or objects, which are a long string. During the process of retrieval, it is time-consuming to compare strings directly. However, S-tree can use bitwise-AND to realize the comparison quickly. And S-tree can retrieve the object containing variables. Therefore, in this paper, we propose an improved filter index based on S-tree.

4. STLIS

In our index framework, we design a two-level index model. The first level is a filter layer, in which we use CPTT index to filter the RDF data. In order to speed up the efficiency of retrieval and save the storage space, we compress each node in CPTT and give two compressed logical operations, that is, *compressed-AND* and *compressed-OR*, to avoid the decoding operation. The second level is an accurate match layer. We design two indexes, namely, HEI and NP, to assist the retrieval



(a) Full paths (b) Path templates (c) Path tree templates

Figure 4: Full path and path template.

of full path. HEI is fit for the constant predicate path and NP is used in the variable predicate path. In the following sections, we introduce each step in detail. Section 4.1 describes how to get path templates (step 1). Using these path templates, CPTT is created in Section 4.2 (step 2). Then, Section 4.3 discusses how to find the matched path template and the corresponding full path from CPTT (step 3). Finally, we present the index and retrieval of full path in Section 4.4 (step 4).

4.1. Path Template

Definition 1 (full path). Given a RDF graph $G = (V, E)$, if a path $p = v_0e_1v_1e_2v_2 \cdots e_mv_m$ of G satisfies $v_i \in V, e_i(v_{i-1}, v_i) \in E$ and v_0 is a start vertex and v_m is an end vertex. We say that p is a full path. In Figure 4(a), we give all full paths about Figure 1, which start from the start vertices n_1 and n_9.

Theorem 2. *Given a RDF graph $G = (V, E)$, if it is decomposed into a set of full paths $P = \{p_1, p_2, \ldots, p_k\}$ by Definition 1, then each vertex v and edge(u, v) in G can be found in at least one full path.*

Proof. (i) For a vertex $v \in V$ in RDF graph G, there are two kinds of states: start vertex and nonstart vertex. We assume v is a start vertex; then v must belong to a full path. In contrast, if v is a nonstart vertex and there must exist a start vertex s which can reach v, then v belong to a full path starting from s. If there is no source vertex s that can reach v, then v is a start vertex. This contradicts the condition that v is a nonsource vertex. Therefore, for any $v \in V$, it must belong to at least one full path. (ii) For an $edge(u, v) \in E$, if it does not belong to any full paths, then vertex u or v does not exist in any full path. This contradicts (i). So, for any $edge(u, v)$ in G, it can be found in at least one full path. □

Theorem 3. *Given a SPARQL query G_Q that is decomposed into a set of full paths $P_Q = \{p_1^Q, p_2^Q, \ldots, p_n^Q\}$, if G_Q is the subgraph of G, then, $\forall p_i^Q \in P_Q$, there must be at least one full path $p_i \in P$, which satisfies the notion that p_i^Q is a subpath of p_i.*

Proof. Assume there is no full path $p_i \in P$, which satisfies the notion that p_i^Q is a subpath of p_i. (i) If the start vertex v_0 and the end vertex v_m of p_i^Q in G_Q are also the start vertex and the end vertex of G, this will contradict the assumption, because there must be a full path from v_0 to v_m. (ii) If the start vertex v_0 and the end vertex v_m of p_i^Q in G_Q are not the start vertex and the end vertex of G, there must exist a start vertex v_s that can reach v_0 and an end vertex v_e, which is reachable from v_m. Therefore, p_i^Q is a subpath of p_i which starts from v_s and ends up with v_e. □

Definition 4 (path template and homogeneous path). According to Definition 1, a RDF graph G can be decomposed into a set of full paths; that is, $P = \{p_1, p_2, \ldots, p_k\}$. For each path $p_i = v_0e_1v_1e_2v_2 \cdots e_mv_m$, we extract $E(p_i) = \{e_1, e_2, \ldots, e_m\}$ from p_i to make up a schema of path. We say that each schema of path is a path template, and the full paths with the same path template are the homogeneous paths. In Figure 4(b), we give the path templates of Figure 1. The full paths p_1 and p_2 have the same path template pt_1, so we say that p_1 and p_2 are homogeneous paths. The wildcard # represents an instance vertex.

We use a *Depth-First Search* (DFS) algorithm to explore the graph and get the set of full paths and path templates. The algorithm is shown in Algorithm 1. Firstly, the algorithm initializes two stacks: node stack and edge stack (lines (1)-(2)). Secondly, the algorithm finds all full paths and path templates relating to each start node (lines (3)–(18)). For each start node, we push it into node stack (lines (3)-(4)). While the node stack is not empty, we get the top node of stack tn (lines (5)-(6)). If tn does not belong to the set of Sinks, then we use function *neighbor_triples*(tn) to find the unvisited triple associating with tn and push the object and predicate of triple into node stack and edge stack, respectively (lines (13)–(15)). If all of triples associating with tn are visited, we pop up the top node and edge from node stack and edge stack (lines (17)-(18)). Instead, if tn belongs to the set of Sinks, the current node stack and edge stack will be a full path and a path template, respectively. Therefore, we print the node stack and the edge stack and pop up the top node and edge (lines (7)–(12)). The process is repeated, until the node stack is empty.

According to Definition 4, We divide the full paths with the same path template to the same cluster. It is obvious that the number of the full paths will be very huge for a large-scale RDF graph. Therefore, the instance vertex will have many duplicates in different cluster. In order to decrease the duplicates, we propose two merging conditions.

```
Input: Sets Triples, Starts, Sinks
Output: Sets Paths, Path_templates
(1)  Initstack(node);
(2)  Initstack(edge);
(3)  for each s ∈ Starts do
(4)      push(node, s);
(5)      while empty(node) is not NULL do
(6)          tn ← top(node);
(7)          If tn ∈ Sinks then
(8)              Paths ← Paths ∪ print(node);
(9)              Path_templates ← Path_templats ∪ print(edge);
(10)             pop(node);
(11)             pop(edge);
(12)             continue;
(13)         if exist ⟨s, p, o⟩ ∈ neighbor_triples(tn) is not visited then
(14)             push(node, o);
(15)             push(edge, p);
(16)         else
(17)             pop_stack(node);
(18)             pop(edge);
```

Algorithm 1: Full path.

```
Input: matrix pre, set Path_templates, k
Output: set pt_templates
(1)  for each pt ∈ Path_templates do
(2)      initstack(pstack);
(3)      if pt is not merged then
(4)          push(pstack, pt);
(5)          while empty(pstack) is not NULL do
(6)              pt ← pop(pstack);
(7)              for each pt' ∈ path_templates
(8)                  if pre(pt', pt) ≥ k and pt' is not merged then
(9)                      ptt ← merge(pt, pt');
(10)                     push(pstack, pt');
(11)     pt_templates ← pt_templates ∪ ptt;
```

Algorithm 2: Prefix_merge.

Definition 5 (path tree template). Given a set of path templates $PT = \{pt_1, pt_2, \ldots, pt_k\}$, if two or more path templates satisfy the following conditions, one would merge these path templates.

Condition 1. If the path templates share a common prefix, then we merge all these path templates into a path tree template. We use k to represent the length of prefix edge. If the length of prefix edge is equal to or greater than k, then all these path templates are merged. For example, when we set k to 2, the path templates pt_1 and pt_2 would be merged into a path tree template PTT_1, and pt_3 and pt_4 would also be merged into PTT_2. The path tree template is shown in Figure 4(c). It is obvious that the parameter k would affect the number of duplicates of instance vertices. We will discuss this problem in Section 5.4.

Algorithm 2 lists the prefix merge method. The matrix *pre* records the length of common prefix between path templates. We use stack to get each path template set, which satisfies the length of common prefix greater than or equal to k. For each path template pt, if pt is not merged, we find all matching path templates with pt (lines (1)–(7)). And then we merge all these path templates (line (9)). The process is repeated, until all path templates are merged.

Condition 2. Given two path templates pt_a and pt_b, if pt_a is the suffix of pt_b, then we merge pt_a into pt_b. For example, the path template pt_1 is the suffix of pt_3, so pt_1 is merged into pt_3. And PTT_1 and PTT_2 can further be merged into one path tree template PTT_2.

4.2. Compressed Path Tree Template (CPTT) Index. Each path tree template corresponds to a set of full paths. For a SPARQL query, we can retrieve the path template of each query path p_i^Q to get the candidate set of full paths. In this section, we will describe the creation process of CPTT in detail.

Table 1: Statistics of datasets used in experiments.

Dataset	#Vertex	#Edge	#Predicate
LUBM50	1, 706, 230	6, 888, 642	18
LUBM2000	66, 059, 204	276, 345, 040	18
SP^2Bench-100M	548, 826	922, 183	12
UniProt	139,942,781	687,025,165	84

Firstly, we encode each path tree template into bit-string. Secondly, we compress these bit-strings and create a k-way S-tree using these compressed bit-strings. Finally, two logical operations are designed to improve the efficiency of creation and retrieval of index.

Definition 6 (path tree template encoding). Given the predicate set *PRE*, the number of predicates is *plen* and each predicate is assigned to a unique *ID*. The encoding of path tree template is a bit-string of length *plen* and each bit corresponds to a predicate. Assume that PE_i is the set of predicates of path tree template pt_i; if $\exists pre \in PE_i$, then one sets the *eId(pre)*th bit of bit-string to "1," where *eId*(*pre*) is used to get the *ID* of *pre*. In Figure 1, the number of predicates is 8, so the bit-string of pt_1 is "11000010."

Table 1 gives the statistics of predicate about the benchmark datasets and real dataset. The number of predicates is much smaller than the number of vertices and edges. Hence, the encoding scheme is feasible. Also, the bit-string exhibits many consecutive 0 or 1 values, so a compressed method can be used to further decrease the storage space of bit-string. In this paper, we use Run Length Encoding (RLE) to achieve the gap compressed representation of the bit-string. A bit-string of pt_1 will be represented as "[1] 2 4 1 1" in which the first bit of compressed bit-string represents the first bit value and the other bits record the alternating run lengths of 0 and 1.

Given a set of compressed bit-strings, we build an index over all these compressed bit-strings, which is called CPTT index (compressed path tree template index). The build and search process of CPTT is similar to S-tree. Bitwise-AND and bitwise-OR are two basic operations of S-tree. In order to avoid the transformation back and forth between compressed encoding and bit-string, in this paper, we propose the *compressed-AND* and *compressed-OR* operations. Algorithm 3 represents the *compressed-AND* operation. In *compressed-AND*, we adopt the replace-alignment method to compute the result. From the first corresponding compressed bit, we choose the greater compressed bit and replace it with the smaller compressed bit and the difference between the two compressed bits. And then we compute the second bit, and so forth, until all compressed bits have been checked. For example, two compressed strings A = [1] 2 4 2 and B = [0] 3 1 4, in which A-2 is less than B-3, so we replace B-3 with B-2 1. And then B = [0] 2 1 1 4. For the second bit, we compare A-4 and B-1 with the same method, and so on. Due to the fact that OR operation can be denoted as A OR B = NOT(NOT(A) AND NOT(B)) and NOT only needs to change the first value of the compressed bit-string,

```
(1)  sign_a ← Get_sign(A); sign_b ← Get_sign(B);
(2)  while i < A.size() and j < B.size() do
(3)      if Get_str(A, i)-Get_str(B, i) > 0 then
(4)          Replace(A, i, Get_str(B, i));
(5)          Insert(A, i + 1, Get_str(A, i)-Get_str(B, i));
(6)          Change_sign(sign_b);
(7)      else if Get_str(A, i)-Get_str(B, i) < 0 then
(8)          Replace(B, j, Get_str(A, i));
(9)          Insert(B, j + 1, Get_str(B, i)-Get_str(A, i));
(10)         Change_sign(sign_a);
(11)     else
(12)         Change_sign(sign_a);
(13)         Change_sign(sign_b);
(14)     i = i + 1; j = j + 1;
(15) return Merge(A, B);
```

Algorithm 3: *Compressed_AND*(A, B).

therefore, we can use the *compressed-AND* to achieve the result of *compressed-OR*.

We use a bottom-top process to create the CPTT index. Each leaf node of CPTT is a path tree template, and each path tree template corresponds to a set of full paths. Each intermediate node is got by superimposing the compressed bit-strings contained in its son nodes. Figure 5 gives an example of CPTT (in order to explain the retrieval process, Figure 5 is created based on the path template of Figure 4(b)).

4.3. Retrieval of CPTT

Definition 7 (match rule). Given a compressed bit-string of path tree template pt^* and a query path compressed bit-string qt^*, pt^* is a match of qt^*, if and only if *compressed-AND*$(pt^*, qt^*) = qt^*$.

Note that, in the path template, if there are variable edges, we only consider the constant edges. For example, in Figure 2, the path template of query path $n_3?p_1?ze_2?xe_7n_{10}$ is $\#?p_1\#e_2\#e_7\#$. $?p_1$ is a variable; therefore, we discard $?p_1$ and only encode the edge of e_2 and e_7 into the bit-string, that is, 01000010. To find the match path tree template over a given CPTT index, we adopt a top-down search strategy. Due to the fact that the root and intermediate node of CPTT are the summary of their corresponding sons, therefore, if qt^* does not match the root or intermediate node, there must not exist the match path template of qt^*. Consequently, we can filter the path tree template and the corresponding paths. For example, in Figure 5, we want to scan the path template "$\#e_1\#e_2\#e_7\#$" whose compressed encoding is [1] 2 4 1 1. The red dash line represents the retrieval path. We can see that the intermediate node n_3^1 does not satisfy Definition 7. Therefore, all descendant nodes do not need to be retrieved. The final candidate path templates are pt_1 and pt_3. If the path template is "$\#?p_1\#e_1\#e_2\#$" then the candidate results are pt_2, pt_4, pt_1, and pt_3. From these, we can see that the exact path template information and the independent path edge set would get a better filter result. Therefore, the Hamming

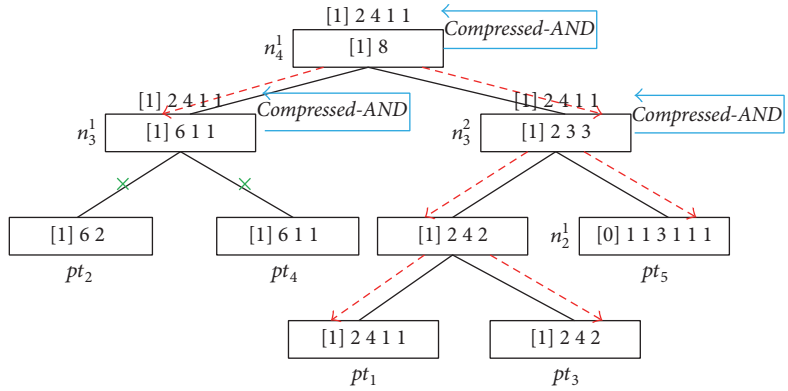


Figure 5: Example of CPTT.

Input: the query path template qt^*, the index of CPTT
Output: a set of match path templates PT

(1) $PT \leftarrow \varnothing$;
(2) $Initstack(ST)$;
(3) $pt^* \leftarrow root(CPTT)$;
(4) $push(ST, pt^*)$;
(5) while ST is not empty
(6) $\quad pt^* \leftarrow pop(ST)$
(7) $\quad$ if $compressed\_AND(pt^*, qt^*) = qt^*$
(8) $\quad\quad$ for each child pt_c^* of pt^* do
(9) $\quad\quad\quad$ if pt_c^* is not a leaf node
(10) $\quad\quad\quad\quad push(ST, pt_c^*)$;
(11) $\quad\quad\quad$ else
(12) $\quad\quad\quad\quad$ if $compressed\_AND(pt_c^*, qt^*) = qt^*$
(13) $\quad\quad\quad\quad\quad PT = PT \cup pt_c^*$;
(14) return PT

Algorithm 4: Retrieval on the CPTT.

distance is used to calculate the similarity between two bit-strings. If the Hamming distance between two bit-strings is the smallest, the two bit-strings will be siblings. The retrieval operation is described briefly in Algorithm 4.

Algorithm 4 uses stack to achieve the retrieval. Firstly, we initialize the stack ST and push the root node into ST (lines (2)–(4)). Next, while ST is not empty, we get the top element pt^* (lines (5)-(6)). We compute the *compressed_AND* between pt^* and qt^* (line (7)). If the result is qt^*, then we check all the child nodes pt_c^* (line (8)). If pt_c^* is not a leaf node, we push pt_c^* into ST, or else we further compute the *compressed_AND* between pt_c^* and qt^*. If the result is qt^*, we merge pt_c^* into PT (lines (9)–(13)). The process is repeated, until the stack ST is empty.

4.4. Index and Retrieval of Full Path. In order to obtain the final query results, we further define a hierarchical edge index (HEI) and a node-predicate (NP) index to assist the retrieval of full path. In this paper, we divide the query path into two categories: the constant predicate path and the variable predicate path. The constant predicate path refers to the notion that one or more predicates in the query path are known. Instead, the variable predicate path refers to the notion that all predicates in query path are unknown. For the constant predicate path, we use CPTT to get the candidate full path sets which contain the known predicates of path edge. The constant predicate path is the most common. We observe a mount query in real RDF datasets, in which most of the edges are known. For example, the edges of query for UniProt datasets in [17] are all constants. So, CPTT index is very effective for pruning the full path.

In the candidate full path sets, we further execute the node match to get the final results. Next, we will introduce the hierarchical edge index (HEI), including the index structure, storage scheme of triples, and the retrieval procedure. HEI contains all edges of path tree template and each edge corresponds to a set of triples and a hierarchical *ID*. As RDF data use Universal Resource Identifiers (URIs) to identify subjects or objects, then, in order to save the storage space and improve the retrieval efficiency, we replace all URIs with *IDs* by mapping dictionaries (see, e.g., [17]). In this paper, we assign *IDs* to subjects and objects with the same *ID* such that subjects and objects having identical values will be treated as the same entity.

Because the query path may be a subset of full paths and some predicates of path edge are unknown, therefore, the retrieval of query path may not start from the root of path tree template. In order to locate the known edge quickly, HEI sequentially indexes each edge of path tree template and assigns a hierarchical *ID*. Meanwhile, each index edge also relates to a set of triples. To save storage space, each triple is encoded with variable-size bits and compressed by RLE. Given a triple, we select the maximum number of bits that encode the *ID* of subject or object as the length of encoding of subject and object. Considering the example of triple $\langle 21, p1, 267\rangle$, the encoding of the subject needs 1 byte and that of the object needs 2 bytes. Therefore, we select 2 bytes as the length of encoding of $\langle 21, p1, 267\rangle$, for example, 0000000000010101, 0000000100001011. Due to the large number of consecutive 0 or 1 values, therefore, we adopt RLE to achieve the gap compressed representation of the bit-string. Figure 6 gives an example about the index and storage of HEI.

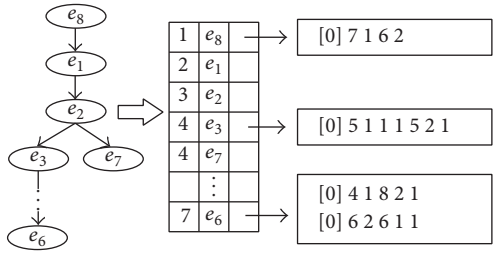


Figure 6: Example of HEI.

For a constant predicate path, the query step is as follows: (1) selecting a constant predicate, locate the predicate edge in HEI; (2) encoding the triple pattern of constant predicate with the same encoding method mentioned above, if a variable exists in the triple pattern, the encoding of corresponding variable is set to "0"; the encoding of the triple pattern also needs to be compressed; (3) use *compressed-AND* to compare the triple pattern with the set of corresponding edge triples; (4) if the match result exists, the retrieval would continue along the path backwards and forwards with the assist of hierarchical *ID*.

Through CPTT, the paths that do not meet the match conditions are filtered. HEI uses the constant predicate to retrieve the unknown information of triple, which provides more related information for the adjacent triples. By this way, the joins among triples decrease dramatically. However, the above method only suits the constant predicate path; in face of the variable predicate path, CPTT is invalid. However, the variable predicate path is extremely rare in the SPARQL of a real dataset. But, in order to ensure the whole efficiency of retrieval, we define an auxiliary index, that is, NP index, to solve this problem.

NP index is a bit-matrix, in which the row corresponds to the set of nodes and the column represents the set of predicates. If a node relates to a predicate, then we set the corresponding bit to be "1." For all the triples of a query path like $\langle s, ?p, ?o\rangle$, $\langle ?s, ?p, o\rangle$, and $\langle s, ?p, o\rangle$, we locate the constant node s or o in bit-matrix to get the related predicate p. Next, all the predicates p in the query path are encoded into a compressed bit-string according to Definition 6. Using this compressed bit-string, we can query other unknown information by CPTT and HEI.

5. Experiments

In this section, we compare STLIS scheme with some popular index schemes, including RDF-3X (v0.3.8), Bitmat, and TripleBit, on both synthetic and real datasets.

5.1. Datasets and Setting. To evaluate the performance of STLIS, two representative RDF benchmarks, LUBM (the Lehigh University Benchmark) [25] and SP^{2}Bench [26], are used in our experiments. The LUBM features a university domain, and the SP^{2}Bench dataset features a DBLP domain. A real dataset, UniProt, is also utilized in our experiments, which is a protein dataset [27]. Table 1 shows the detailed information about each dataset. For query information about LUBM, refer to Appendix A.2 in [28]; for UniProt, see [17]; and for SP^{2}Bench, see [26].

Table 2: Query response time on LUBM50 (seconds).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
RDF-3X	3.09	0.37	2.34	**0.45**	0.47	1.38	3.28
Bitmat	3.51	0.51	2.37	1.03	0.59	2.69	3.95
TripleBit	2.43	**0.25**	1.07	0.86	**0.43**	0.79	0.66
STLIS	**0.38**	0.42	**0.29**	0.78	0.53	**0.19**	**0.57**

Table 3: Query response time on LUBM2000 (seconds).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
RDF-3X	280.65	4.179	153.91	5.036	3.67	10.344	143.35
Bitmat	15.48	0.96	9.56	1.54	**1.24**	5.98	9.78
TripleBit	23.029	3.062	6.38	6.056	5.011	12.105	35.799
STLIS	**6.67**	**0.83**	**4.54**	**1.03**	1.44	**2.37**	**3.38**

STLIS index is implemented using C++ and compiled using G++, with-O2 option for optimization. The experiments are performed on a PC with Intel Xeon at 2.00 GHz × 24, with 20 GB memory, running 64-bit Linux. To account for caching, each of the queries is executed three times consecutively. We take the average result to avoid artifacts caused by OS activity.

5.2. Performance. During the construction process of STLIS, there are 81 species path templates in LUBM dataset. We set the number of prefix edges k to 3 and get 26 kinds of merged path tree templates. We run the same queries of LUBM as [28] did, and the query response times are listed in Tables 2 and 3. We can see that the query response time of STLIS outperforms the other three index structures, especially the complex queries. The primary reasons lie in the filter condition firstly. CPTT is a combination index of path edges. The relation among edges enhances the filter ability. Therefore, a large number of unrelated triples are filtered, which dramatically reduces the query scale.

Secondly, the intermediate result is another important cause. Regarding intermediate results, we refer to both the number of triples that match the query patterns and the data loaded into the main memory during query evaluation [17]. Due to the compressed encoding and compressed operations, our scheme improves the memory utilization and reduces the I/O cost. Furthermore, in the further retrieval of full path, most of the joins among triples happen in the interior of path retrieval, which leads to less intermediate result for final joins.

As shown in Tables 2 and 3, STLIS is very effective on large-scale dataset. The query response time of RDF-3X increases 7 to 90 times from LUBM50 to LUBM2000, especially the complex queries (Q1, Q3). But the maximal change of STLIS is only 17.55 times. The reasons lie in the fact that RDF-3X needs to load more permutation indexes into the memory for scan. Moreover, the load data need to be decompressed. Therefore, the I/O is heavier in RDF-3X and much time is consumed on decompressed data. The

Table 4: Query response time on UniProt (seconds).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
RDF-3X	75.11	1.95	337.28	24.36	15.64	169.76	45.34	28.36
TripleBit	14.27	**1.05**	112.48	6.68	8.29	6.86	13.65	17.34
STLIS	**9.68**	1.34	**13.56**	**4.38**	**3.27**	**4.28**	**5.67**	**1.24**

Table 5: Query response time on SP^{2}Bench-100M (seconds).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
RDF-3X	0.053	0.46	0.072	0.32	0.095	0.27	0.69	0.77
Bitmat	0.028	0.39	0.064	0.26	0.14	0.19	0.53	0.59
TripleBit	0.014	0.27	0.043	0.27	0.078	0.16	0.42	0.44
STLIS	**0.008**	**0.03**	**0.007**	**0.07**	**0.016**	**0.04**	**0.09**	**0.03**

Table 6: Storage space (GB).

	RDF-3X	Bitmat	TripleBit	STLIS
LUBM50	0.35	0.32	0.28	0.22
LUBM2000	13.95	13.64	8.74	8.83
UniProt	33.89	55.83	15.19	17.95
SP^{2}Bench-100M	0.038	0.044	0.034	0.026

queries of Bitmat and TripleBit are all based on triples, which produce more intermediate results than STLIS, so the query performance is also lower than STLIS.

For the UniProt dataset, the triple scale is up to 0.7 billion and the number of prefix edges k is equal to 3. Bitmat cannot get the query results using the current operation environment. Therefore, Table 4 only lists the comparison among RDF-3X, TripleBit, and STLIS methods. We can see that STLIS outperforms RDF-3X and TripleBit, especially the complex query. The reason lies in the fact that STLIS decomposes the query into paths. Each path edge set is seen as a filter condition. Compared with the triple-based filter, STLIS gets a higher quality candidate set. In the candidate set, the joins between triples only occur in the interior of a query path, which effectively reduce on the scale of intermediate results. Therefore, our query performance is superior to other methods.

We also execute the queries on SP^{2}Bench dataset. The representative queries in [26] are selected to evaluate the query response time. The same as LUBM, we also set k to 3. The experiment results are listed in Table 5. Due to the fact that SP^{2}Bench dataset is very small, the intermediate results of most of the queries can reside in memory directly; therefore, the advantage of STLIS is not obvious.

5.3. Storage Space. We compare the storage space of STLIS with RDF-3X, Bitmat, and TripleBit. The total space cost refers to the size of the whole database and indexes. Besides Bitmat, the other three storage spaces all include the dictionary facility. The detailed comparison is listed in Table 6. We can see that STLIS outperforms RDF-3X in all datasets.

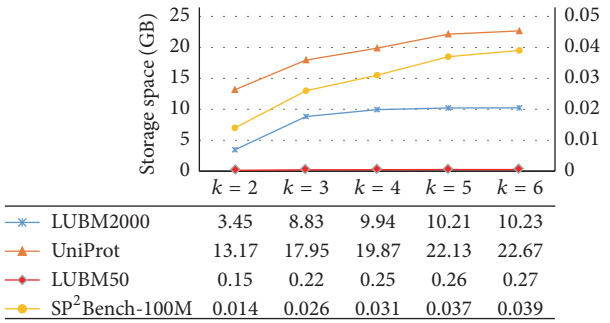


Figure 7: The comparison of storage spaces in parameter k.

The reason is that RDF-3X creates six clustered indexes and 9 aggregate indexes. The high efficiency of RDF-3X is at the cost of space. In addition, with the data scale growth, the storage of RDF-3X and Bitmat grows faster than STLIS. Thus, we conclude that STLIS is more suitable for large-scale dataset. In this paper, we adopt the same dictionary mapping algorithm as TripleBit. As the predicate numbers of LUBM and SP^{2}Bench are very small, in consequence, the merged quality of path template is very high, leading to a smaller duplicate. So, STLIS is comparable with TripleBit. However, for UniProt, the storage space of STLIS is higher than TripleBit.

5.4. Parameter Analysis. In this section, we evaluate the effect of parameter k. In the process of merge path template, we use k to represent the number of common prefix edges. If the number of prefix edges is equal to or greater than k, then all path templates including common prefix edges are merged.

We vary k from 2 to 6 to test the change of storage space. Figure 7 shows that the storage spaces of all datasets increase when k becomes larger. This is because the larger the value of k is, the smaller the common prefix path templates are, therefore leading to the increase of duplicates. However, with the increase of k, the fluctuation of storage space becomes smaller. As shown in Figure 7, while the value of k is equal to 4 or 5, the storage space tends to stability for LUBM and SP^{2}Bench. But, for UniProt, the range is between 5 and 6. The reason is that the number of predicates of UniProt is larger than other datasets.

Figure 8 is the comparison of query response times. The experiment result demonstrates that k has an optimum value which usually is not the biggest or the smallest in query response time. For example, in LUMB and SP^{2}Bench dataset, k is equal to 3 and the value of k is 4 in UniProt. The primary reasons lie in the fact that if k is set to a small value, a large number of path templates would be merged into a path tree template. The candidate set of full paths becomes very large, which would increase the query time. Instead, we select a large value, though the candidate set is smaller than the above selection method, but the increase of duplicates also consumes the query time.

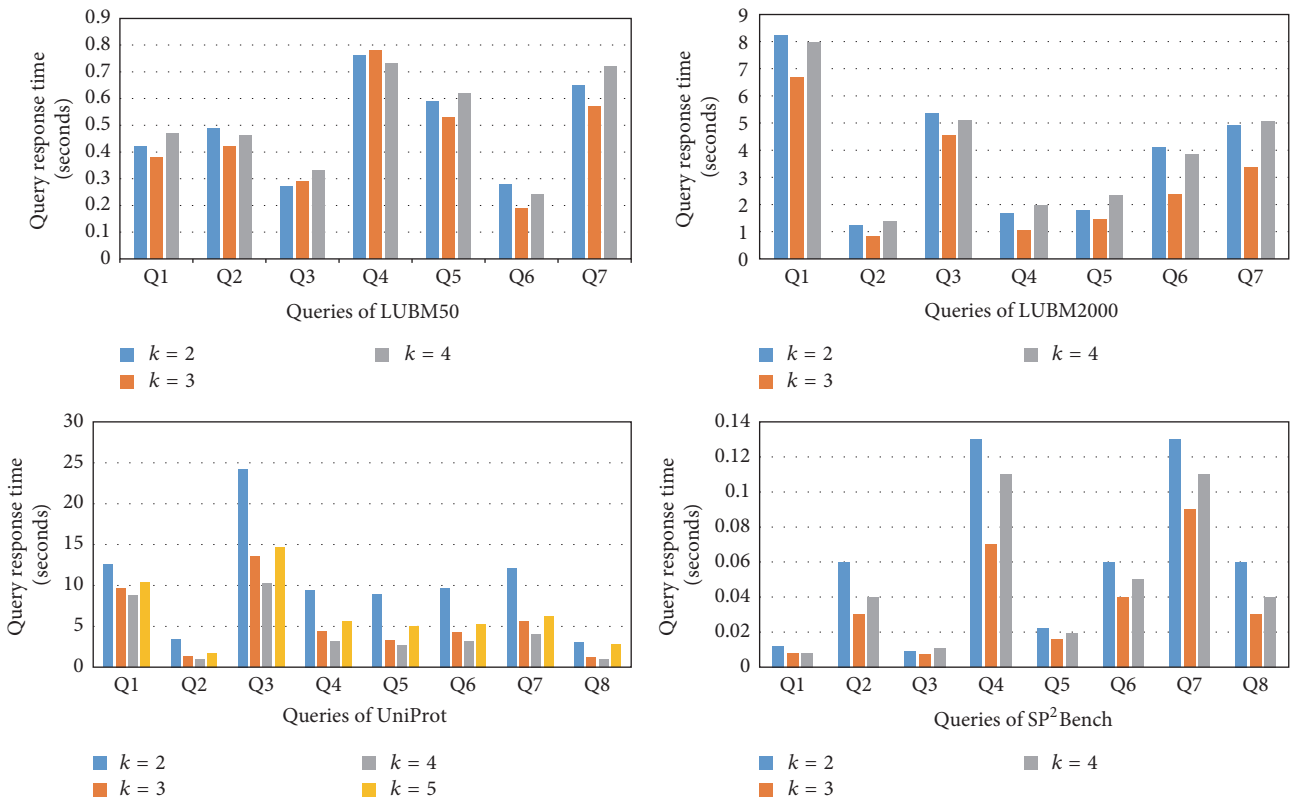


Figure 8: The change of query response time in parameter k.

6. Conclusion

In this paper, we presented a scalable two-level index scheme (STLIS) for big data in IoT. The first level is a filter layer, in which we used CPTT index to get the candidate set of full paths. In order to speed up retrieval and save storage space, each node of CPTT was a compressed bit-string. And, to avoid the decoding operation, we gave two compressed logical operations, that is, *compressed-AND* and *compressed-OR*. The second level was an accurate match layer. In this layer, we designed two auxiliary indexes, that is, HEI and NP, to assist the retrieval of full path. HEI fitted for the retrieval of constant predicate path and NP was used in variable predicate path. Experimental results demonstrated that our proposed scheme can respond to the complex query in real time, and it was effective to reduce the storage space by encoding the *ID*s of subject and object with variable-size bits and compressed technologies.

Furthermore, distributed index and query is the most effective scheme to deal with the big increasing number of IoT data. Our index scheme is very scalable for the distributed environment. CPTT index can be partitioned into several subtrees; each subtree corresponds to a set of leaf nodes and each leaf node is related to a set of full paths. Those subtrees can be distributed to different computing nodes of a compute cluster. Thus, queries can be executed in parallel using a distributed framework.

Competing Interests

The authors declare that there are competing interests regarding the publication of this paper.

Acknowledgments

This work is partially supported by the State Key Program of National Natural Science of China under Grant U1301253, the Science and Technology Planning Key Project of Guangdong Province under Grant 2015B010110006, and the National Natural Science Foundation of China under Grant 61672123.

References

[1] D. Singh, G. Tripathi, and A. J. Jara, "A survey of Internet-of-Things: future vision, architecture, challenges and services," in *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT '14)*, pp. 287–292, March 2014.

[2] J. Bradley, J. Barbier, and D. Handler, "Embracing the Internet of everything to capture your share of $14.4 trillion," White Paper, Cisco, 2013.

[3] IDC Market in a Minute: Internet of Things, http://www.idc com/downloads/idc_market_in_a_minute_iot_infographic.pdf.

[4] RDF, http://www.w3.org/TR/rdf-concepts/.

[5] X. Su, H. Zhang, J. Riekki et al., "Connecting IoT sensors to knowledge-based systems by transforming SenML to RDF," *Procedia Computer Science* , vol. 32, pp. 215–222, 2014.

[6] S. Harris and N. Gibbins, "3store: efficient Bulk RDF Storage," in *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems*, pp. 1–15, 2003.

[7] J. Broekstra, A. Kampman, and F. van Harmelen, "A generic architecture for storing and querying rdf and rdf schema," in *The Semantic Web—ISWC 2002: 1st International Semantic Web Conference Sardinia, Italy, June 9–12, 2002 Proceedings*, vol. 2342 of *Lecture Notes in Computer Science*, pp. 54–68, Springer, Berlin, Germany, 2002.

[8] D. J. Abadi, A. Marcus, S. R. Madden et al., "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pp. 411–422, VLDB Endowment, September 2007.

[9] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "SW-Store: a vertically partitioned DBMS for semantic web data management," *The VLDB Journal*, vol. 18, no. 2, pp. 385–406, 2009.

[10] J. J. Carroll, I. Dickinson, C. Dollin et al., "Jena: implementing the semantic web recommendations," in *Proceedings of the 13th ACM International World Wide Web Conference on Alternate Track Papers & Posters (WWW '04)*, pp. 74–83, New York, NY, USA, May 2004.

[11] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[12] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[13] K. Mulay and P. S. Kumar, "SPOVC: a scalable RDF store using horizontal partitioning and column oriented DBMS," in *Proceedings of the 4th International Workshop on Semantic Web Information Management (SWIM '12)*, ACM, May 2012.

[14] M. Atre, J. Srinivasan, and J. Hendler, "Bitmat: a main-memory bit matrix of RDF triples for conjunctive triple pattern queries," in *Proceedings of the International Conference on Posters and Demonstrations*, vol. 401, pp. 1–2, Aachen, Germany, 2007.

[15] A. Matono, S. M. Pahlevi, and I. Kojima, "RDFCube: a P2P-based three-dimensional index for structural joins on distributed triple stores," in *Databases, Information Systems, and Peer-to-Peer Computing: International Workshops, DBISP2P 2005/2006, Trondheim, Norway, August 28-29, 2005, Seoul, Korea, September 11, 2006, Revised Selected Papers*, vol. 4125 of *Lecture Notes in Computer Science*, pp. 323–330, Springer, Berlin, Germany, 2007.

[16] L. Zou, J. Mo, L. Chen et al., "gStore: answering SPARQL queries via subgraph matching," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[17] P. Yuan, P. Liu, B. Wu et al., "TripleBit: a fast and compact system for large scale RDF data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 2013.

[18] K. Kim, B. Moon, and H.-J. Kim, "R3F: RDF triple filtering method for efficient SPARQL query processing," *World Wide Web*, vol. 18, no. 2, pp. 317–357, 2013.

[19] B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin, "Scalable SPARQL querying using path partitioning," in *Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE '15)*, pp. 795–806, IEEE, Seoul, South Korea, April 2015.

[20] O. Udrea, A. Pugliese, and V. S. Subrahmanian, "GRIN: a graph based RDF index," in *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI '07)*, vol. 1, pp. 1465–1470, July 2007.

[21] T. Tran and G. Ladwig, "Structure index for RDF data," in *Proceedings of the Workshop on Semantic Data Management (SemData@ VLDB '10)*, 2(010), 2010.

[22] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: ranked keyword searches on graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*, pp. 305–316, ACM, Beijing, China, June 2007.

[23] SPARQL, http://www.w3.org/TR/rdf-sparql-query/.

[24] U. Deppisch, "S-tree: a dynamic balanced signature index for office retrieval," in *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '86)*, pp. 77–87, San Diego, Calif, USA, October 1986.

[25] Y. Guo, Z. Pan, and J. Heflin, "LUBM: a benchmark for OWL knowledge base systems," *Web Semantics*, vol. 3, no. 2-3, pp. 158–182, 2005.

[26] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: a SPARQL performance benchmark," in *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE '09)*, pp. 222–233, IEEE, Shanghai, China, April 2009.

[27] Uniprot RDF, http://www.ebi.ac.uk/uniprot/.

[28] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix 'bit' loaded: a scalable lightweight join query processor for RDF data," in *Proceedings of the 19th International World Wide Web Conference (WWW '10)*, pp. 41–50, Raleigh, NC, USA, April 2010.