PathBit: A Bit Index Based on Path for Large-Scale Knowledge Graph

Yonglin Leng(✉), Peiyi Qu, Ying Guo, and Chaoliang Xi

College of Information Science and Technology, Bohai University, Jinzhou 121000, China
lengyonglin@qq.com

Abstract. As the latest achievement of symbolism, knowledge graph is an important cornerstone of artificial intelligence. In order to better manage the knowledge graph, RDF triples have been used to represent knowledge graph. The rapid growth of data brings great challenges to knowledge graph storage and quick retrieval. Among them, self joins, high storage cost and intermediate results are the main problems. In this paper, we propose a bit index structure based on path (PathBit) for large scale knowledge graph. PathBit includes an index based on predicate path tree (IPT) and a k^2-tree index (k^2TIP) according to the hierarchy of each predicate path tree. IPT is in charge of the filter of complete path set. k^2TIP according to the hierarchy of each predicate path tree to realize fast association matching of known predicate path triples. Meanwhile, the compression mechanism is used to implement the compressed storage and retrieval algorithm of triples. In addition, two auxiliary indexes: SP and OP are added to assist predicate path retrieval. Finally, we conduct a series of experiments on two representative datasets and compare the results with RDF-3X, Bitmat and TripleBit. Results indicate that PathBit can achieve better response time on complex queries and has greater advantages in storage space compared with RDF-3X and Bitmat.

Keywords: Knowledge Graph · Index · Predicate Path · Compressed storage

1 Introduction

As the supporting foundation of AI, knowledge graph shows more and more value in semantic search, intelligent question answering, data analysis, natural language processing, vision understanding and IoT. In order to better manage the knowledge graph, RDF triples have been used to represent knowledge graph. The rapid growth of RDF data also bring great challenges to query. SPARQL is the most widely used query language in RDF data query [1, 2]. A SPARQL query includes many triple patterns, which can also be described as a directed query graph. The query graph generally consists of four basic sub graphs: star, chain, ring and tree topology [3, 4]. The basic sub graph has a lot of connections. These connection relations can be divided into chain and star relations. The chain relation refers to the subject of triple pattern is the object of another triple pattern. Star relation refers to a group of triple patterns with the same subject or object. In these basic sub graphs, the chain relation is an important structure in SPARQL. Because the ring and tree query all contain chain structure.

V. C. M. Leung et al. (Eds.): Qshine 2023, LNICST 573, pp. 181–195, 2024.
https://doi.org/10.1007/978-3-031-65126-7_18

The rapid increase of RDF data brings great challenges to traditional data storage, index and query. The triple table, vertical partition and attribute table use an alternative relational storage mode and mature management mechanism of relational database to accelerate data retrieval. However, these relational data models could not fully reflect the logical structure of RDF data [5].

Some native storage systems, such as RDF-3X [6], Hexastore [7] and SPOVC [8], store multiple copies of data according to different combinations of subject, predicate and object to assist in generating better query plan. Although the query efficiency is improved, these systems are at the expense of storage space. Bitmat [9] and RDFcube [10] use three-dimensional matrix to store triples, and divide the three-dimensional matrix into two-dimensional matrices along a certain dimension. For each two-dimensional matrix, D-gap compression method is used for row compression storage. However, in the face of large-scale data, it is difficult for Bitmat to load all indexes into memory at one time. Triplebit [11, 12] reduces the storage scale of RDF-3X, and only stores two combinations of subject and object (SO) and object and subject (OS) based on predicate. At the same time, Triplebit establishes the corresponding index according to the predicate and realizes the compressed storage. For a SPARQL query, Triplebit generates the corresponding query plan according to certain heuristic rules, and dynamically modifies the query plan to reduce the intermediate results.

All these storage systems view triple mode as retrieval unit to realize data retrieval. Through certain query plan and optimization technology, these storage systems reduce intermediate results, and realize fast connection of intermediate results. These indexes are based on triples and do not consider the structure and semantics of RDF graphs. As mentioned above, triple patterns in SPARQL queries have certain connection relations, which not only reflect the structural information of RDF graphs, but also reflect certain semantic relations. This paper proposes a bit index structure based on path (PathBit) for large scale RDF Graph. The major contributions include:

(1) Taking the complete path from source to sink point in RDF graph as the structure object, we create the bit index based on predicate path tree to realize the retrieval and filtering mechanism, and reduce the connection scale of intermediate results in triple pattern matching;
(2) For each complete path tree, we will create a k^2-tree index (k^2TIP) according to the hierarchy of each predicate path tree to realize fast association matching of known predicate path triples;
(3) We use k^2-tree compression mechanism to implement the compressed storage and retrieval algorithm of triples. At the same time, two auxiliary indexes: SP and OP are added to assist predicate path retrieval.

The other parts of the paper are summarized as follows: part two introduces the relate works; part three describes the design scheme of PathBit in detail; finally, experiments verify the performance of PathBit and draw the conclusion.

2 Related Work

In order to improve the retrieval efficiency, researchers have conducted extensive research on the storage and index of RDF. This paper analyzes the current research status from three different perspectives.

RDF storage and index technology based on relationships utilizes relational database query technology to convert SPARQL queries into SQL to realize data retrieval. 3-Store [12] and Sesame [13] all use triple tables. Due to all data exists in a large table, SPARQL queries are easy to result in many self joins and decrease the query efficiency. Jena2 [14] uses an attribute table, which greatly reduces self joins and merge operations. But not all objects have the same properties, which leads to a large number of empty values. In addition, a large number of multi-valued attributes can also generate more multi-valued dependencies. Therefore, the attribute table is not a universal storage model. SW-store [15] decomposes triples based on the predicate, storing triples with the same predicate in the same table. For the two columns table, a subject based clustered index can be created to achieve rapid subject localization. This scheme not only reduces the merging operation of the same predicate, but also avoids the control problems caused by the attribute table.

Triple index scheme is a combination and permutation of S, P and O. RDF-3X [6] stores all permutations and combinations of subject, predicate and object on a B+ tree, respectively. Moreover, RDF-3X also combines two or single elements to directly form a clustered index. Similar to RDF-3X, Hexastore [7] also establishes six indexes based on the triple table. The difference is that in the establishment process of index, Hexastore considers the order relationship between the subject, predicate and object. Meanwhile, Hexastore will reduce the redundancy of memory by sharing index lists. SPOVC [8] creates five index types based on subject, predicate, object, object data types, and triple classes. Each index type was horizontally segmented according to certain rules, which is effective for the query of range or rule expressions.

Bitmat [9] and RDFcube [10] map S, P and O into a three-dimensional space to form a three-dimensional matrix. Each element in the matrix corresponds to a triple. Bitmat is a memory based bit matrix primarily used to handle concatenation operations in triple pattern. Although these two index types utilize bit technology to achieve high compression of triples, they face large-scale data, especially Bitmat, which makes it difficult to load the index into memory at once.

RDF itself is a directed graph, so SPARQL query can be seen as a sub graph matching problem. GRIN [17] indexes RDF graphs with a balanced binary tree. By utilizing the distance conditions, it can quickly filter the data that does not meet the criteria. But GRIN index has poor scalability. Zou et al. [18, 19] proposed VS-tree and VS*-tree index to handle precise and wildcard SPARQL queries. PIG [20] (Parameterized Index Graph) index corresponds to a set of vertices with similar or identical neighborhood structures in the original data graph. PIG first retrieve edges that are homomorphic to the edges in the query graph to form a set of candidate edges, and then perform join operations in the set of candidate edges. He et al. [21] proposed a two-layer index scheme (BLINKS) for searching the top-k keywords on a graph, which only supports searching on node labeled directed graphs. In order to reduce redundant intermediate results, RP-index [22] creates a path based index to index the RDF graph in-edge. During the executive process,

filtering operations are used to filter out irrelevant data in the input triples. TripleBit [11] vertically divides the triple matrix based on predicates, and sorts triples with the same predicate in the order of subject or object. During the query process, two index structures was introduced to minimize the cost of index selection. In summary, path, compression and index tree are very effective techniques for improving query efficiency and reducing storage space.

3 PathBit

PathBit includes an index based on predicate path tree (IPT) and a k^2-tree index (k^2TIP) according to the hierarchy of each predicate path tree. IPT is in charge of the filter of complete path set, which related to the retrieval path. k^2TIP according to the hierarchy of each predicate path tree to realize fast association matching of known predicate path triples. Meanwhile, the compression mechanism is used to implement the compressed storage and retrieval algorithm of triples. In addition, two auxiliary indexes: SP and OP are added to assist predicate path retrieval.

3.1 Complete Predicate Path

An RDF database is a set of RDF triples, we use $T = \{t \mid t \in S \times P \times O\}$ to describe the dataset, where S, P, O are the set of subjects, predicates and objects, respectively.

Definition 1 (Path). Given an RDF $G = (V, E, L)$, a path is a set of ordered vertices, denoted by $R = (v_0 v_1 v_2 \ldots v_m)$, $\forall k \in [0, m-1]$, $<v_k, v_{k+1}> \in E$.

Definition 2 (Complete Path). For any path R in RDF graph, if v_0 is a source vertex and v_m is a sink vertex, we say that R is a complete path. We use $CPath = \{R_1, R_2, \ldots, R_m\}$ *to* denote a set of RDF complete paths.

Theorem 3. Given an RDF G, $\forall v \in V$ and $e(u, v) \in E$ must belong to at least one complete path.

Proof: (1) Assuming SV is the set of source vertices. For any vertex v in graph G, there are two states. The first is $v \in SV$. If $v \in SV$, because any complete path starts from a source vertex, v must exist in a complete path. The second is $v \notin SV$. If $v \notin SV$, then there must be a source vertex s, so that s to v can be reached, that is, the vertex v belongs to a complete path whose source vertex is s. If s doesn't exist, then v becomes the source vertex, which conflicts with the condition. Therefore, for any vertex v in set V must belong to at least one complete path. (2) For any edge $e(u, v) \in E$ in graph G, if it does not belong to any complete path, then the two vertices u or v do not exist in any complete path, which is in contradiction with that any vertex $v \in V$ belongs to at least one complete path. Therefore, any edge $e(u, v) \in E$ belongs to at least one full path.

Theorem 4. Given a SPARQL query G_q, according to the Definition 4, G_q is decomposed into a set of complete query paths. We use $QCPath = \{R_1^q, R_2^q, \ldots R_n^q\}$ to represent it. If G_q is a subgraph of G, then $\forall R_i^q \in QCPath$, there must exist at least one complete path R_i, satisfying R_i^q is a subpath of R_i.

Proof: Suppose there is no complete path $Ri \in CPath$, satisfying R_i^q is a subpath of R_i. (1) If the source vertex v_0 and sink vertex v_m of the complete query path R_i^q in G_q is also the source and sink vertices of G, then this will conflict with the hypothesis, because there must be a complete path between v_0 and v_m. (2) If the source vertex v_0 and sink vertex v_m of R_i^q in G_q is not the source and sink vertices of G, then there must be a source vertex v_s, which v_s to v_0 is reachable and there is also a sink vertex v_e, which v_m to v_e is also reachable. That is to say, there is at least one complete path from v_s to v_e and v_0 to v_m is a subpath of the whole path, which contradicts the hypothesis. To sum up, the hypothesis does not hold, that is, there is at least one complete path $R_i \in CPath$, satisfying R_i^q is a subpath of R_i.

Definition 5 (Predicate Path). Given an RDF graph G, according to the Definition 4, G is decomposed into a set of complete paths. We use $CPath = \{R_1, R_2, \ldots, R_m\}$ to represent it. For any path R, Extract the edge information of the path to construct a summary path $E(R_i) = \{e_1, e_2, \ldots, e_m\}$, then $E(R_i)$ is called predicate path.

Definition 6 (Isomorphism Path). If the complete paths have the same predicate path, they are called to be isomorphic paths.

Obviously, after decomposing the RDF graph into the set of complete path, a large number of vertices and edges repeatedly appear in different complete paths, which puts a lot of pressure on data storage. However, many complete paths have similar predicate path, that is, many vertices information is the same. If these complete paths are divided into the same class, the number of copies of vertices will be reduced. So we define the following two conditions to merge predicate path.

Definition 7 (Predicate Path Tree). If two or more predicate paths meet the following conditions: (i) Two or more predicate paths have a common prefix and the length of the edge of the common prefix is greater than or equal to a threshold. (ii) A predicate path is the suffix of another predicate path. We will merge these paths into one predicate path tree. The predicate path tree is denoted as *PPtree.*

3.2 Index of Predicate Path Tree

Each predicate path tree corresponds to a complete path set. For a SPARQL query, it is decomposed into several query paths in the same way. We can obtain the complete path set corresponding to each query path by retrieving the predicate path tree. In order to quickly locate the complete path set of the query paths, we create an index based on predicate path tree (IPT). The establishment process of IPT mainly include three steps: the first is to code the predicate path tree, the second is the construction process of IPT and the last is how to retrieve IPT.

Definition 8 (Encoding Predicate Path Tree). Assign a unique id to each predicate in L in order. Obviously, the maximum id is the number of elements in L, which is represented by ℓ. The encoding of predicate path tree is a bit string of length ℓ, and each

bit of the bit string corresponds to a unique predicate. Supposing $E(ppt_i)$ is the predicate set of a predicate path tree, where ppt_i is the ith predicate path tree in *PPtree*. If $\exists pre \in ppt_i$, set the *eId* (*pre*) bit of the bit string to 1, where *eId* (*pre*) represents the id of the predicate *pre*.

The establishment of IPT is based on the bottom-up process. Each leaf node of IPT corresponds to a predicate path tree, and each path template tree corresponds to a full path set. Non-leaf nodes of IPT are obtained by performing a logical 'OR' operation on their sons.

3.3 Retrieval of Predicate Path Tree

When retrieving the query predicate path tree on the IPT index tree, we encode the query path tree in the same way according to Definition 8. If the query predicate path tree includes predicate variables, the code of predicate variable is set to 0. Then use the top-down method to search for the match paths in IPT index tree. If the query predicate path tree and the node of IPT meet the matching principle of Definition 9, the search continues, otherwise the subtrees corresponding to the unmatched node on IPT index tree are pruned.

Definition 9 (Matching Principle). Given a bit string of predicate path tree bit string *ppt** and a query path bit string *qtt**, if *ptt** matches *qtt**, if and only if the logical 'and' operations satisfy AND(*ptt**, *qtt**) = *qtt**.

3.4 Match of Complete Path

Retrieved results in IPT are a candidate set of complete path. In order to get the final query results, it is necessary to accurately match the candidate path set. In this section, we will create a k^2-tree index (k^2TIP) for the corresponding complete path collection according to the hierarchy of each predicate path tree. The k^2TIP adopts two stage compression modes. Figure 1 shows the k^2-tree index structure. The k^2TIP index contains each edge in the predicate path template tree and its corresponding hierarchical ID information, and each edge points to a storage area, which is used to store the triple set associated with the edge in the whole path. Since all triples of this set have a common predicate, we use k^2-tree [23] structure to store triples for each triple set, and compress triples on this basis.

Each predicate in the predicate path tree corresponds to a triple set, and these triples share the same predicate. In order to reduce the storage space, we compress each triple set, and the compression method adopts k^2-tree.

k^2-tree first uses a two-dimensional bit matrix to establish the corresponding relationship between subject and object. If there is a corresponding relationship between the subject and the object, the corresponding bit is set to 1, otherwise it is 0. A large number of subjects and objects are not related, so the bit matrix is a sparse matrix. Therefore, we divide the bit matrix into k^2 sub matrices, and each sub matrix corresponds to a sub

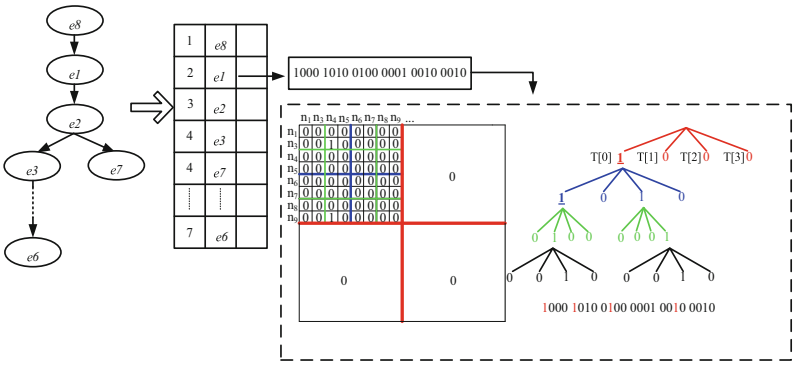


Fig. 1. Example of k^2TIP

node of the root node in k^2 tree. If the bit element of the sub matrix contains 1, then the corresponding sub node of k^2 tree is 1, otherwise the sub node is 0. After the first level node is created, the matrix corresponding to the node with the value of 1 will continue to be divided in the same way until the sub matrix is 0 or there are only k^2 bits in the sub matrix.

The generated k^2-tree connects nodes corresponding to 0 or 1 from top to bottom and left to right to form a compressed bit string. Figure 1 describes the generation process of bit matrix, k^2-tree and compressed bit string of a predicate path tree. For a triple set of a predicate, it is usually necessary to search all the objects corresponding to a known subject, or to search all the subjects corresponding to the known subject. As shown in Fig. 2 shows all objects associated with the subject n_3, where the id of n_3 is 2 and the value of k is 2. In order to get the all objects associated with n_3, we have to find all the columns with cell 1 in the row of n_3 in the matrix. The specific steps are as follows:

Step 1: starting from the root node of k^2-tree, set its position *pos* as 0, and its four sub nodes corresponding to the four sub matrices respectively. The sub matrix corresponding to the first two sub nodes intersect with the row of n_3, while the other two child nodes has no association with n_3. So we only need to consider the first two sub nodes. Set the two sub matrices to T[0] and T[1]. T[1] sub matrix is 0, which means that there is no object associated with n_3 in T[1]. In order to get the number of the column in which the object associated with n_3, it is necessary to record the starting position of the corresponding column of the incidence sub matrix when locating the incidence sub matrix. For example, the starting position of T[0] sub matrix is 0.
Step 2: in the compressed bit string, the starting position *pos* of sub node corresponding to T[0] is 4. The corresponding bit of four sub matrices of T[0] is '1010'. The id of n_3 indicates that the sub matrix associated with it is T[0][0] and T[0][1], where T[0][0] is 1 and T[0][1] is 0. And the starting position of T[0][0] sub matrix is 0.
Step 3: according to this method, continue to search down until the leaf node. If the leaf node is 1 and satisfies the association with n_3, then the column corresponding to the node is the object associated with n_3.

According to Theorem 4, given a SPARQL query G_q, G_q is decomposed into a complete path set $QCPath = \{R_1^q, R_2^q, \ldots R_n^q\}$. If G_q is a subgraph of G, then $\forall R_i^q \in QCPath$, there must exist at least one complete path R_i, satisfying R_i^q is a subpath of R_i. Therefore, a SPARQL query need to be decomposed into multiple search paths from the source vertex to the sink vertex. The decomposition principle is to follow the full coverage of vertices and edges. That is, starting from any source vertex, if the decomposed full path already contains all the edges in the query, the decomposition ends. Because the complete path decomposition already includes all possible complete paths, there must be a complete path corresponding to it.

According to whether the predicate path contains a constant, the decomposed search path can be divided into two categories: constant predicate path and variable predicate path. Constant predicate refers to the path containing one or more known predicates, while variable predicate refers to the path in which all predicates are unknown.

The analysis result shows that most of predicate paths of SPARQL queries are constant predicate path. For constant predicate paths, the retrieval is performed on the IPT index to obtain the candidate complete paths containing known predicates. Then according to the connection relationship of adjacent predicates, they are divided into six types as shown in Table 1. When performing the retrieval, the type of adjacent predicate is judged from the source vertex in turn, and is executed in the order from low to high.

Table 1. Connection Types of triple pattern1

Category	Type
1	$s\,p_1\,?x \bowtie ?x\,p_2\,o$
2	$?s\,p_1\,?x \bowtie ?x\,p_2\,o$
3	$?s\,p_1\,?x \bowtie ?x\,p_2\,?o$
4	$s\,p_1\,?x \bowtie ?x\,?p_2\,o$
5	$?s\,?p_1\,?x \bowtie ?x\,p_2\,o$
6	$?s\,?p_1\,?x \bowtie ?x\,p_2\,?o$

When the search path is a variable predicate path, the adjacent predicate connections can be divided into three types as shown in Table 2. Since k^2TIP is only applicable to the case that there is a constant predicate in the retrieval path. For the variable predicate path, not only IPT is invalid, but k^2TIP is also invalid. In order to ensure the validity of the index, we design two auxiliary indexes, namely SP and OP, to solve this problem. SP and OP store all predicates corresponding to each subject or object, respectively. SP and OP indexes adopt the compressed representation and retrieval method proposed in Triplebit, which will not be explained in detail here.

Table 2. Connection types of triple pattern2

Category	Type
1	$s\ ?p_1\ ?x \bowtie ?x\ ?p_2\ o$
2	$?s\ ?p_1\ ?x \bowtie ?x\ ?p_2\ o$
3	$?s\ ?p_1\ ?x \bowtie ?x\ ?p_2\ ?o$

4 Experiments

In this section, PathBit indexing scheme is tested on synthetic and real datasets.

4.1 Datasets and Setting

Table 3. Test datasets

Data set	Vertex	Triple	Predicate
LUBM50	1,706,230	6,888, 642	18
LUBM2000	66,059, 204	276, 345,040	18
SP2Bench	56,125,032	113,246,165	22
Uniprot	139,942,781	687,025,165	84

In the experiment, two synthetic datasets LUBM and Sp^2Bench were selected. The LUBM features a university domain, and the SP^2Bench dataset features a DBLP domain [24, 25]. In our experiments, we also use a protein dataset Uniprot [26] (Table 3).

PathBit index is written in C++ and compiled with GCC. We select the optimization level of O2. The experiment runs on a server with Intel Xeon 2.00GHz processor and 20GB memory. Considering the influence of warm cache on experimental error, each query is executed five times, and the arithmetic average is taken as the final experimental result.

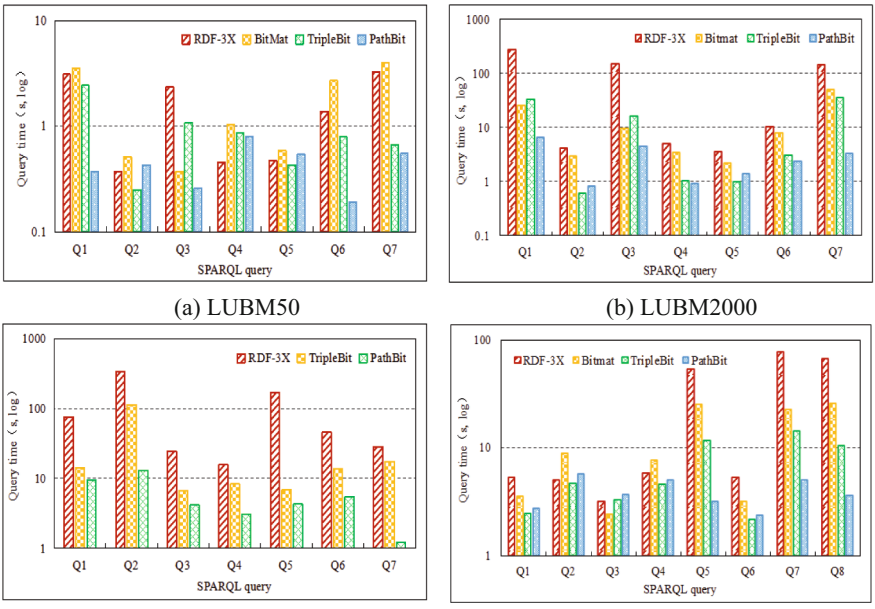
4.2 Comparison of Query Performance

In the experiment, LUBM data set generates 81 predicate paths. If the merging common prefix parameter l is set to 3, we obtain 26 predicate path trees. Figure 2(a) and (b) show the query execution time of SPARQL. The query time of Q1, Q3, Q6 and Q7 are better than the other indexes. These four queries have longer join paths than the star queries Q2, Q4 and Q5. Using the path association information to search can filter a large number of unrelated triples and narrow the retrieval range. Hierarchical path index decomposes the connection between triples into smaller ones, which reduces the connection size of triples and improves the matching efficiency. Intermediate results are also an affecting factor

of query efficiency. The intermediate result in this paper refers to the number of triples matched with the query and the data loaded into memory during the query. Because of the compression method and the direct search in the compressed form, PathBit loads more query data in the same memory and reduce the I/O cost.

It also shows that PathBit is very effective in retrieving large data sets. When the size of LUBM increases from 50 to 2000, the minimum change of query time on RDF-3X is 7.5 times, and the maximum change is 90.83 times, especially for complex queries Q1 and Q3. However, the maximum change of PathBit was only 18.11 times. The reason is that RDF-3X query needs to load more indexes into memory, and at the same time, it also needs to decompress. Therefore, the I/O is larger. Bitmat and Triplebit are both based on triple mode.

In the face of complex queries, they need to join and merge triple more times, so the query performance is lower than that of PathBit. Q2, Q4 and Q5 are star queries. The semantic relevance of predicate path information obtained by star structure is relatively low, but the subject set meeting the conditions is obtained by auxiliary index SP. Combined with subject set and hierarchical edge index, a large number of unrelated triples can be filtered, and the scale of merging results can be reduced. Therefore, the execution efficiency on LUBM 2000 dataset is still better than RDF-3X and Bitmat, which is equivalent to Triplebit. Because the LUBM 50 dataset is small, the index can load memory at once, so RDF-3X retrieval is the highest.



(a) LUBM50 (b) LUBM2000

(c) Uniprot (d) SP²Bench

Fig. 2. Comparision of query performance

The size of Unirpot dataset is 700 million. When the merging common prefix parameter l is set to 3 and the current hardware environment is used to execute the query, Bitmat cannot get the query result. Therefore, Fig. 2(c) only lists the query time comparison of RDF-3X, Triplebit and PathBit. Similarly, after PathBit decomposes the query into query paths, many join operations between triples are decomposed into each path set to complete separately, and the intermediate result set involved in join becomes smaller, especially Q1, Q2, Q5, Q6 and Q7 contain long paths. Combined with the auxiliary indexes, the query performance is superior to Triplebit and RDF-3X. However Q3 and Q4 are star queries, so the overall performance is not as good as long-path retrieval.

The same index is also used to execute queries on SP^2Bench dataset. As with LUBM dataset, the merging common prefix parameter l is also set to 3. Figure 2(d) shows the execution time of each query. Since most of SP^2Bench standard data queries are star structured and query design pays more attention to the use of query operators, the overall query efficiency takes less time, but the filtering and merging operations of the results take a long time. Figure 2(d) depicts the time taken to execute a basic query. Among them, Q1, Q2, Q3 and Q4 are star queries, which are comparable to Triplebit, but better than RDF-3X. However, Q5 and Q7 contain the long path queries, especially Q7, so the query efficiency is significantly improved.

4.3 Comparison of Storage Space

In this part, we compare the storage space of PathBit, RDF-3X and Triplebit. Here, the storage space refers to the space consumed by storing datasets and indexes. Since Bitmat does not contain dictionary tools, the comparison results don't include Bitmat. The merging common prefix parameter l is also set to 3. Table 4 lists the space consumed of different datasets. It can be seen that the storage space of PathBit on all datasets is lower than the other three indexes. As explained earlier, RDF-3X needs to create 6 cluster indexes and 9 clustered indexes. As we all know, the high efficiency of RDF-3X is at the cost of storage space. The dictionary tool used by PathBit is the same as Triplebit. Due to the small number of predicates in LUBM and SP^2Bench and the high merging rate of predicate path, the number of copies is greatly reduced. Therefore, PathBit is better than Triplebit on these two datasets. But the storage space on UniProt dataset is higher than Triplebit.

Table 4. Comparision of storage space (GB)

	RDF-3X	TripleBit	PathBit
LUBM50	0.35	0.28	0.19
LUBM2000	13.95	8.74	7.11
Uniprot	33.89	15.19	17.28
SP^2Bench	7.28	4.17	3.88

4.4 Parameter Analysis

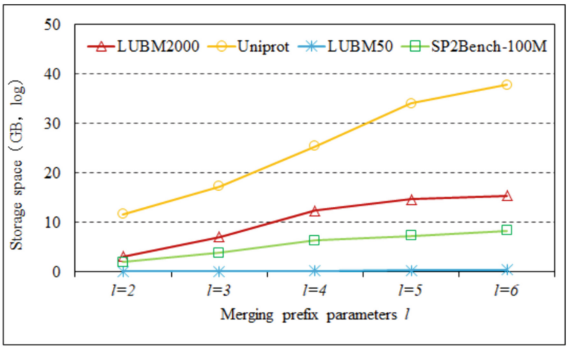
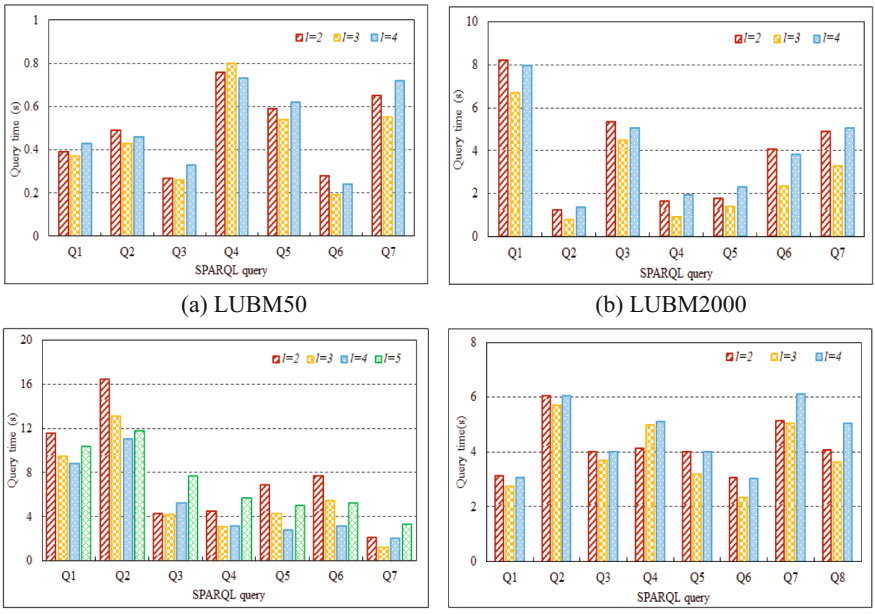


Fig. 3. Comparison of storage space on different values of parameter l

In the process of predicate path merging, the length of common prefix edge is controlled by parameter l. When the length of the common prefix is greater than or equal to l, the edges with the common prefix are merged. This part will test the influence of the parameter l on the storage space.

The range value of l is from 2 to 6. Figure 3 shows that the storage space of all datasets increases with the increase of l. The reason is that the larger the value of l, the smaller the probability of having a common prefix, and the fewer replica nodes that can be merged. In addition, Fig. 3 also shows that with the gradual increase of l value, the change of storage space will be smaller and smaller. When l is set to 4 or 5, the storage space tends to be stable for LUBM and Sp^{2}bench datasets. However, for UniProt, l varies from 5 to 6, because the predicates in UniProt are larger than the other two datasets, which makes the length of common prefix between paths longer.

Figure 4 shows a comparison of query performance. The experimental results show that l has an optimal value, but this value is not directly proportional to the value of l. As shown in Fig. 4, the optimal value of l is 3 for LUBM and Sp^{2}bench datasets, and 4 for UniProt dataset. There are two main reasons. First, when l value is too small, a large number of predicate path are merged, resulting in more candidate paths in the path template matching, which affects the final path matching efficiency. On the contrary, when the value of l is too large, the candidate set becomes smaller and the number of copies increases, which also reduces the query efficiency. Considering the storage space and query performance, the query performance is the best when l takes the storage space to be stable.



(a) LUBM50 (b) LUBM2000

(c) Uniprot (d) SP^{2}Bench

Fig. 4. Comparision of query process time on different values of parameter *l*

5 Conclusions

Aiming at the frequent self joins in triple based retrieval and the semantic association characteristics reflected by chain structure information in SPARQL complex query. This paper proposed a bit index structure based on path (PathBit) for large scale RDF Graph. PathBit created predicate path tree (IPT) to filter complete path sets associated with SPARQL query and designed a k^2-tree index (k^{2}TIP) according to the hierarchy of each predicate path tree. k^{2}TIP realized fast association matching of known predicate path triples. Meanwhile, the compression mechanism is used to implement the compressed storage and retrieval algorithm of triples. In addition, two auxiliary indexes: SP and OP are added to assist predicate path retrieval. In the experiment, we compare PathBit with the three existing index storage schemes. The experimental results show that PathBit is very effective for complex queries, especially for queries with long paths. And with the expansion of data scale, PathBit has higher retrieval advantages. At the same time, the storage space of the compressed storage method used in this paper is 0.96 times less than that of RDF-3X in four datasets under the parameter of combined prefix, and has certain advantages over Triplebit.

In addition, with distributed data processing become mainstream, distributed indexing and querying have become research hotspots. The index structure of PathBit can be divided into several sub trees, and the leaf nodes of each sub tree are related to a complete set of paths. Allocating these sub trees to various computing nodes can achieve

parallel queries. Of course, distributed query systems involve communication and load balancing issues, which will also be our future research direction.

Acknowledgments. This work is partially supported by the Scientific research project of The Educational Department of Liaoning Provincial under Grant LJ2020016 and Research Institute Project of Bohai University under Grant XK202134-3.

References

1. Ning, Z., Huang, J., Wang, X.: Vehicular fog computing: enabling real-time traffic management for smart cities. IEEE Wirel. Commun. **26**(1), 87–93 (2019)
2. Pirrò, G.: Building relatedness explanations from knowledge graphs. Semant. Web **10**(6), 963–990 (2019)
3. Ning, Z., Kwok, R., Zhang, K., et al.: Joint computing and caching in 5G-envisioned Internet of Vehicles: a deep reinforcement learning-based traffic control system. IEEE Trans. Intell. Transp. Syst. **22**(8), 5201–5212 (2020)
4. Feng, J., Meng, C., Song, J., et al.: SPARQL query parallel processing: a survey. In: IEEE International Conference on Big Data, Boston, USA, pp. 444–451 (2017)
5. Wang, X., Ning, Z., et al.: Offloading in Internet of Vehicles: a fog-enabled real-time traffic management system. IEEE Trans. Ind. Inform. **14**(10), 4568–4578 (2018)
6. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010). https://doi.org/10.1007/s00778-009-0165-y
7. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. In: 34th International Conference on VLDB, Auckland, New Zealand, pp. 1008–1019 (2008)
8. Mulay, K., Kumar, P.S.: SPOVC: a scalable RDF store using horizontal partitioning and column oriented DBMS. In: 4th International Workshop on Semantic Web Information Management, Scottsdale, USA, pp. 1–8 (2012)
9. Atre, M., Chaoji, V., Zaki, M.J., et al.: Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In: 19th International Conference on World Wide Web, Raleigh, USA, pp. 41–50 (2010)
10. Matono, A., Pahlevi, S.M., Kojima, I.: RDFCube: a P2P-based three-dimensional index for structural joins on distributed triple stores. In: The International Conference on Databases, Information System, and Peer-to-Peer Computing, Seoul, Korea, pp. 323–330 (2005)
11. Yuan, P., Liu, P., Wu, B., et al.: TripleBit: a fast and compact system for large scale RDF data. In: 39th International Conference on VLDB, Trento, Italy, pp. 517–528 (2013)
12. Harris, S., Gibbins, N.: 3store: efficient bulk RDF storage. In: 1st International Workshop on Practical and Scalable Semantic Systems,Florida, USA, pp. 1–15 (2003)
13. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48005-6_7
14. Carroll, J.J., Dickinson, I., Dollin, C., et al.: Jena: implementing the semantic web recommendations. In: 13th International World Wide Web Conference on Alternate Track Papers & Posters,New York, USA, pp. 74–83 (2004)
15. Abadi, D.J., Marcus, A., Madden, S.R., et al.: SW-store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. **18**(2), 385–406 (2009)
16. Ning, Z., Xia, F., Ullah, N., Kong, X., Xiping, Hu.: Vehicular social networks: enabling smart mobility. IEEE Commun. Mag. **55**(5), 16–55 (2017). https://doi.org/10.1109/MCOM.2017.1600263

17. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: a graph based RDF index. In: 22th AAAI Conference on Artificial Intelligence, British Columbia, Canada, pp. 1465–1470(2007)
18. Zou, L., Özsu, M.T., Chen, L., et al.: GStore: a graph-based SPARQL query engine. VLDB J. **23**(4), 565–590 (2014)
19. Wang, D., Zou, L., Feng, Y., et al.: S-store: an engine for large RDF graph integrating spatial information. In: 18th International Conference on Database Systems for Advanced Applications, Wuhan, China, pp. 31–47 (2013)
20. Tran, T., Ladwig, G.: Structure index for RDF data. In: Proceeding of the Workshop on Semantic Data Management (2010)
21. He, H., Wang, H., Yang, J., et al: BLINKS: ranked keyword searches on graphs. In: The ACM SIGMOD International Proceedings on Management of Data,Beijing, China, pp. 305–316 (2007)
22. Kim, K., Moon, B., Kim, H.J.: R3F: RDF triple filtering method for efficient SPARQL query processing. World Wide Web-Internet Web Inf. Syst. **18**(2), 317–357 (2015)
23. Brisaboa, N.R., Ladra, S., Navarro, G.: Compact representation of Web graphs with extended functionality. Inf. Syst. **39**(1), 152–174 (2014)
24. Schmidt, M., Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. Web Semant. Sci. Serv. Agents World Wide Web **3**(2), 158–182 (2005)
25. Hornung, T., Lausen, G., et al: SP2Bench: a SPARQL performance benchmark.In: 25th International Proceedings on Data Engineering, Shanghai, China, pp. 222–233 (2009)
26. Uniprot RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/