TripleBit: a Fast and Compact System for Large Scale RDF Data

Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang
Services Computing Tech. and System Lab.
School of Computer Science & Technology
Huazhong University of Science and Technology, China
ppyuan, hjin@hust.edu.cn

Ling Liu
Distributed Data Intensive Systems Lab.
School of Computer Science
College of Computing
Georgia Institute of Technology, USA
lingliu@cc.gatech.edu

ABSTRACT

The volume of RDF data continues to grow over the past decade and many known RDF datasets have billions of triples. A grant challenge of managing this huge RDF data is how to access this big RDF data efficiently. A popular approach to addressing the problem is to build a full set of permutations of (S, P, O) indexes. Although this approach has shown to accelerate joins by orders of magnitude, the large space overhead limits the scalability of this approach and makes it heavyweight. In this paper, we present TripleBit, a fast and compact system for storing and accessing RDF data. The design of TripleBit has three salient features. First, the compact design of TripleBit reduces both the size of stored RDF data and the size of its indexes. Second, TripleBit introduces two auxiliary index structures, ID-Chunk bit matrix and ID-Predicate bit matrix, to minimize the cost of index selection during query evaluation. Third, its query processor dynamically generates an optimal execution ordering for join queries, leading to fast query execution and effective reduction on the size of intermediate results. Our experiments show that TripleBit outperforms RDF-3X, MonetDB, BitMat on LUBM, UniProt and BTC 2012 benchmark queries and it offers orders of mangnitude performance improvement for some complex join queries.

1. INTRODUCTION

The Resource Description Framework (RDF) data model and its query language SPARQL are widely adopted today for managing schema-free structured information. Large amount of semantic data are available in the RDF format in many fields of science, engineering, and business, including bioinformatics, life sciences, business intelligence and social networks. A growing number of organizations or Community driven projects, such as White House, New York Times, Wikipedia and Science Commons, have begun exporting RDF data [22]. Linked Open Data Project announced 52 billion triples were published by March 2012 [22].

RDF data are a collection of triples, each with three columns, denoted by Subject (S), Predicate (P) and Object (O). RDF triples tend to have rich relationships, forming a huge and complex RDF graph. Managing large-scale RDF data imposes technical challenges to the conventional storage layout, indexing and query processing [17, 18]. A fair amount of work has been engaged in RDF data management. Triples table [8, 17], column store with vertically partitioning [3] and property tables [25] are the three most popular alternative storage layouts for storing and accessing RDF data. The storage layouts may not favor all queries. However, queries constrained on S, P or O values are equally important for real-world applications. A popular approach to achieving this goal is to maintain all six permutation indexes on the RDF data in order to provide efficient query processing for all possible access patterns [5, 9, 17, 24]. Although the permutation indexing techniques can speed up joins by orders of magnitude, they may result in significant demand for both main memory and disk storage. First, RDF stores need load those indexes into limited memory of computer in order to generate query plans when processing complex join queries. Consequently, frequent memory swap in/out, and out of memory problems are common when querying RDF data with over a billion of triples [14]. Furthermore, the large space overhead also places a heavy burden on both memory and disk I/O. One way to address the space cost is to use compression techniques, such as D-Gap [5], delta compression [17], in storing and accessing RDF data. Multiple permutation indexes also complicate the decision on the choices of the indexes for a given query.

In this paper, we present TripleBit, a fast and compact system for large scale RDF data. TripeBit is designed based on two important observations. First, it is important to design an RDF data storage structure that can directly and efficiently query the compressed data. This motivates us to design a compact storage and index structure in TripleBit. Second, in order to truly scale the RDF query processor, we need efficient index structures and query evaluation algorithms to minimize the size of intermediate results generated when evaluating queries, especially complex join queries. This leads us to the design decision that we should not only reduce the size of indexes (e.g., through compression techniques) but also minimize the number of indexes used in query evaluation.

The main contributions of the paper are three folds: *First*, we present a compact RDF store - TripleBit, including the design of a bit matrix storage structure and the encoding-based compression method for storing huge RDF graphs more efficiently. The storage structure enables TripleBit to use merge joins extensively for join processing. *Second*, we develop two auxiliary indexing structures, ID-Chunk bit matrix and ID-Predicate bit matrix, to reduce the number and the size of indexes to the minimum while providing orders of magnitude speedup for scan and merge-join perfor-

Proceedings of the VLDB Endowment, Vol. 6, No. 7

mance. The ID-Chunk bit matrix provides a fast search of the relevant chunks matching to a given subject (S) or object (O). The ID-Predicate bit matrix provides a mapping of a subject (S) or an object (O) to the list of predicates to which it relates. *Third*, we employ the dynamic query plan generation algorithm to generate an optimal execution plan for a join query, aiming at reducing the size of intermediate results as early as possible. We evaluate TripleBit through extensive experiments against RDF graphs of up to 2.9 billion triples. Our experimental comparison with the state of art RDF stores, such as RDF-3X, MonetDB, shows that the TripleBit consistently outperforms them and delivers up to 2-4 orders of magnitude better performance for complex long join queries over large scale RDF data.

2. OVERVIEW & RELATED WORK

A fair number of RDF storage systems has been developed in the past decade, such as Sesame [8], Jena [25], RDF-3X [17, 18], Hexastore [24], BitMat [5], gStore [28] etc. These systems can be broadly classified into four categories: triples table [17], property table [25], column store with vertical partitioning [3] and RDF graph based store. We will illustrate and analyze these four categories of RDF stores using the following example RDF dataset.

T1: person1 isNamed "Tom".
T2: publication1 hasAuthor person1.
T3: publication1 isTitled "Pub1".
T4: person2 isNamed "James".
T5: publication2 hasAuthor person2.
T6: publication2 isTitled "Pub2".
T7: publication1 hasCitation publication2.

Triple table. A natural approach to storing RDF data is to store (S, P, O) statements in a 3-column table with each row representing a RDF statement. The 3-column table is called the triple table [17]. For the above example, the seven statements will correspond to seven rows of a triple table. There are several variants of the triple table, e.g., storing literals and URIs in a separate table and using pointers to refer to literals and URIs in the triple table. However, querying over an RDF table of billions of rows can be challenging. First, most of queries involve self-joins over this long table. Second, larger table size leads to larger table scan and larger index look up time, which complicates both selectivity estimation and query optimization [3]. A popular approach to improving performance of queries over a triple table is to use an exhaustive indexing method that creates a full set of (S, P, O) permutations of indexes [17, 27]. For example, RDF-3X, one of the best RDF stores today, built clustered B+-trees on all six (S, P, O) permutations − (SPO, SOP, PSO, POS, OSP, OPS), thus each RDF dataset is stored in six duplicates, one per index. In order to choose the fastest index among the six indexes for a given query, another set of 9 aggregate indexes, including all six binary projections − (SP, SO, PO, PS, OS, OP), and three unary projections − (S, P, O) [17, 18], are created and maintained, each providing some selectivity statistics. By maintaining such aggregate indexes, RDF-3X eliminates the problem of expensive self-joins and provides significant performance improvement. However, storing all permutation indexes may be expensive and the performance penalty can be high as the volume of dataset increases due to the cost of storing and accessing these indexes and the cost of deciding which of these indexes to use at the query evaluation.

Property table. Instead of using a "long and slim" triple table, the property table typically stores RDF data in a "fat" property table with subject as the first column and the list of distinct predicates as the remaining columns [25]. Each row of the property table corresponds to a distinct S-value. Each of the remaining columns corresponds to a predicate (P-value). Each table cell represents an O-value of a triple with the given S-value and P-value. Consider our example RDF dataset of 7 triples with 4 distinct properties (predicates), and thus we will have a 5-column property table. *publication1* has three properties: *hasAuthor*, *isTitled*, *hasCitation*. Thus, three statements are mapped into one row corresponding to *publication1* in the table. Clearly, for a big RDF dataset, a single property table can be extremely sparse and contains many *NULL* values. Thus multiple-property tables with different clusters of properties are proposed in Jena [25] as an optimization technique. BitMat [5] represents an alternative design of the property table approach, in which RDF triples are represented as a 3D bit-cube, representing subjects, predicates and objects respectively and slicing along a dimension to get 2D matrices: SO, PO and PS.

An advantage of the property tables is that the subject-subject self-joins on the subject column can be eliminated. However, the property table approach suffers from several problems [3]: First, the space overhead of the wide property table(s) with sparse attributes is high. Second, processing of RDF queries that have no restriction on property values may involve scanning all property tables. Furthermore, experimental results in [12] have shown that the performance of the property table approach degrades dramatically when dealing with large scale RDF data.

Column store with vertical partitioning. This approach stores RDF data [3] using multiple two-column tables, one for each unique predicate. The first column is for subject whereas the other column is for object. Consider our running example with four properties, this approach will map 7 statements to four 2-column tables. Although those tables can be stored using either row-oriented or column-oriented DBMS, the column store is a more popular storage solution for vertically partitioned schema [3]. This approach is easy to implement and can provide superior performance for queries with value-based restrictions on properties. However, this approach may suffer from scalability problems when the size of tables varied significantly [19]. Furthermore, processing join queries with multiple join conditions and unrestricted properties can be extremely expensive due to the need of accessing all of the 2-column tables and the possibility of generating large intermediate results.

Graph-based store. Graph-based approaches represent an orthogonal dimension of RDF store research [7, 14], aiming at improving the performance of graph-based manipulations on RDF datasets beyond RDF SPARQL queries. However, most of these graph based approaches focus more on improving the performance of specialized graph operations rather than the scalability and efficiency of RDF query processing [24]. Large scale RDF data is a very big sparse graph, which poses a significant challenge to store and query such an RDF graph efficiently. [4] developed a compressed RDF engine k^2-triples. gStore [28] proposes VS-tree and VS*-tree index to process both exact and wildcard SPARQL queries by efficient subgraph matching.

In comparison, TripleBit advocates two important design principles: First, in order to truly scale the RDF query processor, we should design compact storage structure and minimize the number of indexes used in query evaluation. Second, we need compact index structure as well as efficient index utilization techniques to minimize the size of intermediate results generated during query processing and to process complex joins efficiently.

3. TRIPLE MATRIX AND ITS STORAGE STRUCTURE

We design the TripleBit storage structure with three objectives in mind: improving storage compactness, improving encoding or compression efficiency and improving query processing efficiency.

Table 1: The Triple Matrix of the example

	isNamed		hasAuthor		isTitled		hasCitation
	T1	T4	T2	T5	T3	T6	T7
person1	1	0	1	0	0	0	0
person2	0	1	0	1	0	0	0
publication1	0	0	1	0	1	0	1
publication2	0	0	0	1	0	1	1
"Tom"	1	0	0	0	0	0	0
"James"	0	1	0	0	0	0	0
"Pub1"	0	0	0	0	1	0	0
"Pub2"	0	0	0	0	0	1	0

We first present the Triple Matrix model and then describe how the storage layout design of the Triple Matrix model offers more compact storage, higher encoding efficiency and faster query execution.

In Triple Matrix model, RDF triples are represented as a two dimensional bit matrix. We call it the triple matrix. Concretely, given an RDF dataset, let V_S, V_P, V_O and V_T denote the set of distinct subjects, predicates, objects and triples respectively, The triple matrix is created with entity $e \in V_E = V_S \cup V_O$ as one dimension (row) and triples $t \in V_T$ as the other dimension (column). Thus, we can view the corresponding triple matrix as a two-dimensional table with $|V_T|$ columns and $|V_E|$ rows. Each column of the matrix corresponds to an RDF triple, with only two entries of bit value ('1'), corresponding to the subject entity and object entity of the triple and all the rest of ($|V_E| - 2$) entries of bit ('0'). Each row is defined by a distinct entity value, with the presence ('1') in a subset of entries, representing a collection of the triples having the same entity. We sort the columns by predicates in lexicographic order and vertically partition the matrix into multiple disjoint buckets, one per predicate (property). For a new triple with predicate p, it will be inserted into a column in the predicate bucket corresponding to p. Assume that the triple to be inserted has subject i, object j and predicate p, and it corresponds to the column k, then the entries that lie in the i-th row, the j-th row and the k-th column of the matrix are set to '1'. The other entries in the k-th column are set to '0'. Table 1 shows the triple matrix for the running example of 7 triples. It has 7 columns, one per triple, and eight entities, representing four subject values and four object values. We use strings instead of row *ids* in this example solely for the readability.

In addition, during the construction of the triple matrix from RDF data, each $e \in V_E$ is assigned a unique *ID* using the row number in the matrix. We assign IDs to subjects and objects using the same ID space such that subjects and objects having identical values will be treated as the same entity. We observe that a fair amount of entities in many real RDF datasets are used as subject of a triple and object of another triple. For example, Table 5 in Section 6 showed that more than 57% subjects of UniProt are also objects. TripleBit utilizes unique IDs for the same entities to achieve a more compact storage and to further improve the query processing efficiency. This is because query processor does not need to distinguish whether IDs represent subject or object entities when processing joins. Furthermore, our approach facilitates index construction and makes the join on subject and object more efficient than the approaches where subjects and objects have independent ID space [5].

3.1 Dictionary

In RDF specification, Universal Resource Identifiers (URIs) are used to identify resources, such as subjects and objects. A resource may have many properties and corresponding values. It is not economical to store URIs in each appearance of resources. To reduce the redundancy, like many RDF stores, such as RDF-3X, in TripleBit we replace all strings (URI, literals and blank node) by

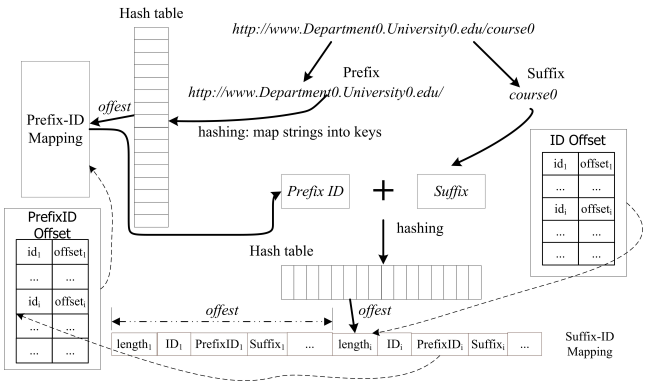

Figure 1: String-ID Mapping and ID-String Mapping

*ID*s using mapping dictionaries. Considering the existence of long common prefixes in URIs, we adopt a prefix compression method, which is similar to Front Coding to obtain compressed dictionaries. The prefix compression method splits each URI into a prefix slice and a suffix slice at the last occurrence of separator '/'. The strings which do not contain '/' are considered as suffixes. We assign each prefix a *PrefixID* and construct Prefix-ID mapping table. We concatenate prefix ID and suffix to get a new string which is also assigned an *ID* in an independent ID space. Similarly, we build a Suffix-ID mapping table (Fig. 1).

During query translation, hashing function maps prefix of a string *str* to its index and then the *offset* of prefix stored in corresponding slot of the hash table is accessed. Using the *offset*, we can get its *PrefixID* in Prefix-ID mapping table. Using the above process, the *ID* assigned to *str* is returned. The process to translate a string to its *ID* is illustrated using the solid lines in Fig. 1.

Before query results are returned to users or applications, *ID*s in the results must be translated back into strings. In order to speed up the reverse process, we build two inverted tables (PrefixID Offset and ID Offset in Fig. 1) to translate IDs back into strings. Both tables store *(id, offset)* pairs where *id* corresponds to the *PrefixID* or IDs, and *offset* represents the position where the prefix or suffix and the *ID* are stored. Inverted table structures make the mapping of *id*s to literals or URIs more efficient. The process to transform *id*s back to strings is shown by dashed line in Fig. 1.

In summary, searching *id*s in the dictionary using strings does not require much time thanks to the hashing index and the fact that the number of strings occurring in a query is very small. However, the reverse mapping process can be costly when the query result size is big [16].

3.2 ID Encoding in the Triple Matrix

In a triple matrix, *ID* is an integer. The size for storing an integer is typically a word. Modern computers usually have a word size of 32 bits or 64 bits. Not all integers require the whole word to store them. For example, it is enough to store the ID of value "100" using 1 byte. It is wasteful to store values with a large number of bytes when a small number of bytes are sufficient. Also, since TripleBit is designed for large scale RDF data, it is difficult to know the maximal number of entities in future data sets. For example, 32-bits ID might be a good choice for current RDF data, but insufficient in the near future. Thus, we encode the entity *ID* in triple matrix using variable-size integer such that the minimum number of bytes are used to encode this integer. Furthermore, the most significant bit of each compressed byte is used to indicate whether an *ID* is the subject(0) or object (1) of a statement and the remaining 7 bits are used to store the value. Consider the example in Table 1. The entity per-

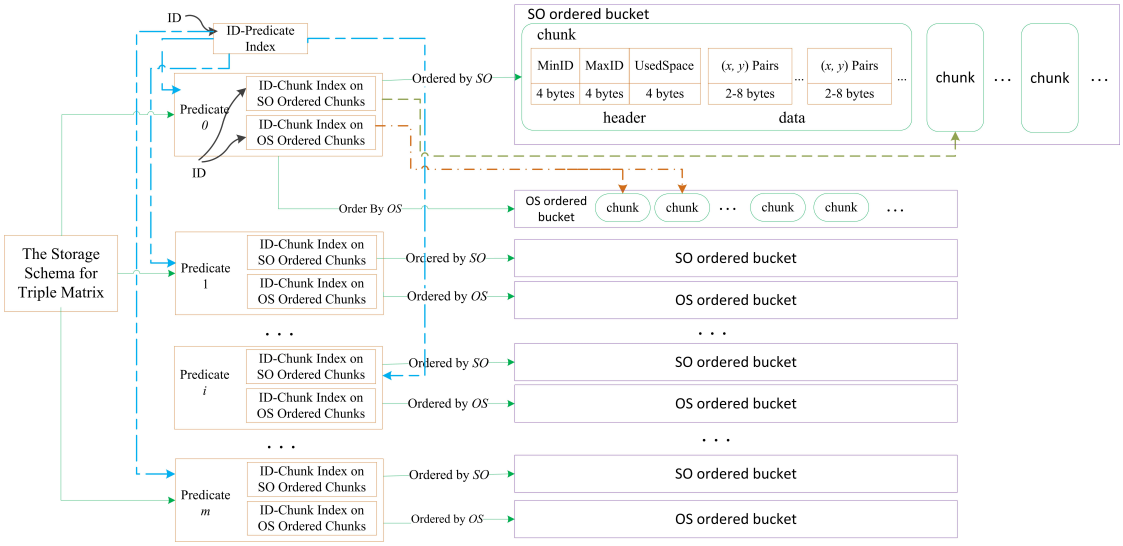

Figure 2: The Storage scheme of TripleBit

Table 2: Storage space of Triple Matrix for six datasets

	LUBM 10M	LUBM 50M	LUBM 100M	LUBM 500M	LUBM 1B	UniProt
Two Copies of Triple Matrix (MB)	133	722	1,461	7,363	15,207	24,371
Per Triple (bytes)	5.02	5.48	5.54	5.59	5.97	4.33

son1 denotes a subject in $T1$ and an object in $T2$. Thus, the row of this entity has its 1st and 3rd columns set to '1'. We use 00000001 (subject) in the column encoding of $T1$ and use 10000001 (object) in the column encoding of $T2$ respectively. By utilizing the significant bit of each compressed byte, we can easily find the triple of interest without scanning the entire chunk.

Many RDF stores use fixed-sized integer (e.g., an integer of 4 bytes in 32-bit computer) to encode *IDs*. Comparing with fixed-sized integer, the overhead of the variable-size integer encoding approach is 1 bit per byte but this approach saves more with higher flexibility. Our experiments on the 6 datasets in Table 2 show that each ID needs about 2.1-3 bytes on average. Furthermore, our approach is highly extensible comparing with fixed-sized *id*s since the former can encode any number of subjects or objects. For example, we can encode *ID*s larger than 2^{32} using 5 or more bytes while fixed-sized integer does not have such flexibility.

3.3 Triple Matrix Column Compression

The triple matrix is inherently sparse. To achieve the internal compact representation of the triple matrix, we store the bit matrix in a compressed format using column compression. Given that each column of the matrix corresponds to a triple and thus has only two entries with '1', we show that a column-level compression scheme for storing the triple matrix is more effective than the row-level, byte-level or bit-level compression scheme [5]. Concretely, for each column of the triple matrix, instead of storing the entire column of size $|V_E|$, we use only the two row numbers (i.e., *IDs*) that correspond to the two '1' entries. Consider Table 1, the first column ($T1$) and the third column ($T2$) are represented as 00000001 10000101 and 10000001 00000011 respectively. By combining with the variable-size integer encoding approach for the two IDs, each column requires only 2-8 bytes for storing one triple in TripleBit if the number of entities (or rows) is less than 2^{28}. Furthermore, instead of storing two full ids of a column we can store the first id and the changes between two ids. If two ids are similar, it will further save storage. Our experiments on the six data sets in Table 2 show that the storage per triple is about 4.3-6 bytes on average without other storage optimization. This saving at per-triple level is significant compared to 12 bytes per triple in the row stores and 8 bytes per triple in the column stores, leading to higher efficiency in storing and scanning data on storage media as well as high reduction in both the size of intermediate results and the time complexity of query processing.

3.4 Triple Matrix Chunk Storage

As described earlier, we partition a triple matrix vertically into predicate-based buckets, each containing triples with the same predicate. Triples of each bucket are stored in fixed-size chunks. Chunks are physically clustered by predicates such that chunk clusters having the same predicates are placed adjacently on storage media. We assign chunks of each bucket with chunk IDs consecutively. The size of a chunk can be set according to a number of parameters, such as the size of dataset, the memory capacity, the IO page size. Although search in small size chunk is faster, larger chunk simplifies the construction of ID-Chunk index and reduces I/O. Larger chunk also has less storage overhead. In the first implementation of TripleBit, we set the chunk size to be 64KB.

Since the triple matrix does not indicate which entity is subject or object in a column, we choose to store each column of a bucket in a consistent order, either SO or OS. Compared to some existing RDF systems [17, 24], which store all six permutations of RDF data, TripleBit stores the triple matrix for each RDF dataset physically in two duplicates, one in S-O order and another in O-S order. The triples in a SO-bucket or an OS-bucket are sorted by S-O order or O-S order respectively and SO pairs or OS pairs are stored consecutively in the chunks of each bucket (see Fig. 2). Thus, each chunk is either SO-ordered chunk or OS-ordered chunk. Considering the example in Table 1, the triples corresponding to '*isNamed*' are stored in the SO chunk as follows: 00000001 10000101 00000010 10000110.

Fig. 2 gives a sketch of TripleBit storage layout. In the head of each chunk, we store the minimum and maximum subject IDs in each SO chunk and the minimum and maximum object IDs in each OS chunk, as well as the amount of used space.

Consider a query with a given predicate and a given subject having ID of value "*id*". We process this query in three steps: (1) By

using the given predicate, we locate the corresponding SO bucket containing triples with the given predicate. (2) We need to find the relevant SO chunks that contain triples with the given subject value "*id*" by checking whether target *id* falls inside the *MinID*s and *MaxID*s of chunks. (3) Now we examine each relevant SO chunk to find the SO pairs matching the given subject value "*id*" using binary search instead of full scan. Recall Section 3.2, our ID encoding in the triple matrix utilizes the most significant bit of each byte of an entity *ID* to indicate whether the *ID* refers to a subject or an object of a triple. This feature allows TripleBit to get an SO pair (or OS pair) more efficiently in an SO-chunk (or OS chunk). Concretely, we start at the middle byte of an SO ordered chunk, say "00001001". TripleBit finds the matching SO pair, namely the *ID* of the subject and the *ID* of the object of the matching triple, in two steps. First, TripleBit reads the previous bytes and next bytes until the most significant bit of the bytes are not '0'. Then TripleBit reads next bytes till the most significant bit of the bytes are not '1' and get the SO pair. Now TripleBit compares the query input *id* to the subject *ID* of the SO pair. If it is a match, it returns the subject and object of the matching triple. Otherwise, it continues to compare and determine whether the input *id* is less or greater than the stored subject *ID* and starts the next round of binary search with the search scope reduced by a half. TripleBit repeats this process. The search space is reduced by a half at each iteration and thus it can quickly locate the range of matching SO pairs. Similar process applies to the OS-chunks if the object of the query is given.

In summary, the Triple Matrix model is conceptually attractive as it can facilitate the design of compact RDF storage layout, compact RDF indexes and ease of query processing. The Triple Matrix model prevails over other existing RDF stores, such as triple row store, column store with vertically partitioning, and property table based model, for a number of reasons. First, the Triple Matrix model allows efficient encoding of each entity using its row ID through variable-sized integer encoding. Second, the Triple Matrix model enables effective column level encoding. By combining both entity ID level and column level compression, Triple Matrix model enables TripleBit to generate a very compact storage model for RDF triples. Third, the Triple Matrix model and its predicate-based triple buckets provide a natural and intuitive way to organize TripleBit storage layout by placing the triples with the same predicate adjacently in contiguous chunks of the corresponding bucket. Thus, TripleBit significantly speeds up queries with restricted predicate P-value (see Section 5 for more detail). Furthermore, the triple matrix is by design suitable for parallel processing of queries. For example, Triple Matrix can be partitioned into several sub-matrixes, each of which corresponds to a subgraph of the whole RDF graph. Those sub-matrixes can be placed onto different server nodes of a compute cluster. Thus, queries can be executed in parallel across multiple nodes using a distributed framework, such as MapReduce [11].

In addition, TripleBit can provide optimized storage for reified statements and can store and process reified statements naturally using its triple matrix by establishing a mapping of a reified statement identifier to a row id, avoiding the use of four separate triples for each reified statement. Due to the page limit, we refer readers to our technical report[1] for further detail.

4. INDEXING

With the triple matrix and the predicate-based triple buckets of SO chunks and OS chunks, TripleBit only needs two of the six permutations of (S, P, O) in its physical storage, namely PSO and POS. In order to speed up the processing of RDF queries, we design two auxiliary index structures: ID-Chunk matrix and ID-Predicate bit-Matrix.

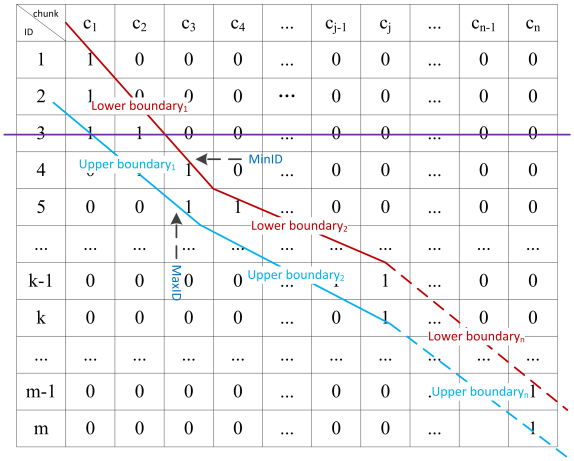

Figure 3: ID-Chunk Bit Matrix

4.1 ID-Chunk Index

ID-Chunk index is created as an ID-Chunk matrix for each distinct predicate and it captures the storage relationship between IDs (rows) and Chunks (columns) having the same predicate. An entry in the ID-Chunk matrix is a bit to denote the presence ('1') or absence ('0') of an ID in the corresponding chunk. Since the triples having the same predicate is stored physically in two buckets, we maintain two ID-Chunk index for each predicate: one for SO ordering chunks and the other for OS ordering chunks (Fig. 2).

In each ID-chunk matrix, rows and columns are sorted in an ascending order of IDs and sorted chunks respectively. Given an entity *id*, the set of chunks that store the triples with this *id* are adjacent physically in the storage media and thus appeared in the consecutive columns in the ID-chunk matrix with non-zero entries around the main diagonal. The degree of shift of the non-zero diagonal from the main diagonal of the matrix depends on the total number of triples containing this *id* as subject or object. We can draw two finite sequences of line segments, which bound the non-zero entries of the matrix (as shown in Fig. 3). Considering the *MinID*s and *MaxID*s of chunks as two set of data points, we can fit the upper boundary lines and lower boundary lines using curve fitting. There are multiple curve fitting methods, such as lines, polynomial curves or Bspline, etc. Complicated fitting methods involve large overhead when computing index. Thus, currently we divide the rows into several parts (e.g., 4 parts). The upper bound and the lower bound of each part are fitted using two lines whose parameters are determined by least square method. Since non-zero entries of the ID-Chunk Matrix are expressed using two set of lines, we only need to keep the parameters of two set of lines.

The ID-Chunk index gives the lower bound Chunk ID and upper bound of Chunk ID for each chunk in the given predicate bucket (shown by the boundary lines). Thus, a query with a given predicate and a given subject *id* (or object *id*) can be processed by first hashing the given predicate to get the corresponding bucket. Then, instead of a full scan over all chunks in the predicate bucket, TripleBit only scan the range of contiguous chunks in the bucket where the given subject or object *id* appears, namely finding the lower bound Chunk ID and upper bound of Chunk ID corresponding to the given *id* using the ID-Chunk index. For those chunks identified by the

[1]http://www.cc.gatech.edu/~lingliu/TechReport/TripleBit-report-v2.Dec.2012.pdf

Table 3: Index lookup under varying chunk sizes (time in μs)

	Cold cache				Warm cache			
Chunk size	4KB	16KB	32KB	64KB	4KB	16KB	32KB	64KB
ID-Chunk	43	56	59	144	3.8	23.3	23.8	104
B+-Tree	16373	15393	15554	292378	16072	13797	13742	13647

Table 4: Query time of LUBM-Q2 under varying chunk sizes

Chunk size	1KB	2KB	4KB	16KB	32KB	64KB
cold caches	0.0497	0.0489	0.0466	0.0440	0.0336	0.0295
warm caches	0.00017	0.00018	0.00019	0.00025	0.00026	0.00021

ID-Chunk index, we can further examine the query relevance of the triples stored in the chunks by utilizing the *MinID* and *MaxID* stored at the head of each chunk and a binary search, instead of a full scan of all triples in each chunk (recall Section 3.4).

To better understand the effectiveness of ID-Chunk index for TripleBit, we compare ID-Chunk index with B+-Tree index under different chunk sizes by constructing a B+-Tree index on chunks. We choose triples sharing the same predicate *rdf:type* of LUBM-1B as the test dataset. Generally, each subject declares its type. Thus the dataset is big. The time to construct the ID-Chunk index is slightly smaller (about 140*s*) than the construction time of B+-Tree (about 146*s*). Table 3 shows lookup in B+-Tree is significantly slower than lookup using ID-Chunk index in all chunk sizes. When the chunk size is 64KB, the average times required to lookup in B+-Tree and find the target pairs are 292.378*ms* (cold cache) and 13.647*ms* (warm cache) while the times for lookup in ID-Chunk are 0.144*ms* (cold cache) and 0.104*ms* (warm cache) respectively. A primary reason is that B+-Tree require large storage space (8.6MB-60MB) while ID-Chunk stores only parameters of functions (128 bytes). We also provide an experimental study of the performance impact of chunk size on TripleBit. Table 4 shows the query time (in seconds) of LUBM Q2 running on LUBM-500M under varying chunk sizes. LUBM Q2 is chosen because triples matching Q2 are in a single chunk and the intermediate result size of Q2 is also not big. Thus, the factors impact on the performance of index lookup is more obvious than other complex queries with larger intermediate results. For both indexes, larger chunk has better performance in cold cache because of less I/O, and smaller chunk has better performance in warm cache cases. For more complex queries which need access more inconsecutive chunks, ID-Chunk index outperforms B+-tree by higher orders of magnitude.

4.2 ID-Predicate Index

The second index structure is the **ID-Predicate bit matrix**. We use this index to speed up the queries with un-restricted predicates. Given a query with no restriction on any predicate, instead of a sequential scan of all predicate buckets, we introduce the ID-Predicate bit matrix index structure. An entry with a bit of '1' in the ID-Predicate matrix indicates the occurrence relationship between the ID row and the predicate column. With the ID-Predicate index, if a subject or an object is known in a query, TripleBit can determine the set of relevant predicates. For each relevant predicate, TripleBit can use ID-Chunk matrix to locate the relevant chunks and return the matching triples by binary search within each relevant chunk.

The ID-Predicate matrix is huge and sparse for large scale RDF data. In TripleBit we use semantic preserving compression techniques that can make the matrix compact in storage and memory but remain searchable by IDs. We decompose ID-Predicate matrix into a set of block matrixes and treat each block of the ID-Predicate matrix as a bit vector. We devise the following byte encoding technique by adapting the Word Aligned Hybrid (WAH) compression scheme [26]. Instead of imposing the word alignment requirement as is done by WAH, we choose to impose the byte alignment requirement on the blocks of the matrix. For example, we first divide the bit vector into 7-bit segments and then merge the 7-bit segments into groups such that consecutive identical bit segments are grouped together. A *fill* is a consecutive group of 7-bits where the bits are either all 0 or all 1, and a *literal* is a consecutive group of 7-bits with a mixture of 0 and 1. Then we encode each fill as follows: The most significant bit of each byte is used to differentiate between literal (0) and fill (1) bytes. The second most significant bit of a fill word indicates the fill bit (0 or 1), and the remaining bits store the fill length. Compressed blocks are stored into fixed-size storage structure adjacently. Thus, given an *id*, it is easy to locate the blocks where the *id*th row is.

	uncompressed (in 7-bit groups)
A	31 00 03 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E
B	56 7F 7F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	compressed
A	31 81 03 50 8F 0E
B	56 C2 93

Figure 4: Two compressed bit vectors

Fig. 4 shows the compressed representation of two examples. The second and third line in Fig. 4 show the hexadecimal representation of the bit vector as 7-bit groups. The last two lines show the compressed bytes also as hexadecimal numbers. For example, the first byte of the last line ("56") is a literal byte, and the second and third are fill bytes. The fill byte "C2" indicates a 1-fill of 14 bits long, and the fill byte "93" denotes a 0-fill, containing 133 consecutive 0 bits.

4.3 Aggregate Indexes

The execution time of a query is heavily influenced by the number and execution order of joins to be performed and the means to find the results of the query. Therefore, the query processor needs to utilize the selectivity estimation of query patterns to select the most effective indexes, minimize the number of indexes needed and determine the query plan. In SPARQL queries, there are eight triple query patterns: one full scan and 7 triple selection patterns. All triples in the store match $(?s\ ?p\ ?o)$ and thus a full scan is required. In the other end of the spectrum, the number of triples matching $(s\ p\ o)$ is 0 or 1. The selectivity of these two patterns is known intuitively without aggregate indexes. The statistics of triple pattern $(?s\ p\ ?o)$ can be obtained directly in the storage structure corresponding to the bucket of predicate p. Hence, we need to estimate the selectivity of five triple query patterns: $(s\ p\ ?o)$; $(?s\ p\ o)$; $(s\ ?p\ o)$; $(s\ ?p\ ?o)$; $(?s\ ?p\ o)$.

In TripleBit, we additionally build two binary aggregate indexes: SP and OP (instead of 9 aggregate indexes [16–18]). The SP aggregate index stores the count of the triples with the same subject and the same predicate. With SP aggregate index, we can compute statistics about $(s\ p\ ?o)$ and $(s\ ?p\ ?o)$. For example, to get the number of triples matching $(s\ ?p\ ?o)$, TripleBit searches SP aggregate index and locates the first tuples containing s. Since SP pairs are stored lexicographically, TripleBit can count all the tuples having the same s for each predicate and return the count of triples. Similarly, the OP aggregate index gives the count of the triples having the same object and the same predicate for fast computation of the statistics about $(?s\ ?p\ o)$ and $(?s\ p\ o)$. Finally, statistics about $(s\ ?p\ o)$ can be computed efficiently using ID-Predicate index with SP and OP indexes. Both aggregated indexes (SP and OP) are compressed using delta compression [17] and stored in chunks.

In summary, the indexing structure in TripleBit is also compact. We minimize the size of the indexes through storing ID-Chunk index as a list of functions which requires tiny storage. We re-

duce the number of indexes by utilizing a novel triple matrix based storage layout and two auxiliary bit matrix indexes. More importantly, the compactness of TripleBit storage and index structures makes it highly effective for complex long join queries, compared to exhaustive-indexing used in some triples table [17, 24]. For example, using SO and OS chunk pairs in the storage and ID-Chunk index, we can replace PSO, POS indexes. By adding the ID-Predicate matrix, TripleBit can cover all the other four permutations of (S, P, O). To estimate selectivity, we only use two aggregate indexes instead of all permutations of 9 aggregate indexes [16, 18].

5. QUERY PROCESSING

There are two types SPARQL queries: queries with single selection triple pattern and queries with join triple patterns. Processing queries with single selection triple pattern is straightforward. When a query consists of multiple triple patterns that share at least one variable, we call the query the join triple pattern query. For this type of queries, TripleBit generates the query plan dynamically, aiming at reducing the size of intermediate results and then executes the final full joins accordingly. In this section we describe how the triple matrix, the ID-Chunk and ID-Predicate indexes are utilized to process these two types of queries efficiently.

5.1 Queries with Selection Triple Pattern

We have briefly discussed in Section 4.3 about the eight selection triple patterns: $(?s\ p\ ?o)$; $(s\ p\ ?o)$; $(?s\ p\ o)$; $(s\ p\ o)$; $(s\ ?p\ o)$; $(s\ ?p\ ?o)$; $(?s\ ?p\ o)$; $(?s\ ?p\ ?o)$. Processing queries with these simple selection triple patterns is straightforward. Due to page length limit, we below describe the steps for evaluating two representative triple patterns: $(s\ p\ ?o)$ and $(s\ ?p\ o)$.

For $(s\ p\ ?o)$, we first hash by p to obtain the predicate bucket of p and use the ID-Chunk index of the bucket p to locate the range of chunks relevant to the given s. Next, the query processor examines each of the candidate SO chunks to see if s falls inside the range of *MinID* and *MaxID* of this chunk to prune out irrelevant chunks. For the relevant chunks, a binary search is performed over each of such chunks to find the matching SO pairs. There are three special cases: (i) If the *MinID* and *MaxID* of a chunk equal to s, then all the SO pairs in the chunk are matching the queries and the query processor just return all the triples in the chunk. (ii) If the *MinID* of a chunk equals to s, the query processor just locates the first pair which does not match the query pattern and returns all the SO pairs before that pair. (iii) If the *MaxID* of a chunk equals to s, similarly, the query processor just locates the last pair whose subject is not s and returns all the SO pairs after that pair.

To execute the selection triple query pattern $(s\ ?p\ o)$, the query processor first needs to determine which predicates are relevant using s and o. It searches the ID-Predicate index using both s and o, and get two sets of candidate predicates, one based on s and the other based on o. Then the query processor computes the intersection of the two sets of predicates, which gives the set of relevant predicates connecting s to o. For each matching predicate p, the query processor first determines whether to use SO chunks or OS chunks by comparing the selectivity of sp and the selectivity of op using the aggregate indexes: SP and OP, denoted by $\sigma_f(sp)$ and $\sigma_f(op)$ respectively. If $\sigma_f(sp) \geq \sigma_f(op)$, then the chunks ordered by s is more selective and we search the S-O chunk(s) corresponding to p. Otherwise, we search the chunks ordered by o. Finally, we output those triples matching $(s\ p\ o)$.

5.2 Queries with Join Triple Patterns

A query with join triple patterns typically forms a query graph [5, 10, 21], with selection triple patterns or variables as nodes and the types of joins as edges. We classify multi-triple pattern queries into three categories: (i) *star join* where many triple patterns are joined on one common variable, (ii) *cyclic join* where join variables are connected as a cycle, and (iii) *bridge join* where several stars are connected as a chain. In LUBM, Q1, Q2 are star joins; Q3, Q5, Q6 are cyclic joins; and Q4 is bridge join. SPARQL queries tend to contain multiple star-shaped sub-queries [17]. For example, cyclic joins and bridge joins are star joins connected as a cycle or a chain.

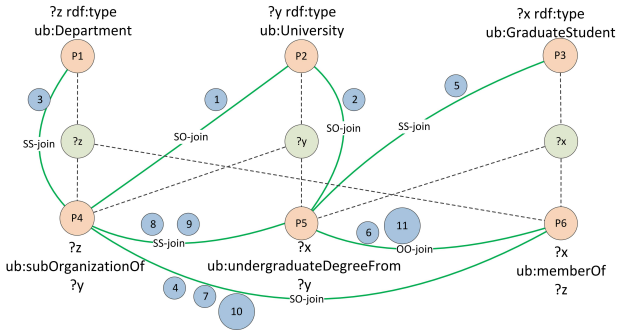

Figure 5: Query graph and its query plan of LUBM Q5

Here, we use the query graph model of [5] to represent a query. In this query graph model, nodes are triple patterns and join variables. There are two kinds of edges connecting the nodes: One type of edges between pattern nodes and variable nodes, indicating the variables appearing in the corresponding triple patterns. The other type of edges between two triple patterns, denoting one of the three join types: *SS-join*, the subject-subject join, *SO-join*, the subject-object join and *OO-join*, the object-object join. Fig. 5 is the query graph of LUBM-Q5, with some edges (e.g., the join edge between $P1$ and $P6$) omitted for presentation clarity.

When a query involves multiple join patterns as shown in Fig. 5, the most important task is to produce an optimal execution ordering of the join nodes (join triple patterns) of the query graph. We can compute the optimal ordering of join patterns in terms of three factors: (i) the triple pattern selectivity estimation, (ii) the reduction on the size of intermediate results and (iii) the opportunity to use merge joins instead of hash or nested-loop joins. All these factors aim at progressively reducing the cost of query processing by following the optimal order of joins.

5.2.1 Dynamic Query Plan Generation

For executing queries with multiple join patterns, TripleBit employs a Dynamic Query Plan Generation Algorithm (DQPGA) and the pseudo code is given in Algorithm 1. This algorithm consists of three parts: processing star-joins (Lines 1 - 12); further reduction (Lines 15 - 28); final join (Lines 29).

In DQPGA, a number of optimization tactics are employed to produce an optimal execution order of the join patterns for the query. Consider the example query in Fig. 5 with the optimal join order marked on the edges. First, SPARQL queries generally contain multiple star-shaped subqueries. Star joins are simple tree queries [6] and impose restrictions on the common variables. The query processor can reduce intermediate results by executing star joins before other types of join queries. The second tactic is to use semi-joins [5, 6, 20]. So we can further reduce the number of bindings involved in the subsequent join operations. For example, considering $P4 \ltimes P2$, the bindings of $P4$, which do not match $P2$, are removed. Semi-joins also reduce the amount of computation, such as sorting or hashing, required to perform subsequent joins that are more expensive. In addition, to determine an optimal execution plan, we consider three types of selectivity: triple pat-

Algorithm 1 Dynamic Query Plan Generation

Input: $queryGraph$

1: $jVars$=getJoinVar($queryGraph$);
2: **while** $jVars$ IS NOT $NULL$ **do**
3: $var = getVarwithMaxSel(jVars)$;
4: p=getPatternwithMaxSel(var);
5: **for** each pattern t adjacent to p **do**
6: $e.v1 = p$; $e.v2 = t$; $e.sel$=$sel(p) \times sel(t) \times factor$;
7: insert($jEdges, e$);
8: sortjoinSelectivity($jEdges$);
9: **for** $i \leftarrow 0$ to $sizeof[jEdges]$ - 1 **do**
10: semi-join($jEdges[i]$);
11: remove the patterns which only contains one join variable var from $queryGraph$;
12: remove var from $jVars$;
13: **if** $queryGraph$ is star joins **then**
14: goto 29;
15: $p = getPatternwithMaxSel(queryGraph)$;
16: $flag = true$; $jEdges = NULL$;
17: **while** $jEdges$ IS NOT $NULL\ OR\ flag = true$ **do**
18: **for** each pattern t adjacent to p **do**
19: $e.v1 = p$; $e.v2 = t$; $e.sel$=$sel(p) \times sel(t) \times factor$;
20: **if** $visited[e] == NULL$ **then**
21: insert($jEdges, e$); $visited[e]$=$false$;
22: **else**
23: **if** $visited[e] == false$ **then**
24: update($jEdges, e$);
25: sortjoinSelectivity($jEdges$);
26: e=getfirstEdge($jEdges$);
27: semi-join(e); remove($jEdges$,e); $visited[e]$=$true$;
28: p=$e.v1 == p?e.v2 : e.v1$; $flag = false$;
29: generate plan following the above steps and execute the plan using full joins.

tern selectivity, variable selectivity and join selectivity. Triple pattern selectivity is computed as the fraction of the number of triples which match the triple pattern [21]. We refer to the largest triple pattern selectivity as the variable selectivity. The join selectivity is the product of the selectivity estimates of the two join patterns.

DQPGA begins by selecting the star sub-query associated with the maximum variable selectivity, orders its edges based on their join selectivity and adds them to the query plan (Lines 1 - 12 in Algorithm 1). In the example of Fig. 5, it is the star query associated with $?y$. In star query $?y$ (we name the star query using its common variable), we first choose the pattern node with the largest selectivity, namely $P2$. Then we compute the join selectivity, which is the product of selectivity of two join patterns, namely the SO-join between $P2$ and $P4$ and the SO-join between $P2$ and $P5$, denoted by the two edges directly connected with $P2$ from $P4$ and $P5$. By comparing the join selectivity, we determine the join order with the SO-join between $P2$ and $P4$ first and followed by the SO-join between $P2$ and $P5$. Now the query processor will execute each of the two SO-joins using semi-join. A semi-join between two patterns can be implemented in two ways, for example, $P4 \ltimes P2$ or $P2 \ltimes P4$. To reduce the communication cost incurred during semi-join of the two patterns, TripleBit chooses the semi-join order having the largest selectivity to reduce the bindings of the other pattern having smaller selectivity. Considering the semi-join between $P2$ and $P4$, TripleBit will execute $P4 \ltimes P2$ instead of $P2 \ltimes P4$. Once the star sub-query is processed, the patterns containing one variable (e.g. $P2$) can be removed from the query graph because bindings of those patterns are joined with other patterns. Similarly, the bindings of patterns, such as $P4$ may be dropped and the triple pattern selectivity may change. We compute variable selectivity again, order the remaining variable nodes and process next star sub-query associated with the maximum variable selectivity after all patterns associated to the first star query $?y$ is processed. The query processor will repeat this process until all variable nodes are processed.

After all variable nodes are processed, for cyclic queries and bridge queries, the query processor will repeat the similar procedure as mentioned above. Concretely, the query processor will choose the pattern with the largest triple pattern selectivity from the remaining patterns, order the edges based on their join selectivity, add them to the query plan and execute it using semi-joins (Lines 15 - 28 in Algorithm 1). Consider Fig. 5, the query of the remaining patterns ($P4$, $P5$, $P6$) is a cyclic query. $P4$ is most selective as it has the largest selectivity. By comparing the join selectivity of $P4$ and $P6$, $P4$ and $P5$, we can determine the join order by executing the join between $P4$ and $P6$ first and followed by the join between $P4$ and $P5$.

Once the above process ends, the query processor will generate a final plan for the remaining patterns, execute the plan using full joins (Lines 29 in Algorithm 1), and final results will be generated.

5.2.2 Reducing the Size of Intermediate Results

The query response time can be further improved if we can minimize the size of intermediate results produced during query evaluation. 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. The compact design of TripleBit reduces the size of intermediate data in several ways. First, TripleBit does not load triples of the form (x, y, z), but (x, y) pairs. Thus, the size of the intermediate results is at most $\frac{2}{3}$ of the size of the intermediate results of (x, y, z) format. Second, TripleBit uses less indexes for each query, which leads to less data loaded into main memory during query evaluation.

Furthermore, TripleBit reduces the number of triples with matching patterns in two phases: initializing patterns and join processing.

To initialize the triple patterns involved in a query, TripleBit considers the minimal and maximal IDs of matching triples of adjacent patterns when loading bindings of a pattern. For example, in Fig. 5, the query processor first initializes $P2$ since it has the largest selectivity. When triples matching $P2$ are loaded, the bindings of join variable $?y$ should be bounded. Obviously, it is not necessary to load the triples beyond the boundaries even though they are bindings of $P4$. TripleBit will filter those triples beyond the boundaries. Filtering before materializing is super efficient for star queries, such as Q1, Q2 of LUBM. By filtering early, TripleBit reduces the intermediate results when loading triples.

When processing queries using semi-joins, the query processor tries to further reduce the bindings of triple patterns. Consider Fig. 5, $P4 \ltimes P2$, $P5 \ltimes P2$. Those bindings of the former, which have no matching in the latter are dropped. Hence, the constraints on join variable bindings of the latter are propagated to the former.

By reducing intermediate results, we gain two benefits: (i) we achieve lower memory bandwidth usage and (ii) we accomplish the computation of joins with smaller intermediate results.

5.2.3 Join Processing

The most efficient way to execute star joins is merge-join which is faster than hash join or nested-loop join [17]. TripleBit uses merge joins extensively. This entails preserving interesting orders. The bindings of joins of triple patterns with a given predicate p are either SO ordered pairs or OS ordered pairs in TripleBit storage. Note that the second elements of these pairs are also sorted if they

Table 5: Dataset characteristics

Dataset	#Triples	#S	#O	#($S \bigcap O$)	#P
LUBM 10M	13,879,970	2,181,772	1,623,318	501,365	18
LUBM 50M	69,099,760	10,857,180	8,072,359	2,490,221	18
LUBM 100M	138,318,414	21,735,127	16,156,825	4,986,781	18
LUBM 500M	691,085,836	108,598,613	80,715,573	24,897,405	18
LUBM 1B	1,335,081,176	217,206,844	161,413,041	49,799,142	18
UniProt	2,954,208,208	543,722,436	387,880,076	312,418,311	112
BTC 2012	1,048,920,108	183,825,838	342,670,279	165,532,701	57,193

Table 6: LUBM 500M (time in seconds)

	Q1	Q2	Q3	Q4	Q5	Q6	Geom. Mean
#Results	10	10	0	8	2528	219772	
Cold caches							
RDF3X	0.2684	0.2077	21.9648	0.2473	1198.6520	295.0343	6.8911
MonetDB	279.4118	281.0015	366.9872	283.1664	524.3888	468.7374	355.1170
TripleBit	**0.0319**	**0.0328**	**9.3829**	**0.0824**	**23.4303**	**26.2114**	**0.8900**
Warm caches							
RDF-3X	0.0013	0.0027	16.8547	0.0035	1114.5000	45.9599	0.4687
MonetDB	1.4475	1.2411	66.8715	3.3124	45.1842	86.2431	10.7585
TripleBit	**0.0001**	**0.0002**	**3.5898**	**0.0009**	**14.2093**	**17.7789**	**0.0504**

have the same first element. Thus, merge joins can be utilized naturally. For the triple patterns with un-restricted predicates, such as $(s\ ?p\ ?o)$, $(?s\ ?p\ o)$ or $(?s\ ?p\ ?o)$, the bindings are PSO ordered or POS ordered. Given that the intermediate data format is often ordered pairs, we say that TripleBit by design facilitates merge joins.

Considering the join $P4 \ltimes P2$, the query processor loads OS ordered pairs to initialize $P2$ because there exists two bounded components in $P2$. For the subsequent join, we transform the bindings of $P2$ into SO ordered pairs easily because O is a fixed value. TripleBit can load OS ordered pairs or SO ordered pairs to initialize $P4$ because S and O of the pattern are variables. Considering the subsequent merge-join with $P2$, TripleBit loads OS ordered pairs to initialize $P4$ since the bindings of $P2$ are SO ordered pairs.

TripleBit makes use of order-preserving merge-joins whenever possible. If the intermediate results are not in an order suitable for a subsequent merge-join (e.g., $P4 \ltimes P1$), we can transform them into suitable ordered pairs such that merge sort can be used efficiently. The transformation is cheap because the bindings of most patterns are (x, y) ordered pairs.

If it is expensive to transform bindings of two patterns to an order suitable for later merge join, TripleBit will switch to hash-joins. The query processor will also execute hash joins when intermediate results of triple joins are not in the form of $(x,\ y)$ pairs, such as $P4 \bowtie P5$ in Fig. 5.

6. EVALUATION

TripleBit was implemented using C++, compiled with GCC, using -O2 option to optimize. In this section we evaluate the TripleBit against some existing popular RDF stores using the known RDF benchmark datasets. We choose RDF-3X (v0.3.5), MonetDB (2010.11 release) and BitMat [5] for our evaluation, since they show much better performance than many others [5, 18]. BitMat did not have a dictionary component and cannot translate strings to IDs and IDs to strings [5]. Thus, we only use it in some selected experiments, such as core execution time comparison (Table 8).

All experiments are conducted using the three well-known benchmarks: LUBM [13], UniProt (2012.2 release) [2] and Billion Triples Challenge (BTC) 2012 data set [1]. Table 5 gives the characteristics of the datasets. All experiments (except those in Table 8, 9, 10) were conducted on a server with 4 Way 4-core 2.13GHz Intel Xeon CPU E7420, 64GB memory; Red Hat Enterprise Linux Server 5.1 (2.6.18 kernel), 64GB Disk swap space and one SAS local disk with 300GB 15000RPM. The server is also connected with a storage system which consists of 20 disks, each of which is 1TB 7200 RPM. Other experiments (Table 8, 9, 10) were running on a server with the same configuration except that its OS is CentOS 5.6 (2.6.18 kernel) and it did not connect with the storage system.

Table 7: LUBM 1 Billion (time in seconds)

	Q1	Q2	Q3	Q4	Q5	Q6	Geom. Mean
#Results	10	10	0	8	2528	439997	
Cold caches							
RDF-3X	0.3064	0.3372	53.3966	0.3616	2335.8000	496.1967	11.4992
MonetDB	560.3912	562.0915	3081.1862	579.2889	966.9349	4929.8527	1178.575
TripleBit	**0.0623**	**0.0721**	**13.2015**	**0.1034**	**36.8445**	**53.1482**	**1.5132**
Warm caches							
RDF-3X	0.0018	0.0029	33.4861	0.0035	2227.5900	91.5595	0.7069
MonetDB	2.5824	2.4552	910.7387	6.8218	95.7446	4608.3609	50.8953
TripleBit	**0.0002**	**0.0002**	**7.5977**	**0.0009**	**27.2772**	**36.5613**	**0.0805**

6.1 LUBM dataset

To evaluate how well TripleBit can scale, we used 5 LUBM datasets of varying sizes generated using LUBM data generator [13] (Table 5). We run almost the same set of LUBM benchmark queries as [5] did, except that Q4 and Q2 in [5] are similar, so we modify Q2 in [5] and name it as $Q4$. Given that Q6 in [5] is similar to Q2 in terms of both query pattern and result size, we drop Q6 when plotting the experimental results, considering the space constraint. All queries are listed in Appendix A. To account for caching, each of the queries is executed for three times consecutively. We took the average result to avoid artifacts caused by OS activity. We also include the geometric mean of the query times. All results are rounded to 4 decimal places. Furthermore, due to page limit, we only report the experimental results on LUBM-500M, LUBM-1B (Table 6, 7, best times are boldfaced) because larger datasets tend to put more stress on RDF stores for all queries.

The first observation is that TripleBit performs much better than RDF-3X and MonetDB for all queries. TripleBit outperforms RDF-3X in both the cold-cache cases and warm cache cases. The typical factors (the query run time of opponents divided by the query run time of TripleBit) in the geometric mean are among 7.5-7.7 (cold cache) and 8-10 (warm cache), and sometimes even by more than 78 (Q5 in LUBM-1B). TripleBit improves MonetDB on the cold cache time by nearly a factor of 399-778 in the geometric mean, and the warm-cache time by a factor of 213-632 in the geometric mean. Furthermore, for several queries (such as Q1, Q2, Q4 in LUBM-1B) the performance gain of TripleBit is more than a factor of 1000.

Another important factor for evaluating RDF systems is how the performance scales with the size of data. It is worth noting that the TripleBit scales linearly and smoothly (no large variation is observed) when the scale of the LUBM datasets increases from 500M to 1 Billion triples. Furthermore, the experimental results show that the storage and index structures in TripleBit are compact and efficient such that only data relevant to the queries will be accessed most of the time. Thus the time spent for accessing data in TripleBit is directly related to the type of query patterns, but less sensitive to the scale of data in the RDF store. For example, the variation in the execution times of Q1, Q2, Q4 for 2 LUBM data sets are 0-0.0001s (warm cache). The variation in the execution times of Q3, Q5, Q6 for the two datasets are larger. This is because intermediate results related with the query patterns increase with the scale of the dataset. The query processor needs to access more chunks and perform more joins, thus the query run-time increases. In fact, this set of experiments also shows that TripleBit prevails over RDF-3X and MonetDB in terms of the size of intermediate results and the compactness of its storage and index structures.

We also compared our core query execution time (without dictionary lookup) with RDF-3X, MonetDB and BitMat in Table 8 (time in seconds). In this set of experiments, we execute the queries on RDF-3X and TripleBit and obtain their core query execution time

Table 8: LUBM 500M (excluding dictionary lookup)

	Q1	Q2	Q3	Q4	Q5	Q6	Geom. Mean
Cold caches							
BitMat	0.2669	394.2805	**3.9379**	28.2388	193.9220	123.0972	25.5679
RDF-3X	0.1190	0.1260	18.9420	0.1700	1139.3210	207.1380	4.7437
MonetDB	12.287	11.963	125.371	236.139	147.849	287.328	75.4758
TripleBit	**0.0318**	**0.0327**	9.3829	**0.0823**	**23.4196**	**25.5188**	**0.8848**
Warm caches							
BitMat	0.2190	381.0911	**1.9379**	26.0217	190.6282	119.9245	21.4058
RDF-3X	0.0010	0.0020	16.4640	0.0030	1115.5120	46.0710	0.4146
MonetDB	0.099	0.094	53.033	2.273	22.861	24.472	2.9260
TripleBit	**0.0001**	**0.0002**	3.5898	**0.0008**	**14.1987**	**17.0969**	**0.0440**

by inserting performance measurement codes. However, it is not easy to get the core query execution time of MonetDB. BitMat did not have dictionary components. Thus, to provide a meaningful comparison, for MonetDB and BitMat, we load MonetDB and BitMat by inserting the integer IDs generated out of RDF-3X dictionary mapping [5]. TripleBit shows better performance than RDF-3X, MonetDB and BitMat (except Q3). Moreover, by Table 6, 8, TripleBit shows better performance in both queries with small result set and queries with large result set. Thus, the dictionary lookup is not a dominating factor for the better performance of TripleBit. For Q3, BitMat is the best, because the result set of Q3 is empty and BitMat has a mechanism to check early stop condition [5]. Once BitMat knows that the query returns zero result, it terminates the query evaluation immediately.

In summary, TripleBit outperforms the other three systems thanks for the following three design characteristics:

Compact design in storage and index. The saving from compact storage and index structure design also leads to efficient memory utilization and reduced I/O cost. For example, the run-time of RDF-3X increases rapidly for Q5 and Q6 because these queries produce much larger intermediate results, and thus require more time for I/O and sometimes it may induce I/O burst (e.g., many memory swap in/out). Concretely, in LUBM-500M there are 52M triples matching $P6$ of Q5 (Fig. 5). For initializing $P6$ of Q5, RDF-3X needs to load and decompress about 600MB data in a short time in addition to loading aggregate indexes to help the selection of permutation indexes. However, TripleBit only requires about 278MB because its intermediate results are SO pairs or OS pairs. It further reduces intermediate results using the techniques described in Section 5.2.2. Also TripleBit needs not decompress the data. In comparison, RDF-3X places heavier burden on I/O and requires much time for decompression. Furthermore, the storage structure of TripleBit allows more efficient data access than RDF-3X and BitMat as we discussed in Section 3.4.

Efficient indexes. First, index scan using TripleBit is fast as we discussed in Section 4.1. Second, TripleBit is fast in selecting the suitable indexes from many available indexes. For example, to evaluate Q3, RDF-3X needs to access 3 aggregate indexes and 3 permutation indexes whereas TripleBit only accesses one aggregate index and the ID-Chunk index. Consequently, RDF-3X loads more indexes into memory for scan. It also requires much memory to hold the indexes.

Join processing. TripleBit mainly uses merge join for join processing in star queries (Q1, Q2) and bridge queries (Q4). RDF-3X employs merge join, hash join and nested loop join depending on actual cases. However, by storing $(x,\ y,\ z)$ triples, RDF-3X has less opportunities to execute merge join than TripleBit does. BitMat reduces bindings of each pattern in the reduction phase and then processes join using nested loop join [5]. In star queries and bridge queries (Q1, Q2, Q4), TripleBit improves BitMat and MonetDB on the cold cache time by a factor of 8 to 12507, and the warm-cache time by a factor of 2190 to 1,905,455 because these queries have one highly selective pattern. For such queries, RDF-3X is also faster than BitMat and MonetDB. Moreover, TripleBit can reduce intermediate results during query evaluation as early as possible, thus join queries have smaller intermediate results, leading to better performance in cyclic queries Q5, Q6. Another reason that TripleBit outperforms the others is due to its capability to employ merge join and hash join to process join operations. Comparing with executing LUBM Q1-Q4, it still takes TripleBit relatively much time to execute LUBM Q5, Q6, because TripleBit executes each join operation sequentially. We can overlap some join operations and thus further improve the performance of join processing.

Table 9: UniProt (time in seconds)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Geom.
#Results	0	0	26	1	14	838568	4196	167	Mean
Cold caches									
RDF-3X	0.4039	0.3047	0.5046	0.3152	0.3859	377.4531	16.7898	3.0905	1.8675
MonetDB	98.2642	41.8234	>30min	26.2415	56.9313	792.3241	112.4354	76.3957	>88.2798
TripleBit	**0.0558**	**0.0352**	**0.1701**	**0.0864**	**0.1159**	**8.4504**	**0.6447**	**1.4522**	**0.2678**
Warm caches									
RDF-3X	0.0047	0.0015	0.0099	0.0079	0.0053	14.4739	0.1679	0.8786	0.0398
MonetDB	15.5636	3.4345	>30min	0.0032	0.0153	26.7242	17.1546	3.5359	>1.2293
TripleBit	**0.0002**	**0.0001**	**0.0032**	**0.0002**	**0.0008**	**6.1344**	**0.0474**	**0.6862**	**0.0061**

Table 10: BTC 2012 dataset (time in seconds)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Geom.
#Results	4	2	1	4	13	1	664	321	Mean
Cold caches									
RDF-3X	0.3506	0.3205	0.392	0.7232	0.7658	0.675	6.5977	6.5589	0.9585
MonetDB	>30min		0.601	>30min		0.413	106.3342	>30min	>2.9774
TripleBit	**0.0785**	**0.1432**	**0.0705**	**0.2834**	**0.1969**	**0.2966**	**1.5299**	**3.1717**	**0.2989**
Warm caches									
RDF-3X	0.0047	0.0046	0.0061	0.0114	0.0852	0.0204	0.569	1.0528	0.0334
MonetDB	>30min		0.0249	>30min		0.0262	0.3806	>30min	>0.0629
TripleBit	**0.0005**	**0.0012**	**0.0003**	**0.0041**	**0.0033**	**0.0028**	**0.2504**	**0.0232**	**0.0038**

6.2 UniProt

UniProt is a protein dataset [2]. We choose two queries from [16], one query from [5], and design five additional queries in order to compare the performance of the four RDF stores using a representative set of queries. All queries are listed in Appendix B. Among 8 queries, Q1, Q3, Q6, Q7 are bridge joins, and Q2, Q4, Q5 are star joins. Q8 is a query with loop joins. Table 9 shows the results. BitMat cannot handle datasets over 1 billion triples and thus cannot be included in the results of the experiments. TripleBit again outperforms RDF-3X and MonetDB for all queries in both cold and warm cache. Comparing with RDF-3X, TripleBit gains the performance by factor of 6.97 (cold cache) and 6.47 (warm cache) in geometric mean. For MonetDB, the typical factor ranges 19-500, and sometimes higher than 77,818 (e.g. Q1). For 4 queries: Q1, Q2, Q3, Q7, TripleBit gained more than 361x improvement over the MonetDB in warm cache. In summary, TripleBit reduces the geometric means to 0.2678s (cold) and 0.0061s (warm), which is significantly faster than RDF-3X and MonetDB.

6.3 BTC 2012 Dataset

Billion Triples Challenge (BTC) 2012 dataset was crawled during May/June 2012 and provided by the Semantic Web Challenge 2012 [1]. BTC dataset is a composition of multiple RDF web sources. We ignored those redundant triples that appeared many times in the dataset. This resulted in 1,048,920,108 unique triples (Table 5). An obvious feature of the BTC dataset, different from other datasets, is that there are 57,193 distinct predicates. The space consumption of RDF-3X, MonetDB and TripleBit for this dataset is shown in Table 11, 12. We ran almost the same set of benchmark queries as those in [16] (See Appendix C). The sizes of results returned by the queries are not big: from 1 to 664. Predicates of some triple patterns in Q1, Q2, Q4, Q5, Q8 are blank nodes. In the SPARQL specification, blank nodes are treated as non-distinguished variables [23]. It is not an issue for RDF-3X

Table 11: Storage space in GB

	LUBM 10M	LUBM 50M	LUBM 100M	LUBM 500M	LUBM 1B	UniProt	BTC 2012
RDF-3X	0.67	3.35	6.83	34.84	69.89	145.74	81.32
MonetDB	0.35	1.7	3.5	22.8	45.6	78.34	46.98
TripleBit	0.42	2.39	4.88	22.01	44.5	63.81	53.08

Table 12: Storage space (Excluding dictionary) in GB

	LUBM 10M	LUBM 50M	LUBM 100M	LUBM 500M	LUBM 1B	UniProt	BTC 2012
RDF-3X	0.40	2.00	4.12	21.16	42.49	98.17	43.65
MonetDB	0.14	0.67	1.6	5.9	12	22.04	8.33
BitMat	0.69	3.5	6.9	34.1	abort	abort	abort
TripleBit	0.17	1.24	2.58	11.12	22.68	28.37	23.51

and TripleBit to process queries containing non-fixed predicates, but MonetDB with the vertical partitioning approach handles this poorly [16]. The query run-times are shown in Table 10. TripleBit performs consistently the best for all queries.

6.4 Storage space

We compare the required disk space of TripleBit with RDF-3X and MonetDB in Table 11. BitMat is excluded because BitMat does not have the dictionary facility in its public released package. TripleBit outperforms RDF-3X for all datasets in storage space. The reason is that RDF-3X maintains all six permutations of S, P and O in separate indexes, plus 9 aggregate indexes [17]. The storage of TripleBit is larger than MonetDB when loading smaller datasets, such as LUBM-50M, LUBM-100M datasets. The reason is that MonetDB only stores SO pairs of RDF data. However, MonetDB requires more storage space than TripleBit when loading LUBM-500M, LUBM-1B and UniProt. For UniProt, TripleBit only needs 63.81GB storage, more compact than MonetDB and significantly more efficient than RDF-3X since TripleBit only consumes 43.8% of storage required by RDF-3X. Although BTC 2012 contains fewer triples than LUBM-1B, TripleBit consumes more storage space when loading BTC 2012 than LUBM-1B for two reasons: (i) More predicates leads to larger storage for aggregate indexes and ID-Predicate index; and (ii) strings of BTC 2012 do not share as many common prefixes as other two datasets.

Since the dictionary is usually large, Table 12 shows a comparison of the core storage (not including storage for dictionary) among 4 systems. TripleBit remains to be more compact than RDF-3X and BitMat. BitMat has the largest storage among the four systems (BitMat experiments on LUBM-1B, UniProt and BTC 2012 aborted). According to Table 11, 12, the dictionary sizes of TripleBit in all data sets are smaller than the dictionary sizes of RDF-3X.

During query processing, the memory is allocated for holding intermediate results and data structures for join processing. For example, RDF-3X will construct several hash tables for hash joins. Larger intermediate results lead to larger hash tables. Thus, the size of memory allocated for data structures used in processing join is also highly relevant with the size of intermediate results. We note that intermediate results are not only the triples matching patterns, but also the intermediate data, such as indexes loaded into memory during query evaluation. Table 13 shows a comparison of TripleBit with RDF-3X and MonetDB on the peak memory usage during the execution of LUBM Q6. LUBM Q6 is chosen because of its large intermediate results. Both the peak virtual and physical memory usage of MonetDB are the largest compared to TripleBit and RDF-3X. The query time in MonetDB also grew quickly. To some extent, this showed that the MonetDB process spent much more time in the kernel waiting for the memory pages to be allocated. Table 13 also showed that the memory consumption of RDF and TripleBit is in proportion to the result sizes. For example, the virtual memory consumption of RDF-3X in LUBM-1B is about 2 times of its virtual memory consumption in LUBM-500M. We can find the similar phenomenon in other data sets. It is also true for TripleBit. However, comparing with RDF-3X and MonetDB, TripleBit requires the smallest virtual memory, and the size of virtual memory for TripleBit grows slower than RDF-3X and MonetDB. In LUBM-1B, TripleBit's virtual memory size is about 40% of those of RDF-3X and MonetDB. These experimental results show that comparing with RDF-3X and MonetDB, TripleBit can significantly reduce the intermediate result size.

Table 13: Peak memory usage in GB

	LUBM 10M		LUBM 50M		LUBM 100M		LUBM 500M		LUBM 1B	
#Results	4,462		22,001		44,190		219,772		439,997	
	virtual	phy.	virtual	phy.	virtual	phy.	virtual	phy.	virtual	phy.
RDF-3X	0.793	0.145	4.058	0.816	8.249	1.6	42.1	9.6	84.1	18
MonetDB	1.577	0.851	5.104	4	10.5	8.1	44.8	40	89.3	61
TripleBit	0.713	0.331	2.818	1.8	5.545	3.7	23.7	16	47.8	33

7. CONCLUSION AND FUTURE WORK

We have presented TripleBit, a fast and compact system for large scale RDF data. TripleBit is both space efficient and query efficient with three salient features. First, the design of a triple matrix storage structure allows us to utilize the variable-size integer encoding of IDs and the column-level compression scheme for storing huge RDF graphs more efficiently. Second, the design of the two auxiliary indexing structures, ID-Chunk bit matrix and ID-Predicate bit matrix, allow us to reduce both the size and the number of indexes to the minimum while providing orders of magnitude speedup for scan and merge-join performance. In addition, the query processing framework of TripleBit best utilizes its compact storage and index structures. Our experimental comparison shows that TripleBit consistently outperforms RDF-3X, MonetDB, BitMat and delivers up to 2-4 orders of magnitude better performance for complex join queries over large scale RDF data.

Our work on TripleBit development continues along two dimensions. First, we are working on extending TripleBit for scaling big RDF data using distributed computing architecture. Second, we are interested in exploring the potential of using TripleBit as a core component of RDF reasoners [15] to speedup the reasoning using conjunctive rules.

8. ACKNOWLEDGMENTS

We would like to thank all reviewers for their valuable suggestions. The research is supported by National Science Foundation of China (61073096) and 863 Program (No.2012AA011003). Ling Liu is partially supported by grants from NSF NetSE program, SaTC program and Intel ISTC on Cloud Computing.

9. REFERENCES

[1] Semantic web challenge 2012. http://challenge.semanticweb.org/2012/.

[2] UniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.

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

[4] S. Álvarez García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k^2-triples for full-in-memory RDF engines. In *Proc. of AMCIS 2011*.

[5] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix bit loaded: A scalable lightweight join query processor for RDF data. In *Proc. of WWW 2010*, pages 41–50. ACM, 2010.

[6] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the Associanon for Computing Machinery*, 28(1):25–40, 1981.

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

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

[9] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *Proc. of ISWC/ASWC2007*, pages 211–224.

[10] O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In *Proc. of ESWC 2007*, pages 564–578.

[11] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134.

[12] M. Janik and K. Kochut. BRAHMS: A workbench RDF store and high performance memory system for semantic association discovery. In *Proc. of ISWC 2005*, pages 431–445. Springer, Berlin, 2005.

[13] LUBM. http://swat.cse.lehigh.edu/projects/lubm/.

[14] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *Proc. of 16th ADC*.

[15] B. Motik, I. Horrocks, and S. M. Kim. Delta-reasoner: a semantic web reasoner for an intelligent mobile platform. In *Proc. of WWW 2012*. ACM, 2012.

[16] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proc. of SIGMOD 2009*, pages 627–639. ACM, 2009.

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

[18] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1-2):256–263, 2010.

[19] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: Not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[20] K. Stocker, D. Kossmann, R. Braumandl, and A. KemperK. Integrating semi-join-reducers into state of the art query processors. In *Proc. of ICDE 2001*, pages 575–584, 2001.

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

[22] SWEO Community Project. Linking open data on the semantic web. http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData.

[23] W3C. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008.

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

[25] K. Wilkinson. Jena property table implementation. In *Proc. of SSWS 2006*, pages 35–46, 2006.

[26] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, March 2006.

[27] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficiently querying RDF data in triple stores. In *Proc. of WWW 2008*, pages 1053–1054. ACM, 2008.

[28] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *PVLDB*, (8):482–493, 2011.

APPENDIX

A. LUBM QUERIES

PREFIX r: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/∼zhp2/2004/0401/univbench.owl#>

Q1-Q3: Same as Q5, Q4, Q3 respectively in [5].

Q4: SELECT ?x WHERE {?x ub:worksFor <http://www.Department0.-University0.edu>. ?x r:type ub:FullProfessor . ?x ub:name ?y1 . ?x ub:emailAddress ?y2 . ?x ub:telephone ?y3 .}

Q5-Q6: Same as Q1 and Q7 respectively in [5].

B. UNIPROT QUERIES

PREFIX r: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX u: <http://purl.uniprot.org/core/>

Q1: Same as Q6 in [5].

Q2-Q3: Same as Q1, Q3 respectively in [16].

Q4: SELECT ?a ?vo WHERE { ?a u:encodedBy ?vo. ?a s:seeAlso <http://-purl.uni-prot.org/refseq/NP_346136.1>. ?a s:seeAlso <http://purl.uni-prot.org/tigr/SP_1698>. ?a s:seeAlso <http://purl.uniprot.org/pfam/-PF00842>. ?a s:seeAlso <http://purl.uniprot.org/prints/PR00992>. }

Q5: SELECT ?a ?vo WHERE { ?a u:annotation ?vo. ?a s:seeAlso <http://-purl.uniprot.org/interpro/IPR000842>. ?a s:seeAlso <http://purl.uni-prot.org/geneid/-945772>. ?a u:citation <http://purl.uniprot.org/citations/9298646>. }

Q6: SELECT ?p ?a WHERE { ?p u:annotation ?a . ?p r:type uni:Protein . ?a r:type <http://purl.uniprot.org/core/Transmembrane_Annotation> . ?a u:range ?range . }

Q7: SELECT ?p ?a WHERE { ?p u:annotation ?a . ?p r:type uni:Protein . ?p u:organism taxon:9606 . ?a r:type <http://purl.uniprot.org/core/Disease_Annotation> . ?a rs:comment ?text . }

Q8: SELECT ?a ?b ?ab WHERE { ?b u:modified "2008-07-22" . ?b r:type uni:Protein . ?a u:replaces ?ab . ?ab u:replacedBy ?b . }

C. BTC 2012 QUERIES

PREFIX geo: <http://www.geonames.org/>
PREFIX pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX dbpedia: <http://dbpedia.org/property/>
PREFIX dbpediares: <http://dbpedia.org/resource/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

Q1: SELECT ?lat ?long where { ?a [] "Bro-C'hall" . ?a geo:ontology#in-Country geo:/countries/#FR . ?a pos:lat ?lat . ?a pos:long ?long . }

Q2: Same as Q2 in [16],

Q3: SELECT ?t ?lat ?long WHERE { ?a dbpedia:region dbpediares:List_of_-World_Heritage_Sites_in_Europe . ?a dbpedia:title ?t . ?a pos:lat ?lat . ?a pos:long ?long . ?a dbpedia:link <http://whc.unesco.org/en/list/728> . }

Q4-Q5: Same as Q4, Q5 in [16],

Q6: SELECT DISTINCT ?d WHERE { ?a dbpedia:senators ?c . ?a db-pedia:name ?d . ?c dbpedia:profession dbpediares:Politician . ?a owl:sameAs ?b . ?b geo:ontology#inCountry geo:countries/#US . }

Q7: SELECT DISTINCT ?a ?b ?lat ?long WHERE { ?a dbpedia:spouse ?b . ?a rdf:type <http://dbpedia.org/ontology/Person> . ?b rdf:type <http://dbpedia.org/-ontology/Person> . ?a dbpedia:placeOfBirth ?c . ?b dbpedia:placeOfBirth ?c . ?c owl:sameAs ?c2 . ?c2 pos:lat ?lat . ?c2 pos:long ?long . }

Q8: SELECT DISTINCT ?a ?y WHERE { ?a a <http://dbpedia.org/class/-yago/Politician110451263> . ?a dbpedia:years ?y. ?a <http://xmlns.-com/foaf/0.1/name> ?n. ?b [] ?n. ?b rdf:type <http://dbpedia.org/ontology/OfficeHolder> . }