

Under consideration for publication in Knowledge and Information  
Systems

## Dynamic and Fast Processing of Queries on Large Scale RDF Data

Pingpeng Yuan<sup>1</sup>, Changfeng Xie<sup>1</sup>, Hai Jin<sup>1</sup>, Ling Liu<sup>2</sup>, Guang Yang<sup>1</sup>  
and Xuanhua Shi<sup>1</sup>

<sup>1</sup>Services Computing Technology and System Lab., School of Computer Science and Technology,  
Huazhong University of Science and Technology, Wuhan, China;

<sup>2</sup>Distributed Data Intensive Systems Lab., School of Computer Science, College of Computing,  
Georgia Institute of Technology, Atlanta, USA

**Abstract.** As RDF data continues to gain popularity, we witness the fast growing trend of RDF datasets in both the number of RDF repositories and the size of RDF datasets. Many known RDF datasets contain billions of RDF triples (subject, predicate and object). One of the grand challenges for managing this huge RDF data is how to execute RDF queries efficiently. In this paper, we address the query processing problems against the billion triple challenges. We first identify some causes for the problems of existing query optimization schemes, such as large intermediate results, initial query cost estimation errors. Then we present our block oriented dynamic query plan generation approach powered with pipelining execution. Our approach consists of two phases. In the first phase, a near optimal execution plan for queries is chosen by identifying the processing blocks of queries. We group the join patterns sharing a join variable into building blocks of the query plan since executing them first provides opportunities to reduce the size of intermediate results generated. In the second phase, we further optimize the initial pipelining for a given query plan. We employ optimization techniques, such as sideways information passing and semijoin, to further reduce the size of intermediate results, improve the query processing cost estimation and speedup the performance of query execution. Experimental results on several RDF datasets of over a billion triples demonstrate that our approach outperforms existing RDF query engines that rely on dynamic programming based static query processing strategies.

**Keywords:** query processing; plan generation; query plan graph; operator

---

*Received May 06, 2013*

*Revised Aug 16, 2013*

*Accepted Nov 16, 2013*

## 1. Introduction

RDF datasets are rapidly growing in both numbers and sizes. The number of RDF datasets that exceed billions of triples continues to grow. The explosion of Big RDF Data puts forward several technical challenges for data scientists. Another attractiveness of RDF data is that many big RDF datasets remain to be open and publically available, represented by the Semantic Web Challenges (<http://challenge.semanticweb.org>) and Linked Open Data Project (SWEO Community Project; 2010). Billion Triple Challenge dataset (Semantic Web Challenge; 2012), collected by the Semantic web community, contains up to 3 billion triples as of 2012. One of the grant challenges of managing this huge RDF data is how to execute RDF queries, especially complex join queries efficiently, at the scale of billions of triples. An RDF triple consists of subject, predicate, object and often we refer to subject or object of an RDF triple as entities and predicate as the relationship from subject to object. RDF uses URI to name the relationship between entities, namely subject (S) and object (O). SPARQL (W3C; 2008) is a W3C standard SQL-like query language defined to query RDF data from RDF stores. The flexible features of RDF allow structured and semi-structured data to co-exist, be exposed and shared across different applications by using the RDF data model. Today RDF data has been used in many business, science and engineering domains, such as government, biologists, life science (UniProt; n.d.), business intelligence, social networks and Wikipedia.

Most of RDF datasets are residing in RDF stores powered with SPARQL and RDF query engines. The query engines, such as RDF-3X, employ Dynamic Programming (Neumann and Weikum; 2009, 2010a) to generate an optimized static query execution plan, which is generally an operator tree. Figure 1 shows an example of such an operator tree for benchmark query Q5 on RDF dataset LUBM (See Appendix A). Unfortunately, dynamic programming is known to have exponential time and space complexity to generate an optimal plan because it needs to search a large solution space (Kossmann and Stocker; 2000). In many scenarios (e.g., queries having large intermediate results), it is difficult to translate a declarative query into an efficient static execution plan due to the difficulty in predicting the size of intermediate results and in estimating accurate query execution cost. Furthermore, the output of an operator in each operator tree is used only by another single operator, its parent operator. As a consequence, the intermediate results are not reused by multiple operators, leading to inefficient processing of some queries on big data. For example, considering  $P1$ ,  $P4$ ,  $P6$  share variable  $?z$  in Figure 1, the number of triples matching this triple pattern is  $\#(P1) < \#(P4) < \#(P6)$  (where  $\#(P_i)$  denotes the number of triples matching triple pattern  $P_i$ ). According to the query plan shown in Figure 1, the query engine will execute  $P' = P4 \bowtie P1$ , then execute join  $P' \bowtie P6$ . Thus, the intermediate results of  $P1$  is only used to reduce  $P4$  although they can also be used to reduce  $P6$ . In fact, the cost of using  $P1$  to reduce both  $P4$  and  $P6$  and then finishing the query processing is cheaper than the cost to execute the query plan shown in Figure 1.

Big RDF datasets tend to produce large intermediate results. Before executing the operator tree, the query engine initializes the query patterns (selection and join patterns) by scan. For large RDF datasets, there are a large number of triples matching a pattern. For example, consider Figure 1 again, there are six leaf nodes, representing the inputs of the operator tree from the underlying scans. The scan for each pattern produces an intermediate relation. Some rela-

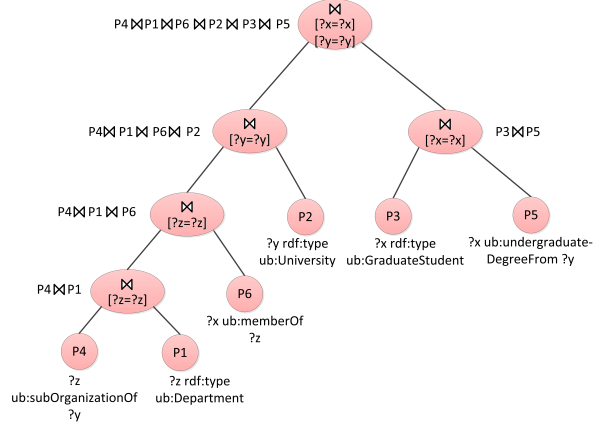


Fig. 1. Operator tree of LUBM Q5

tions contain large number of triples. For instance, there are 52,205,394 triples in the dataset LUBM-500M (Table 1) matching  $P6$ . For the same queries, the bigger the dataset is, the larger the intermediate results are. For example, there are 104,403,077 triples in LUBM-1B matching  $P6$  which is about twice of intermediate results matching  $P6$  in LUBM-500M. Intermediate results are not only the number of triples matching patterns, but also the size of intermediate data loaded into memory during query evaluation. Suppose that each triple is encoded using three integer IDs as is done in most of RDF stores, then the sizes of the relation matching  $P6$  in LUBM-500M and LUBM-1B are about 600MB and 1,195MB respectively. For the join to be performed, the intermediate results involved in the join must be loaded to the main memory for CPU computation. Thus, huge amount of I/O communication takes place, which slows down the join.

By understanding the main problems of answering SPARQL queries on big RDF data, such as the need for processing bigger and more complex intermediate results, we propose a block-based pipelining approach for dynamic and fast processing of SPARQL queries in this paper. Our approach has three unique features: First, we use graphs instead of trees to represent query plan. Our approach considers the blocks of graph, which share same join variables as the basic query processing units in order to reduce intermediate results. When building query plans as graphs, operators can easily reuse intermediate results produced by the proceeding operators because all triples in a block have a same join variable. Second, we develop a two phase join processing framework, which first generates an initial query execution plan and then iteratively refine the plan as more accurate estimation of intermediate result size and the execution cost is obtained. For plan generation, we present a simple but effective selectivity estimation method based on block of triple patterns. For iterative plan execution, we employ pipelining technique to select the next join operations dynamically. Third but not the least, to further reduce the size of intermediate results, we also employ optimization techniques, such as light-weight sideways information passing, semi-joins and sort merge join. We evaluate our approach through extensive experiments on open RDF datasets with up to 2.9 billion triples. Our experimental comparison with RDF-3X and other existing RDF systems shows that our approach consistently

outperforms existing representative RDF query engines, such as RDF-3X and TripleBit.

The remainder of this paper is organized as follows. First, we give an overview of related work in section 2. Section 3 describes a RDF store TripleBit where our methods are implemented. Section 4 presents our query plan generation approach and query plan executing technologies. Section 5 presents the optimized technologies of the query processor. We evaluate the technologies in Section 6. Then, we conclude the paper with a summary and future work in Section 7.

## 2. Related work

Since the RDF data is growing rapidly, query on the large scale RDF data is a critical issue. A lot of RDF engines which focus on the query performance have been proposed, such as YARS2 (Harth et al.; 2007), RDF-3X (Neumann and Weikum; 2009, 2010a,b), SHARD (Rohloff and Schantz; 2010), SpiderStore (Binna et al.; 2010), HexaStore (Weiss et al.; 2008), BRAHAMS (Janik and Kochut; 2005), gStore (Zou et al.; 2011), C-Store (Abadi et al.; 2007) et al. The storage structure of these systems can be roughly classified into following categories (Yuan et al.; 2013): 3-columns row store, property table, column store with vertical partition (Abadi et al.; 2007) and graph store. Three column row store stores the RDF triples in a natural way, but it suffers from too many self-joins. So RDF-3X and HexStore build several clustered B+-trees for all permutations of three columns. To avoid too many self-joins, property table clusters the properties of the subjects which tend to occur all together. However, this kind of storage will waste too much space because of large number of sparse properties. Column store with vertical partitioning (Abadi et al.; 2007) uses one table which has only two columns to store one predicate. Obviously, the shortage of it is the scalability when the number of predicates increases. gStore uses an adjacency list table to store the properties. It is a property table inherently, and it uses a list to skip the null property of the subjects.

In order to accelerate the SPARQL query processing, some RDF engines, such as RDF-3X and HexaStore use additional indexes on combinations of S, P, and O. With all possible permutations of SPO triples which are compressed to reduce the space consuming, the query processing engine chooses the best index to get the candidate answers. On the other hand, (Udrea et al.; 2007) and (Yan et al.; 2004) present a novel index based on the graph, and indexes all the paths and SPO labels, however, it brings some difficult to query optimization. gStore transforms the RDF graph into a data signature graph by encoding entity and class vertex, and uses a novel index (named VS\*-tree) to speed up the query. Huang etc (Huang et al.; 2011) partitions the RDF data in several data nodes and decomposes SPARQL queries into high performance that take advantage of how data is partitioned in a cluster.

No matter how well the storage and index is designed, it is essential to estimate the selectivity of triple patterns in the query which can help the query optimizer generate optimal query plan. For single triple patterns, RDF-3X pre-compute the selectivity of all permutations, such as SP, OP, SO, PS, PO, OS, S, P, O. For the joins, (Stocker et al.; 2008) proposed a heuristics based method which ranges from simple to sophisticate to estimate the selectivity, but the heuristics must be customized. Other systems pre-compute the frequent paths, and keep the statistics. But this should be done by graph mining techniques.

Join operation is expensive and different orders of join operation result in large performance variations. It is a critical issue to determine joins order in query optimization. Dynamic programming (Selinger et al.; 1979) was used in most query optimizers (Kossmann and Stocker; 2000). It will construct millions of partial plans in the search phase and thus results in very memory intensive and computation-intensive operations. While this algorithm produces good optimization results, its high complexity can be prohibitive. And dynamic programming also relied on an optimal substructure of a problem. D. Kossmann etc. proposed Iterative Dynamic Programming (Kossmann and Stocker; 2000). The main idea of IDP is to apply dynamic programming several times in the process of optimizing a query.

Within a query plan, typically two execution methods are used: pipelining and materialization. Pipelining is typically realized by a tree of iterators that implement the physical operations. An iterator allows a consumer to get the results of an operation separately, one at a time. The tuple-at-a-time approach is elegant, simple to understand, and relatively easy to implement. However, it also results in a set of important performance drawbacks that for every tuple there are multiple function calls performed. Materialization (MonetDB; 2010) is not realized as a pipeline of operators, but instead a series of sequentially executing statements, consuming and producing columns of data. used the column-at-a-time approach: every operator is executed at once for all the tuples in the input columns. Its output is fully materialized as a set of columns. While materialization approach in many areas demonstrates a significant performance improvement over the traditional tuple-at-a-time strategy, it also suffers from a number of problems, such as high memory traffic. C. Balkesen etc analyzed the join algorithms proposed in the literature and proposed several important optimizations (Balkesen et al.; 2013). C. Kim etc reexamined hash join and sort-merge join under the latest computer architecture and current architectural trends will imply better scalability potential for sort-merge join (Kim et al.; 2009). Since SPARQL is similar as SQL, some research translates the SPARQL into SQL, and utilizes the rich work in relational database query optimization. However, the translation between the SPARQL and SQL is a critical issue which has to keep the semantics (Chebotko et al.; 2009).

### 3. TripleBit Overview

Since our technologies presented here are implemented in TripeBit (Yuan et al.; 2013), an RDF engine, we will briefly introduce its storage and indexing technologies in the following.

**Clustered storage.** We design a Triple Matrix model where RDF triples are represented as a two dimensional bit matrix. We call it the triple matrix. The triple matrix is created with subjects (S) and objects (O) as one dimension (row) and triples (T) as the other dimension (column). Thus, we can view the corresponding triple matrix as a two dimensional table with  $|S \cup O|$  columns and  $|T|$  rows. Each column of the matrix corresponds to an RDF triple. Each row is defined by a distinct entity value, representing a collection of the triples having the same entity. We group the columns according to predicates such that the triples with the same predicate will be adjacent in the matrix.

In TripleBit, we first vertically partition the matrix into multiple disjoint buckets, one per predicate (property). For each RDF dataset, we store the triple

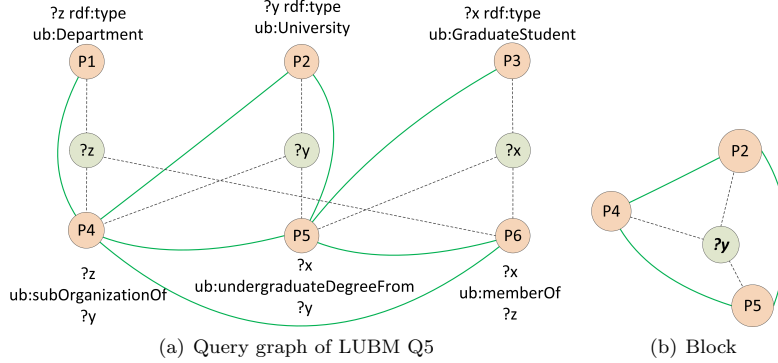
matrix physically in two duplicates, one for S-O order and another for O-S order. We use chunks, fixed-size storage spaces to store a triple matrix. The chunks having the same predicates are placed adjacently on storage media.

**Indexing Technologies.** Indexes include ID-Chunks index and ID-Predicate index are used to accelerate table scanning. ID-Chunk matrix represents the relationship between IDs (rows) and Chunks (columns). Given that chunks with the same predicate are stored consecutively on storage media and triples in each chunk are ordered by S-O or O-S, thus triples related with a subject (or object) will only be stored in several adjacent chunks and the non-zero entries of the ID-Chunk matrix are distributed around the main diagonal of the matrix. We can draw two lines or two finite sequences of straight line segments, which bound the non-zero entries of the matrix. If there are many predicates in RDF data, then it may be very time consuming to locate the candidate chunks. For the purpose of quickly executing queries with unbounded predicates, we introduce the ID-Predicate bit matrix index structure. The entries in the ID-Predicate matrix are bit and "1" indicates the occurrence relationship between IDs and predicates. With the ID-Predicate matrix, if a subject or object is known, TripleBit can determine which predicates are related with it.

#### 4. Dynamic Query Plan Generation and Pipelining Execution

For queries, the query processor must generate an execution plan for it and then execute the plan. The first problem in query plan generation is the representation of the query execution plan. The representation of plan should allow all operators are freely reorderable. We use graphs instead of trees to represent query plan. One reason is that operators can easily reuse intermediate results, as they can share children when building query plans as graphs. Thus, I/O costs can be reduced. Although there are multiple approaches to describe SPARQL query graph (Atre et al.; 2010; Hartig and Heese; 2007; Stocker et al.; 2008), here we use the similar model in (Atre et al.; 2010) to represent query graph. The query graph describes the relationships between variables and patterns as well as the logical joins between the triple patterns. In a query graph, nodes are triple patterns and variables. If there are two variables connected with at least one common triple pattern, we say these two variables dependent with each other. From the query graph, we can derive the relationships between the join variables too. Figure 2(a) is the query graph of LUBM Q5. Due to clarity, some edges, for example the join edge between P1 and P6 are omitted in the Figure 2(a) (A position marked with ? in the triple pattern is variable). There are two kinds of edges connect the nodes. One kind of edges (dashed line) connect pattern nodes with variable nodes if the corresponding variables appears in the corresponding triple patterns. Another kind of edges which are shown in solid line represent the type of join between the triple patterns.

When using a constructive plan generator, we have to determine the building blocks which will eventually be combined to form the full query. A query graph typically consists of multiple components which share common variables. For example, Figure 2(a) contains three components. Actually, the blocks of a query graph represent star shaped sub-queries. Star join query has an important feature. That is star queries impose restrictions on the common variables. Thus,



**Fig. 2.** The query graph of LUBM Q5 and one of its blocks

the query engine can reduce intermediate results. In our query plan generation method, we consider star join as basic building blocks. Since each block has a common variable node, we name the blocks using the variable nodes.  $?z$  block (Figure 2(b)) is one block of LUBM Q5 ( $P2 \bowtie P4 \bowtie P5$ ) which contains three triples  $P2$ ,  $P4$ ,  $P5$ .

Another problem is how to generate optimal plan and execute it. Our approach does not aim to produce a static optimal query plan for the whole query processing, but iteratively refine the plan as more accurate estimation of intermediate result size and the execution cost is obtained. Namely, once a plan is generated, the query engine will execute it. Then the plan generator will generate the plan again for next execution according to current statistics of intermediate results. Generally, the plan generator takes the logical representation of the query and transforms it into a physical plan. Both bottom-up and top-down approach can generate physical plans for queries. However, different from previous generators which only employ top-down approach or bottom-up approach, our plan generator combines both bottom-up approach and top-down approach together. Corresponding to bottom-up and top-down approach, our approach includes two phase. In the first phase, the generator constructs a plan that mainly consists of semi-joins. And in the second phase, a bottom-up plan is generated and performs full joins to obtain the final results. Concretely, by our approach, we first identify the blocks of query and then determine the plan of each block using top-down approach. Then the query engine will execute the plan using dynamic chunk pipelining. Finally, our plan generator starts with the plans of blocks plans and combines these plans using operators until the whole query has been constructed. By this means, the plan generator tries to construct the plan that will produce the query result with minimal costs.

In the following, we first describe how to obtain the query execution cost for a given query plan (Section 4.1). We then discuss how the query plan is generated by ordering blocks and refining it (Section 4.2) and hence the of the plan is executed (Section 4.3).

#### 4.1. Selectivity Estimation

In the query graph representation, a node can receive one or more inputs from its neighborhoods and produce one output forwarded to its neighborhoods for join.

The performance of a query plan is determined largely by the order in which the nodes (triple patterns) are joined. Join order is based on the selectivity and join cost estimation. In order to generate an optimal query plan, the query engine should estimate the selectivity of the query. The selectivity estimation includes single triple selectivity estimation, selectivity of variable and the selectivity estimation of the joins between the patterns. In order to estimate the selectivity more accurately, we build SP (for subject-predicate), OP (for object-predicate), S (for subject only), O (for object only) statistics indexes to store the pre-computed statistics information (Yuan et al.; 2013).

Triple pattern selectivity is computed as the fraction of the number of triples which are matching the triple pattern (Stocker et al.; 2008). The selectivity of single query pattern can be estimated accurately directly by locating the entity *id* in statistics indexes. Especially, when the subject and object are known but the predicate is a variable, we simply estimate the selectivity is 2, because we assume the relations between two entities is much smaller than the other types. When the triple pattern contains no variable, its result cardinality is 1 (assume the result is not empty). If its three components are unbound, the results are all triples in the system. For instance, in Figure 2(a), the number of triples matching triple patterns is  $\#(P2) < \#(P1) < \#(P4) < \#(P3) < \#(P5) < \#(P6)$ , thus, we can get the initial selectivity of the triple patterns as follows:  $SEL(P2) > SEL(P1) > SEL(P4) > SEL(P3) > SEL(P5) > SEL(P6)$  ( $SEL(p)$  represents  $p$ 's selectivity). The selectivity of a variable is computed according to the selectivity of query patterns containing the variable. Here, we considered the largest triple pattern selectivity as the variable selectivity. For example,  $P2, P4, P5$  share the same join variable  $?y$ . So,  $SEL(?y) = SEL(P2)$ . Accordingly, we know  $SEL(?y) > SEL(?z) > SEL(?x)$ .

Next, we estimate the selectivity of the joins between two query patterns (e.g.  $P1, P2$ ) according to the join types as well as the product of the selectivity of the corresponding patterns. The computation formula is shown as Formula 1.

$$sel(P_1 \bowtie P_2) = sel(P_1) \times sel(P_2) \times factor^2 \quad (1)$$

## 4.2. Ordering Blocks

The query plan is mainly focused on the join order of the query patterns and the chosen join type between the query patterns. The plan generator itself builds the query plans bottom-up, clustering those patterns sharing variables into star joins. The query engine should add building blocks with care, as the search space increases exponentially with the number of building blocks. This approach is especially interesting for two reasons. First, it is the simple approach that will construct the optimal solution and guarantee small intermediate results. Second, the building blocks are larger and high level. So, the query optimizer can be extended or improved by introducing new low level plan operators or strategies.

Ordering the blocks is typically employing selectivity estimation algorithms (see Section 4.1) to determine an optimized execution plan. Once the variable selectivity is known, we can order blocks. First, we begin with the block where variable node has the maximal variable selectivity. In the above example (Figure 2(a)), it is  $?y$ . As each block consists of multiple join edges, we must order its join edges. In the block associated with  $?y$ , we choose the pattern node with the maximal selectivity first, namely  $P2$ . Then we compute the join selectivity



which is the product of selectivity of two join patterns, namely edge selectivity which are directly connected with  $P2$ .

If we determine the join order of all the patterns of block  $?y$ , we choose the next block having the next maximal selectivity of variable, namely  $?z$ . In the block, we begin with the pattern which connects this pattern with previous block. The pattern is  $P4$  which has variables  $?y$  and  $?z$ . Then we compute the edge selectivity between  $P4$  and other patterns which have variable  $?z$ . The algorithm terminates when all blocks are processed in the graph (see Algorithm 1).

Once the initial plan is generated, query processor will execute the plan (see Section 4.3). After the plan is executed, some bindings may be dropped. And some patterns (for example  $P1$ ,  $P2$ ,  $P3$ ) may be removed from the query graph because bindings of those patterns are joined with other patterns. Thus, the selectivity of the nodes, edges may change. We need to compute selectivity again. In the situation, the query processor estimates the selectivity of triple patterns according to the intermediate results matching the triple pattern so far. Then the query processor computes selectivity of variables and selectivity of join as previous steps do. According to the selectivity, we can determine the join order of the next round. The final plan, in which final results will be produced, will be generated bottom-up when no selectivity changes.

In RDF storage systems such as RDF-3X (Neumann and Weikum; 2010a,b, 2009), they utilize dynamic programming for enumerating plans in an appropriate order. The method both comes in bottom-up and top-down manner, starting with the simplest sub-queries. If there are  $n$  triple patterns in the query, the time complexity of it is  $O(n^3)$ . In our system, we use a simple but effective query plan generation method which time complexity is  $O(n)$ . What is more, the coefficient of time complexity is much lower than others. The cost of the plan generating approach is cheaper than dynamic programming (Neumann and Weikum; 2009, 2010a). We compare our solution with the bottom-up DP of RDF-3X (Table 2). Interestingly our plan generation scheme outputs plans faster than the bottom-up DP of RDF-3X. One reason is our solution is lightweight.

### 4.3. Dynamic Chunk Pipelining

RDF stores typically exhibit less data and computation overlap, e.g., they invoke a set of operator instances (access data on the disk, scan, sort-merge or hash join etc) sequentially for each query. Sequential operations result in long query latencies. To overcome the issue, we incorporate a pipelining approach in query execution in order to reduce latency between operators. The approach combines two different techniques. First, we adopt a dynamic semi-pipelined query processing, which is a combination of task pools and parallel processing. Task pools are consisted of a set of work queues which are accessed by processes or threads used for the computation. The ratio of work queues to processes or threads can be varied, allowing for a variety of schemes with different properties. The processes get data items from and send data items to work queues depending on its cohesion, which is defined as the difference of maximal value and minimal value in data items. The work queue receives the data items and processes posting data items as soon as possible. Second, chunk instead of a single tuple is processed by the pipelining operation each time. Generally, pipelining is typically realized by a tree of iterators, each of which allows a consumer to get the results of an operation separately, one at a time (Hartig et al.; 2009). The approach is fine

**Algorithm 1** Block Oriented Dynamic Query Plan Generation

---

```

Input: queryStr
1: parse queryStr;
2: for each tp in queryStr do
3:   tp.sel = calPatternSel(tp);
4:   for each var in tp do
5:     jVars[var].tps[jVars[var].tpcount] = tp;
6:     jVars[var].tpcount += 1;
7:     if jVars[var].sel < tp.sel then
8:       jVars[var].sel = tp.sel;
9:     end if
10:  end for
11: end for
12: for each var in jVars do
13:   if jVars[var].tpcount < 2 then
14:     remove var from jVars;
15:   end if
16: end for
17: sortVariableBySelectivity(jVars);
18: while jVars IS NOT NULL do
19:   PipeliningQuery(jVars, ×);
20:   for each var in jVars do
21:     for i ← 0 to jVars[var].tpcount − 1 do
22:       if jVars[var].tps[i] has only one variable then
23:         remove jVars[var].tps[i];
24:       end if
25:     end for
26:   end for
27:   remove var from jVars;
28: end while
29: sortVariableBySelectivity(jVars);
30: PipeliningQuery(jVars, ⊔);

```

---

grained and may induce communication overhead. Hence, instead of iterator-based pipelining to process one triple-at-a-time, intermediate results are broken into smaller chunks, and the operators execute on them in the pipelined fashion. The algorithm is shown in Algorithm 2.

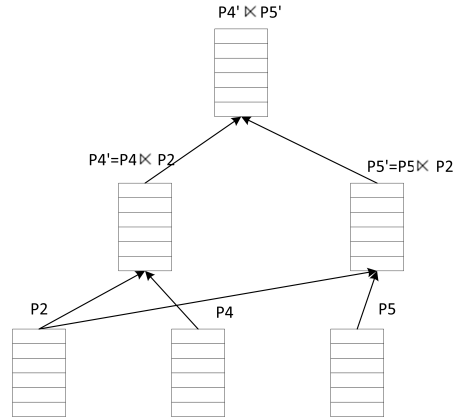
Considering the block of query shown in Figure 2(b), some of its pipelines is shown in Figure 3. The edges show possible pipelining. The actual pipelining depends on the cohesion of data chunks. For example, assume the first data chunk of matching  $P2$  having the maximal cohesion,  $\times$  operator will get data chunks from queues tagged with  $P2$  and  $P4$ . When the operator consumes data chunks, it first performs a buffer pool lookup, and, on a miss, it fetches the data chunks from disk. Buffer pool management techniques only control the access policy for data chunks; they cannot instruct operators to dynamically alter their execution orders. Then the operator will process the data chunks from queues tagged with  $P2$  and  $P4$  respectively. The result will be sent to corresponding queues tagged with  $P4 \times P2$  and  $P5 \times P2$ . The operator will get one input from  $P4'$  or  $P5'$  queue by comparing their cohesion. The output will be pipelined to the queue

**Algorithm 2** PipeliningQueries**Input:** *operator*

```

1: while (dblock=getdatablock(currentpattern))!=NULL do
2:   p=getNextJoinPattern(dblock, queryblock);
3:   while (s=getdatablock(p))!=NULL do
4:     continue;
5:   end while
6:   dblock=operator(dblock, s);
7:   processedpatterns = p ∪ currentpattern;
8:   if processedpatterns contain all patterns then
9:     output dblock;remove dblock;
10:  else
11:    q=getQueue(processedpatterns);
12:    if q==NULL then
13:      create queue q for processedpatterns;
14:    end if
15:    enqueue(q, dblock);
16:  end if
17: end while

```

**Fig. 3.** Dynamic Pipeline Executing

tagged with  $P4' \times P5'$ . Thus, we choose the data chunks having the minimal cohesion to reduce the triples matching other pattern. This helps reduce the cost to execute join operations. Dynamic chunk pipelining combining flexible chunk-at-a-time operations and the pipelined execution strategy can further improve the performance of TripleBit, a high-performance RDF store, and at the same time reduce the memory consumption of query processing.

## 5. Other Performance Optimization Techniques

To perform join operations, relations involved in a join must be shipped to the CPU for join computation. If the relations contain large number of triples, this

will incur huge communication overheads. Thus, communication is the dominant cost when a RDF engine processes big datasets. Since the join cannot be eliminated, one optimization we can apply is to limit communication. One tactic for limiting communication is to drop as many bindings as possible. For example, all restrictions, such as boundaries of variables (see Section 5.1), should be applied early. The second tactic is to use semi-joins (Bernstein and Chiu; 1981; Stocker et al.; 2001). Semi-join was originally proposed in order to reduce the communication costs of a distributed system (Bernstein and Chiu; 1981). When executing queries on large dataset, the query engine will output large intermediate results. It will induce much communication cost and computation cost. By applying semi-join, we can reduce the size of intermediate results early and consequently reduce the cost of other operations.

Join, which combines tuples from two or more relations, is an important and yet expensive operation. Efficient join algorithms will significantly improve the performance of processing queries. Sort-merge join and hash join are two common join algorithms. Although there exists debate over their performance, as computer architecture evolves, sort-merge join outperforms hash joins (Kim et al.; 2009; Neumann and Weikum; 2010a). Thus, we exploit the usage of sort-merge join in join operation and we show that the tactics can help minimize the size of intermediate results and improve the query response time. In the following, we will explain this tactics.

### 5.1. Light-weight and Fine Grained Sideways Information Passing

The non-selective sub-queries will lead to a large number of intermediate results. For example, there are large number of triples in LUBM-1B matching  $P_5$  (Appendix A). Typically, a query optimizer evaluates the query over the data in series of joins. Joins are executed firstly may be unaware of fail-matching triples from inputs until late in plan execution. Thus, a triple may propagate through a series of join operators before it is found to not produce any output. To alleviate the impact of the cases and allow more precise information of join variables, we adopt a light-weight sideways information passing mechanism (Neumann and Weikum; 2009). In sideways information passing, the query optimizer need make an a priori decision about what information to pass, how to pass it, and where to pass it. Here, the query optimizer firstly obtains the knowledge about the variables and patterns. Then, the query optimizer computes and passes the lightweight information of variables to next patterns needed in an efficient way.

In the phase to initialize patterns, the query processor considers the minimal and maximal IDs of matching triples of adjacent patterns when loading bindings of a pattern. For example, in Figure 2(a), query processor initializes  $P_2$  firstly since it has larger selectivity than  $P_4$ . When triples matching  $P_2$  are loaded, the bindings of join variable  $?y$  should have boundaries. Obviously, it is not necessary to load the triples beyond the boundaries even though they are bindings of  $P_4$ . Then ranges of  $?y$  will be passed to  $P_4$  and  $P_5$  as a filter condition to reduce the intermediate results. The query processor will filter those triples beyond the boundaries. Filtering before materializing mechanism is super efficient for star queries. By this means, the query processor reduces intermediate results when loading triples.

When query processing is carried out in parallel, many sub-results are be-

ing computed simultaneously, and several of these computations may produce information that are relevant to several triple patterns. Our information passing takes advantage of the fact that intermediate results are computed and buffered in the storage of corresponding patterns. Hence once a join is fully computed, there is state that can be correlated against arriving triples from another pattern; new tuples that do not satisfy the query conditions may be discarded early. We refer to our method as boundary SIP because of its very light-weight nature: it only passes the range of variables throughout the join edges in query plan graph. This is in contrast to previously proposed SIP strategies such as (Ives and Taylor; 2008) where filter information is passed on only when the producing operator has completed its execution and (Neumann and Weikum; 2009) where information is passed between operators during pipelined executions. In addition, our approach reduces the data volume to be accessed or joined in the query plan graph, and it is also fine-grained since the boundary is the minimal and maximal value of data chunk instead of maximal and minimal values of triples matching a pattern (See Section 4.3).

## 5.2. Semi-joins

Although SPARQL is a powerful tool for processing RDF queries, it exhibits certain performance deficiencies when the RDF datasets being queried upon become very large. We propose to employ semi-join. Semi-joins send join-attribute values to other patterns in the query plan graph, where these value lists serve as run-time filters. Suppose  $R_i$  and  $R_j$  are relations matching patterns  $P_i$  and  $P_j$  respectively. A semijoin from  $R_i$  to  $R_j$  on attribute  $a$  can be denoted as  $R_j \bowtie R_i$ . The query engine then projects  $R_i$  on attribute (or variable)  $a$  (denoted as  $\Pi_a R_i$ ), and ships this projection from storage to CPU without transmitting  $R_i$ . Comparing transmission relation  $R_i$  which contains two or more attributes, sending  $\Pi_a R_i$  which has one attribute reduces transmission cost. Furthermore, we can construct a projection index on the projection, namely, the column being indexed. Thus, the column being indexed can be removed from the original pairs and stored separately, with each entry being in the same position as its corresponding base record. With the index, the query engine can execute join between projection and  $R_j$ .

Using semi-joins, we not only drop the bindings of triple patterns using semi-joins, but also remove the useless patterns in subsequent semijoins. For example, in Figure 2(a), after the initial plan is executed, each triple pattern having only one variable ( $P1, P2, P3$ ) joins with those patterns having two or more variables respectively ( $P4, P5, P6$ ). Those bindings of the latter that are not matching the former are dropped. Hence, the constraints on join variable bindings of the former are propagated to the latter. So we can remove those patterns having only one variable in the next round of joins. An informal proof is given as follows. Without loss of generality, we consider a join query consisted of two patterns  $P_i, P_j$  where a same variable  $a_1$  appears.  $P_i$  contains only one variable, namely  $a_1$ . Variables  $a_1, \dots, a_m$  ( $m \leq 3$ ) appear in  $P_j$ . The join query can be represented as  $\Pi_{a_1, \dots, a_m} P_j \bowtie P_i = \Pi_{a_1, \dots, a_m} ((P_j \times P_i) \bowtie (P_i \times P_j))$ . Let  $P'_j = P_j \times P_i$ , and  $P'_i = P_i \times P_j$ . So,  $\Pi_{a_1, \dots, a_m} P'_i = \Pi_{a_1} P'_i$  because  $P_i$  contains only one variable. And  $\Pi_{a_1} P'_i = \Pi_{a_1} P'_j$  because a semijoin from  $R_i$  to  $R_j$  on attribute  $a$  is performed. We can know  $\Pi_{a_1, \dots, a_m} P_j \bowtie P_i = \Pi_{a_1, \dots, a_m} P'_j = \Pi_{a_1, \dots, a_m} (P_j \times P_i)$ . Thus, we can remove patterns containing only one variable after it semi-joins with other pat-

terns. Another advantage of semijoin brings is that it does not complicate sort of each relations for subsequent semijoins because each relation includes only two attributes.

Once bindings are dropped or patterns are removed, the selectivity of the nodes and edges may change and we need generate new query plan for the next round of semijoin. In each round of semijoin, some bindings of those patterns will be dropped further. Although semi-joins prune the relations, they do not generate final results. Thus, the remaining relations involved are joined in the final query plan and the final results will be produced.

By reducing intermediate results, we benefit in two ways: first, we achieve lower memory bandwidth usage and second, we accomplish the computation of joins with smaller intermediate results.

### 5.3. Sort-Merge Join

Join processing is the most expensive step during query evaluation. Our query processor uses extensively sort-merge joins because they are faster than hash or nested-loop joins (Neumann and Weikum; 2010a). This entails preserving interesting orders. For those intermediate results that are not sorted in an order suitable for a subsequent merge-join, we can transform them into suitable ordered pairs. We divide the intermediate results matching the patterns into  $k$  parts so that the first elements of pairs in each part are the same. Since pairs are SO ordered or OS ordered in storage, the second elements of pairs of each part are also sorted. Hence, we can easily transformed SO ordered pairs into OS ordered pairs (or OS ordered pairs into SO ordered pairs) using merge sort. Considering the join  $P2 \bowtie P4$ , query processor loads OS ordered pairs to initialize  $P2$  because  $P2$  contains two bounded components. For the subsequent join, we transform the bindings of  $P2$  into SO ordered pairs because  $S$  is a join variable. For example, in Figure 4, the OS pairs matching  $P2$  are first divided into data blocks (in the top of the Figure). Each of data blocks has the same  $O$  value. Thus, each data block is also ordered by  $O$  values because OS pairs are OS ordered. We can merge OS pairs of data blocks matching  $P2$  and sort them.

The transformation process is easy and cheap. In sorting  $n$  elements, the standard merge sort has an average and worst-case performance of  $O(n \log n)$ . However, our cases where each part is ordered are different from the standard case where initial elements may be un-ordered. We can deduce the performance of applying merge sort in our cases. If the running time of merge sort for a list of length  $n$  is  $T(n)$  and apply the algorithm to two sublists of the original list, then the recurrence  $T(n) = T(n_l) + T(n_r) + O(n)$  where  $n_l$  and  $n_r$  are respectively the number of leaf nodes in left child tree and right child tree. Since each part of  $k$  parts of intermediate results is ordered, thus, merge sort in our cases has an average performance of  $O(n \log k)$ . The benefit can also contribute to our RDF store storing  $(x, y)$  pairs. For example, to transform  $(x, y, z)$  triples into  $(z, y, x)$  triples is more expensive than transforming  $(x, y)$  pairs to  $(y, x)$  pairs. Compared to those stores, such as RDF-3X, which store  $(x, y, z)$  triples, the query processor require less cost to transform SO/OS ordered pairs into OS/SO ordered pairs and have more chances to execute sort-merge join. We can merge more than two data blocks at a time in order to exploit more thread level parallelism.

The query processor would use order-preserving merge-joins as long as possi-

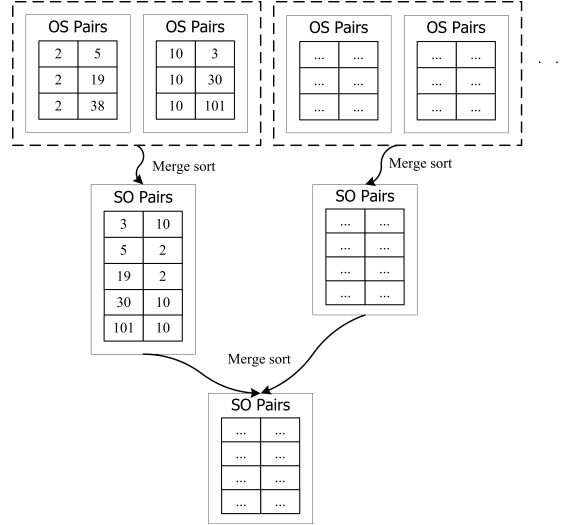


Fig. 4. Merge sort

ble. Considering  $P2 \bowtie P4$  again, the query processor 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  $P2$ , the query processor loads OS ordered pairs to initialize  $P4$  since the bindings of  $P2$  are SO ordered pairs. For star queries, the query processor can use sort-merge join for the whole join processing. If it is expensive to transform bindings of two patterns to an order suitable for later sort-merge join, the query processor will switch to hash-joins. For example, for circle queries ( $Q5$  etc.), we cannot always keep the bindings of each pattern in an order suitable for sort-merge join. Hence, we will use sort-merge join to process some joins and use hash join to process other joins in order to avoid transforming the order of bindings of a pattern frequently.

#### 5.4. Parallel Hash Join

For those intermediate results that are not ordered on the same join variables, hash join is employed. There are many existing hash join algorithms, such as no partitioning join, radix join (Balkesen et al.; 2013). Consider the characteristics of data and the better performance of merge join, we introduce a parallel hash join, which is a combination of hashed partition and merge join. We first partition the two intermediate result sets that match the two patterns into smaller disjoint partitions (referred to as sub-buffers). Concretely, the input tuples are divided up using hash partitioning (via hash function  $f$  in Figure 5) on their key values. Those pairs having the same key value will be in the same partition. Thus, all pairs matching a join pattern will be in the same partition. Triples in a partition are also sorted because triples are SO-ordered or OS-ordered in chunks. If triples are not ordered in some cases (e.g. full join), we will insert triples into suitable position of buckets in order to save the time for sorting triples when we partition triples using hash. By this means, we keep the order of triples and save the time

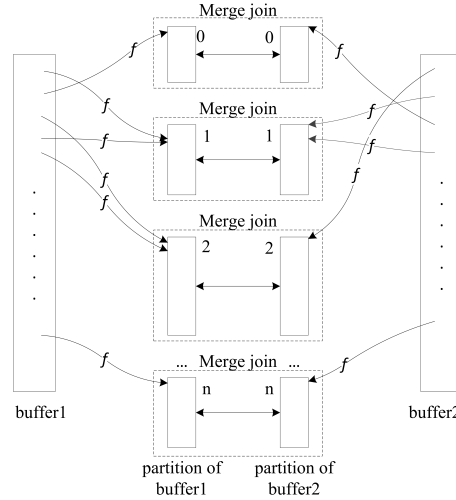


Fig. 5. Parallel Hash Join

to sort them again. Since each segment is ordered, we can perform sort-merge join on segments in a slot. Comparing with those hash functions, such as Bloom filter mapping values  $id$  onto bit-vector positions (Neumann and Weikum; 2009), the cost of our approach is modulo operation. After tuples are partitioned, the actual merge-join between the corresponding sub-buffers will be executed parallel.

## 6. Evaluation

We implemented our approaches by modifying TripleBit (Yuan et al.; 2013). In the experimental evaluation, we measured the unmodified system (TripleBit) against the enhanced system powered by the new methods presented in this paper (*our approach*). We also implement dynamic programming and join processing technologies in TripleBit according to (Neumann and Weikum; 2009, 2010a) (*DP* for brevity). For both modifications of TripleBit, we only change query execution and the rest of the system (storage, dictionary lookup, etc.) is unchanged. Here, we also include RDF-3X as a competitor.

We used three large datasets each with over a billion triples: LUBM (LUBM; 2005), UniProt (UniProt; n.d.) and Billion Triples Challenge 2012 (Semantic Web Challenge; 2012). To choose the queries which output different sizes of result set, we decide to choose most of the queries with larger result set in our experiments. This is because it generally takes query engines much time to run queries with large results. The performance to execute such queries will demonstrate the ability of techniques to process big data.

All experiments were conducted on a server with 4 Way 4-core 2.13GHz Intel Xeon CPU E7420, 64GB memory; CentOS release 5.6 (2.6.18 kernel), 64GB Disk swap space and a SAS local disk with 300GB 15000RPM. In the experiments, 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



**Table 1.** Dataset characteristics

Dataset	#Triples	#S	#O	$\#(S \cap O)$	#P
LUBM 1M	1,270,770	207,615	155,837	84,867	18
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 2.** Plan generation time on LUBM using two plan generation approaches (time in *ms*)

Queries	Approaches	1M	10M	50M	100M	500M	1B	Geom. mean
Cold caches								
Q5	DP	12.69	13.359	21.6	14.848	13.623	32.55	16.9967
	Our Approach	<b>7.123</b>	<b>8.429</b>	<b>14.685</b>	<b>10.804</b>	<b>6.818</b>	<b>19.383</b>	<b>10.3911</b>
Q6	DP	10.818	13.284	20.612	16.806	15.971	29.883	16.9552
	Our Approach	<b>7.078</b>	<b>8.416</b>	<b>14.66</b>	<b>10.785</b>	<b>6.768</b>	<b>19.351</b>	<b>10.3559</b>
Warm caches								
Q5	DP	0.147	0.135	0.135	0.134	0.147	0.162	0.1430
	Our Approach	<b>0.062</b>	<b>0.053</b>	<b>0.054</b>	<b>0.05</b>	<b>0.05</b>	<b>0.051</b>	<b>0.0532</b>
Q6	DP	0.143	0.151	0.145	0.128	0.142	0.157	0.1440
	Our Approach	<b>0.053</b>	<b>0.047</b>	<b>0.047</b>	<b>0.043</b>	<b>0.044</b>	<b>0.044</b>	<b>0.0462</b>

include the geometric mean of the query times. All results are rounded to 4 decimal places.

## 6.1. LUBM

We evaluate the scalability of our approach using several LUBM datasets of varying sizes generated using LUBM data generator (LUBM; 2005). Those LUBM datasets contain about 1M, 10M, 50M, 100M, 500M and 1 Billion triples respectively (Table 1). Therefore, we call the datasets as LUBM-1M, 10M, 50M, 100M, 500M and 1 B respectively. The sizes of LUBM 500M and 1B in their original form are 115.88GB, 231.95GB respectively. All the queries are listed in Appendix A.

First, we evaluate the plan generation time of our approach against *DP* using Q5 and Q6. According to the experimental results shown in Table 2, our approach is faster than *DP* in both cold cache cases and warm cache cases. Our approach outperforms *DP* on the cold cache time by nearly a factor of 1.63 in the geometric mean, and the warm-cache time by a factor of 2-3 in the geometric mean. One of the reasons why our approach outperforms *DP* is that *DP* relied on an optimal substructure of a problem. *DP* tries to find the cheapest valid combination of optimal solutions for subproblems that is equivalent to the query. Thus, the plan generation step itself is basically a combinatorial problem. Since the search space is huge, in order to construct a query plan fast, it should pre-compute as much data as possible to allow for fast tests for valid operator combinations. Furthermore, *DP* may construct millions of partial plans, requiring a large amount of memory.

We compared our query execution time with RDF-3X, *DP*, TripleBit in Table 3, 4 (best times are boldfaced). Here, we only report the experimental results on LUBM-500M, LUBM-1B because larger datasets tend to put more stress on RDF stores. The first observation is that our approach performs much better than RDF-3X, *DP* and TripleBit for all queries. Our approach clearly outper-

**Table 3.** LUBM 500M (time in seconds)

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom.
#Results	0	882,616	949,446	2,639,972	2,528	219,772	4,201,280	Mean
Cold caches								
RDF-3X	24.8759	137.9175	3896.3533	114.6057	1198.6520	295.0343	137.6253	257.2533
DP	25.8708	47.2692	16.8267	36.5537	31.2081	111.2313	66.0054	40.2902
TripleBit	23.9359	32.3817	9.8176	20.1255	23.4303	26.2114	34.9432	22.8841
Our Approach	<b>20.5674</b>	<b>29.4135</b>	<b>8.5311</b>	<b>18.0124</b>	<b>20.6345</b>	<b>24.3663</b>	<b>31.2583</b>	<b>20.3815</b>
Warm caches								
RDF-3X	14.6922	45.1109	2194.7415	18.8331	1114.5000	45.9599	59.6334	97.4868
DP	20.7387	26.9583	8.5854	18.5394	16.1354	34.3071	52.0677	22.0878
TripleBit	13.8401	25.4067	6.9399	16.0397	14.2093	17.7789	27.5051	16.0300
Our Approach	<b>10.6662</b>	<b>22.4087</b>	<b>5.7223</b>	<b>13.6688</b>	<b>10.5698</b>	<b>14.2858</b>	<b>25.1791</b>	<b>13.2336</b>

**Table 4.** LUBM 1B (time in seconds)

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom.
#Results	0	1,765,676	1,899,316	5,276,247	2,528	439,997	8,399,626	Mean
Cold caches								
RDF-3X	35.5183	279.6338	4784.9105	243.9383	2335.8000	496.1967	280.7406	450.6077
DP	62.0331	85.0982	28.8692	55.1597	98.4372	169.7828	153.4137	80.3134
TripleBit	30.2536	58.6571	19.1009	38.3939	36.8445	53.1482	82.4419	41.4466
Our Approach	<b>27.3576</b>	<b>51.8009</b>	<b>16.3639</b>	<b>34.1981</b>	<b>32.4415</b>	<b>47.2091</b>	<b>76.1397</b>	<b>36.8619</b>
Warm caches								
RDF-3X	30.4811	98.8727	4103.0499	54.5362	2227.5900	91.5595	126.5979	208.9890
DP	42.8596	63.2565	18.7916	42.2857	55.2341	69.5412	108.9265	51.0319
TripleBit	27.5299	50.1393	12.6735	31.6847	27.2772	36.5613	57.4734	31.6438
Our Approach	<b>21.6197</b>	<b>45.5613</b>	<b>11.2136</b>	<b>27.2417</b>	<b>20.0444</b>	<b>29.7587</b>	<b>51.6267</b>	<b>26.5365</b>

forms the other three systems for both cold and warm caches, by a typical factor in the geometric mean of 1.12-19 (cold cache) and 1.2-11 (warm cache). The query times of RDF-3X (except Q1) are far larger than other three systems. Especially, although RDF-3X and *DP* share same query processing technologies, *DP* is still much faster than RDF-3X. One reason is that the storage of RDF-3X tend to produce large intermediate results. Thus, our approach outperforms RDF-3X in the cold-cache case by a typical factor of 4 to 58, and sometimes by more than 450. In the warm-cache cases the differences are typically smaller but still substantial (factor of 1-37, sometimes by more than 380). Our approach improves *DP* and TripleBit on the cold cache time by nearly a factor of 1.1-4.5, and the warm-cache time by a factor of 1.1-2.8.

Another important factor for evaluating our approach is how the performance scales with the size of data. It is worth noting that the our approach scales linearly and smoothly when the scale of the LUBM datasets increases from 500M to 1 Billion triples.

## 6.2. UniProt

UniProt (Universal Protein Resource) aims to provide the scientific community with a freely accessible resource of protein sequence and functional information (UniProt; n.d.). UniProt is updated monthly. Here, in the experiment, we use UniProt released in Feb. 2012, which consists of 542.80GB of protein information. The RDF graph has 619,184,201 vertices and more than 2.9 billion edges (Table 1). It is a huge graph, much larger than other two billion datasets. All the queries are listed in Appendix B.

The results are shown in Table 5. Again, Our approach clearly outperforms RDF-3X, *DP* and TripleBit for all queries in both cold and warm Cache. Our approach reduces the geometric means to 11.6499s (cold) and 4.3956s (warm), which is faster than RDF-3X and *DP*. Concretely, our approach clearly outper-

**Table 5.** UniProt(time in seconds)

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom.
#Results	12,112,776	430,633	838,568	355,523	0	43,671,080	1,040,507	Mean
Cold caches								
RDF-3X	634.6494	53.0641	294.1853	105.5517	2.2879	211.5074	95.3784	90.1166
DP	572.7618	61.5337	30.3106	13.4736	1.8926	54.8142	42.2318	34.8998
TripleBit	268.1374	47.2917	9.8352	10.8213	1.5781	27.7419	24.7136	20.3799
Our Approach	<b>185.6983</b>	<b>34.3219</b>	<b>7.4023</b>	<b>6.3147</b>	<b>1.2063</b>	<b>23.6313</b>	<b>21.8481</b>	<b>15.1778</b>
Warm caches								
RDF-3X	332.3779	26.2310	39.1318	2.9810	0.1141	140.6314	23.7241	16.8594
DP	297.4920	28.0256	11.5491	2.8432	0.1098	26.1587	19.5997	10.6374
TripleBit	142.6319	24.8173	6.6047	2.2605	0.0853	16.5995	12.1463	7.0993
Our Approach	<b>108.7275</b>	<b>18.1909</b>	<b>4.5382</b>	<b>1.8704</b>	<b>0.0632</b>	<b>12.8771</b>	<b>8.6122</b>	<b>5.3013</b>

forms RDF-3X by a factor in the geometric mean of 5.9 (cold cache) and 3.2 (warm cache). Comparing with *DP*, our approach gains the performance factor of 2.3 (cold cache) and 2 (warm cache) in geometric mean. The experimental results also show the similar cases, namely that our approach improve RDF-3X and DP more when the results are larger. For example, Q1-Q4, Q6-Q7 have far larger result set than Q5. When running the queries (Q1-Q4, Q6-Q7), our approach outperforms RDF-3X by a factor of 3.4-39.7 (cold cache) and 2.8-10.9 (warm cache). However, when executing Q5, our approach outperforms RDF-3X by nearly a factor of 1.8 (cold cache and warm cache).

### 6.3. BTC 2012 Dataset

Semantic Web community holds up Billion Triples Challenge every year, which requires the participants to make use of the dataset provided by the organizers. Every year, the organizers published Billion Triples Challenge (BTC) dataset and provided the data for challenge participants. BTC 2012 dataset was crawled from Web during May/June 2012 and provided by the Semantic Web Challenge 2012 (Semantic Web Challenge; 2012). BTC 2012 dataset has varying quality due to its composition of multiple web sources. We ignored those noise data including the redundant triples which appeared many times in the dataset. This resulted in 1,048,920,108 unique triples (Table 1). BTC2012 dataset contains 57,193 distinct predicates, which are far larger than other two datasets. All queries is given in Appendix C. The query run-times are shown in Table 6. Our approach performs consistently the best for all queries. Our approach improves *DP* on the cold cache time by nearly a factor of 1.58-85, and the warm-cache time by a factor of 1.45-3. Our approach improves RDF-3X more by a factor in the geometric mean of 30.1 (cold cache) and 2.3 (warm cache).

## 7. Conclusions and future work

We have presented a dynamic and fast query processing approach, which aims to improve the performance of queries over big RDF datasets. Our approach is highlighted by dynamic plan generation and pipelining execution. We process an RDF query in two phases: plan generation phase and execution phase. Two phases are executed iteratively. Plan generation phase identifies blocks of queries and orders them according to the cost estimation. In the second phase, each block is executed using dynamic pipelining, which dynamically and adaptively select the next operator to execute in order to minimize the size of intermediate results

**Table 6.** BTC 2012 dataset (time in seconds)

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom.
#Results	7,241	48,001	13,919	3,446	0	115	583	Mean
Cold caches								
RDF-3X	46.7618	84.6836	67.4986	20.4059	2.2927	96.5408	10.8196	27.8705
DP	11.5731	13.3285	27.3465	2.6178	1.9391	18.9611	10.2275	8.8201
TripleBit	0.1592	0.5921	0.8836	1.5327	1.7638	14.6271	1.5299	1.2599
Our Approach	<b>0.1351</b>	<b>0.4469</b>	<b>0.4374</b>	<b>1.2954</b>	<b>1.2215</b>	<b>10.2368</b>	<b>1.3794</b>	<b>0.9274</b>
Warm caches								
RDF-3X	0.0961	0.6968	0.3761	1.8046	1.2235	9.5634	1.7428	0.9892
DP	0.0826	0.3833	0.3598	1.4653	0.9928	8.7576	1.5759	0.8100
TripleBit	0.0754	0.3614	0.2923	0.8385	0.8947	8.2322	0.6304	0.6088
Our Approach	<b>0.0569</b>	<b>0.2659</b>	<b>0.1761</b>	<b>0.6223</b>	<b>0.6635</b>	<b>5.3293</b>	<b>0.5325</b>	<b>0.4386</b>

generated. We also incorporate optimization techniques, such as lightweight and fine-grained sideways information passing, semi-join and other join processing optimizations to further enhance the performance of our query processing engine. We use TripleBit as a testbed and implement our approach, the dynamic programming and its join processing techniques reported in RDF-3X. We conduct experimental comparison of these approaches on three well-known RDF datasets with over a billion triples and our result shows that our approach consistently outperforms existing query engines that generate query execution plan based on dynamic programming, such as RDF-3X.

Our future work on efficient processing of large-scale RDF data continues along two dimensions. First, we are working on developing a distributed SPARQL query processing system by extending single server TripleBit. Second, we are investigating in query decomposition and RDF data partitioning techniques, including relevance oriented RDF data partitioning and query decomposition, distributed indexing algorithms.

## Acknowledgments

The research is supported by National Science Foundation of China (61073096) and National High Technology Research and Development Program of China (863 Program) under grant No.2012AA011003. Ling Liu acknowledges the partial support of her research from grants of NSF CISE NetSE program, SaTC program and I/UCRC Fundamental Research Program as well as Intel ISTC on Cloud Computing.

## References

- Abadi, D. J., Marcus, A., Madden, S. R. and Hollenbach, K. (2007). Scalable semantic web data management using vertical partitioning, *Proc. of VLDB 2007*, ACM, pp. 411–422.
- Atre, M., Chaoji, V., Zaki, M. J. and Hendler, J. A. (2010). Matrix bit loaded: A scalable lightweight join query processor for RDF data, *Proc. of WWW 2010*, ACM, pp. 41–50.
- Balkesen, C., Teubner, J., Alonso, G. and Özsu, M. T. (2013). Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware, *Proc. of ICDE'13*, IEEE Computer Society.

- Bernstein, P. A. and Chiu, D.-M. W. (1981). Using semi-joins to solve relational queries, *Journal of the Association for Computing Machinery* **28**(1): 25–40.
- Binna, R., Gassler, W., Zangerle, E., Pacher, D. and Specht, G. (2010). Spiderstore: Exploiting main memory for efficient RDF graph representation and fast querying, *Proc. of Workshop on Semantic Data Management (Sem-Data@VLDB) 2010*.
- Chebotko, A., Lu, S. and Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL translation, *Data and Knowledge Engineering* **68**(10): 973–1000.
- Harth, A., Umbrich, J., Hogan, A. and Decker, S. (2007). YARS2: A federated repository for querying graph structured data from the web, *Proc. of ISWC/ASWC2007*, Springer-Verlag Berlin, Heidelberg, pp. 211–224.
- Hartig, O., Bizer, C. and Freytag, J.-C. (2009). Executing SPARQL queries over the web of linked data, *Proc. of ISWC 2009*, Springer, Berlin, pp. 293–309.
- Hartig, O. and Heese, R. (2007). The SPARQL query graph model for query optimization, *Proc. of ESWC 2007*, Springer, Berlin, pp. 564–578.
- Huang, J., Abadi, D. J. and Ren, K. (2011). Scalable SPARQL querying of large RDF graphs, *PVLDB* **4**(11): 1123–1134.
- Ives, Z. G. and Taylor, N. E. (2008). Sideways information passing for push-style query processing, *Proc. of ICDE 2008*.
- Janik, M. and Kochut, K. (2005). BRAHMS: A workbench RDF store and high performance memory system for semantic association discovery, *Proc. of ISWC 2005*, Springer, Berlin, pp. 431–445.
- Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A. D., Blas, A. D., Lee, V. W., Satish, N. and Dubey, P. (2009). Sort vs. hash revisited: Fast join implementation on modern multicore CPUs, *PVLDB* **2**(2): 1378–1389.
- Kossmann, D. and Stocker, K. (2000). Iterative dynamic programming: a new class of query optimization algorithms, *ACM Trans. Database Syst.* **25**(1): 4382.
- LUBM (2005).  
**URL:** <http://swat.cse.lehigh.edu/projects/lubm/>
- MonetDB (2010). Overview.  
**URL:** <http://monetdb.cwi.nl/>
- Neumann, T. and Weikum, G. (2009). Scalable join processing on very large RDF graphs, *Proc. of SIGMOD 2009*, ACM, pp. 627–639.
- Neumann, T. and Weikum, G. (2010a). The RDF-3X engine for scalable management of RDF data, *The VLDB Journal* **19**(1): 91–113.
- Neumann, T. and Weikum, G. (2010b). x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases, *PVLDB* **3**(1-2): 256–263.
- Rohloff, K. and Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triplestore, *Proc. of International Workshop on Programming Support Innovations for Emerging Distributed Applications 2010 (PSI EtA '10)*, ACM.
- Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G. (1979). Access path selection in a relational database management system, *Proc. of SIGMOD'79*, Springer, Berlin, p. 2334.
- Semantic Web Challenge (2012). Semantic web challenge 2012.  
**URL:** <http://challenge.semanticweb.org/2012/>

- Stocker, K., Kossmann, D., Braumandl, R. and KemperK, A. (2001). Integrating semi-join-reducers into state of the art query processors, *Proc. of ICDE 2001*, pp. 575–584.
- Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C. and Reynolds, D. (2008). SPARQL basic graph pattern optimization using selectivity estimation, *Proc. of WWW 2008*, ACM, pp. 595–604.
- SWEO Community Project (2010). Linking open data on the semantic web.  
**URL:** <http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData>
- Udrea, O., Pugliese, A. and Subrahmanian, V. S. (2007). Grin: A graph based RDF index, *Proc. of the 22nd AAAI Conference on Artificial Intelligence*, pp. 1465–1470.
- UniProt (n.d.). UniProt RDF distribution.  
**URL:** [ftp://ftp.uniprot.org/pub/databases/uniprot/current\\_release/rdf/](ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf/)
- W3C (2008). SPARQL query language for RDF.  
**URL:** <http://www.w3.org/TR/rdf-sparql-query/>
- Weiss, C., Karras, P. and Bernstein, A. (2008). Hexastore: Sextuple indexing for semantic web data management, *PVLDB* 1(1): 1008–1019.
- Yan, X., Yu, P. S. and Han, J. (2004). Graph indexing: a frequent structure-based approach, *Proc. of SIGMOD 2004*, ACM, pp. 335–346.
- Yuan, P., Liu, P., Wu, B., Liu, L., Jin, H. and Zhang, W. (2013). TripleBit: a fast and compact system for large scale RDF data, *PVLDB* 6(7): 517–528.
- Zou, L., Mo, J., Chen, L., Özsu, M. T. and Zhao, D. (2011). gStore: Answering SPARQL queries via subgraph matching, *PVLDB* (8): 482–493.

## A. LUBM queries

PREFIX rdf: <<http://www.w3.org/1999/02/22-rdf-syntax-ns#>>

PREFIX ub: <<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>>

- Q1:** SELECT  $?x ?y ?z$  WHERE {  $?y$  ub:teacherOf  $?z$  .  $?y$  rdf:type ub:Assistant-Professor .  $?z$  rdf:type ub:Course .  $?x$  ub:takesCourse  $?z$  .  $?x$  rdf:type ub:GraduateStudent .  $?x$  ub:advisor  $?y$  . }
- Q2:** SELECT  $?x ?y ?z$  WHERE {  $?x$  rdf:type ub:AssistantProfessor .  $?y$  rdf:type ub:Department .  $?x$  ub:worksFor  $?y$  .  $?z$  rdf:type ub:UndergraduateStudent .  $?z$  ub:memberOf  $?y$  .  $?z$  ub:advisor  $?x$  . }
- Q3:** SELECT  $?x ?y$  WHERE {  $?x$  rdf:type ub:AssistantProfessor .  $?x$  ub:worksFor  $?y$  . [] ub:memberOf  $?y$  .  $?y$  rdf:type ub:Department . }
- Q4:** SELECT  $?x ?y$  WHERE {  $?x$  rdf:type ub:FullProfessor .  $?y$  rdf:type ub:UndergraduateStudent .  $?y$  ub:advisor  $?x$  .  $?x$  ub:worksFor [] . }
- Q5:** SELECT  $?x ?y ?z$  WHERE {  $?z$  ub:subOrganizationOf  $?y$  .  $?y$  rdf:type ub:University .  $?z$  rdf:type ub:Department .  $?x$  ub:memberOf  $?z$  .  $?x$  rdf:type ub:GraduateStudent .  $?x$  ub:undergraduateDegreeFrom  $?y$  . }
- Q6:** SELECT  $?x ?y ?z$  WHERE {  $?y$  ub:teacherOf  $?z$  .  $?y$  rdf:type ub:FullProfessor .  $?z$  rdf:type ub:Course .  $?x$  ub:takesCourse  $?z$  .  $?x$  rdf:type ub:UndergraduateStudent .  $?x$  ub:advisor  $?y$  . }
- Q7:** SELECT  $?x ?y$  WHERE {  $?x$  rdf:type ub:GraduateStudent .  $?x$  ub:takesCourse [] .  $?y$  rdf:type ub:AssistantProfessor .  $?x$  ub:advisor  $?y$  . }

## B. UniProt queries

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

PREFIX uni: <http://purl.uniprot.org/core/>

PREFIX taxon: <http://purl.uniprot.org/taxonomy/>

- Q1:** SELECT *?protein* *?annotation* WHERE { *?protein* uni:annotation *?annotation* . *?protein* rdf:type uni:Protein . *?protein* uni:organism [] . *?annotation* rdf:type [] . *?annotation* uni:range *?range* . }
- Q2:** SELECT *?protein* *?annotation* WHERE { *?protein* uni:annotation *?annotation* . *?protein* rdf:type uni:Protein . *?protein* uni:organism taxon:9606 . *?annotation* rdf:type *?type* . *?annotation* rdfs:comment [] . }
- Q3:** SELECT *?protein* *?annotation* WHERE { *?protein* uni:annotation *?annotation* . *?protein* rdf:type uni:Protein . *?annotation* rdf:type <http://purl.uniprot.org/core/Transmembrane\_Annotation> . }
- Q4:** SELECT *?b* *?ab* WHERE { *?b* rdf:type uni:Protein . *?a* uni:replaces *?ab* . *?ab* uni:replacedBy *?b* . }
- Q5:** SELECT *?protein* *?annotation* WHERE { *?protein* uni:annotation *?annotation* . *?protein* rdf:type uni:Protein . *?protein* uni:organism taxon:9606 . *?annotation* rdf:type <http://purl.uniprot.org/core/Disease\_Annotation> . *?protein* uni:modified "2008-07-22" . }
- Q6:** SELECT *?a* *?vo* WHERE { *?a* rdfs:seeAlso *?vo* . *?a* uni:classifiedWith <http://purl.uniprot.org/keywords/67> . *?b* uni:annotation *?annotation* . *?b* rdf:type uni:Protein . *?a* uni:replaces *?ab* . *?ab* uni:replacedBy *?b* . }
- Q7:** SELECT *?annotation* *?a* WHERE { *?annotation* rdf:type <http://purl.uniprot.org/core/Transmembrane\_Annotation> . *?annotation* uni:range *?range* . *?annotation* rdfs:comment *?text* . *?a* rdfs:seeAlso *?vo* . *?a* uni:classifiedWith <http://purl.uniprot.org/keywords/67> . *?a* uni:annotation *?annotation* . }

## 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#>

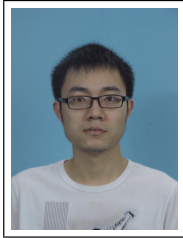
- Q1:** SELECT distinct *?a* *?b* WHERE { *?a* dbpedia:spouse *?b* . *?a* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?b* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . }
- Q2:** SELECT *?a* *?l* WHERE { *?a* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?a* dbpedia:deathPlace *?l* . *?l* pos:lat *?lat* . }
- Q3:** SELECT *?p* *?l* WHERE { *?p* dbpedia:name [] . *?p* dbpedia:deathPlace *?l* . *?p* dbpedia:spouse *?c* . *?p* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?l* pos:long *?long* . }
- Q4:** SELECT *?a* *?b* WHERE { *?a* dbpedia:spouse *?b* . *?a* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?b* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?c* owl:sameAs *?c2* . *?c2* pos:long [] . *?a* dbpedia:deathPlace *?c* . }
- Q5:** SELECT distinct *?a* *?c* *?c2* WHERE { *?a* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?a* dbpedia:placeOfBirth *?c* . *?c* owl:sameAs *?c2* . *?c2* pos:lat *?lat* . *?c2* pos:long "4.256901" . }
- Q6:** SELECT distinct *?a* *?b* *?c* WHERE { *?a* dbpedia:spouse *?b* . *?a* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> [] . *?a* dbpedia:placeOfBirth *?c* . *?b* dbpedia:placeOfBirth *?c* . *?c* owl:sameAs *?c2* . *?c* dbpedia:name *?d* . }
- Q7:** SELECT distinct *?a* *?b* *?lat* *?long* WHERE { *?a* dbpedia:spouse *?b* . *?a* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . *?b* <http://www.w3.org/

```
org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . ?a dbpedia:place-
OfBirth ?c . ?b dbpedia:placeOfBirth ?c . ?c owl:sameAs ?c2 . ?c2 pos:lat ?lat . ?c2 pos:long
?long . }
```

## Author Biographies



**Pingpeng Yuan** is an associate professor in the School of Computer Science and Technology at Huazhong University of Science and Technology. He received his Ph.D. in Computer Science from Zhejiang University. His research interests includes databases, knowledge representation and reasoning and information retrieval. Now he focus on investigating the storage and query processing technologies for big data and innovative applications. Dr. Yuan and his group have developed a number of software, including TripleBit, SemreX.



**Changfeng Xie** received his B.Sc. degree in Computer Science in 2012 from Huazhong University of Science and Technology(HUST). Now he is Master candidate of School of Computer Science and Technology, HUST. His current research interests include semantic web technology, massive data processing and distributed processing.



**Hai Jin** is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is now the Dean of the School of Computer Science and Technology at HUST. Jin received his Ph.D. in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System. Jin is a senior member of the IEEE and a member of the ACM. Jin is the member of Grid Forum Steering Group (GFSG). He has co-authored 15 books and published over 400 research papers. His research interests include computer architecture, virtualization technology, cluster computing and grid computing, peer-to-peer computing, network storage, and network security.





**Ling Liu** is a Professor in the School of Computer Science at Georgia Institute of Technology. She directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of large scale data intensive systems. Prof. Ling Liu is an internationally recognized expert in the areas of Database Systems, Distributed Computing, Internet Data Management, and Service oriented computing. Prof. Liu has published over 300 international journal and conference articles and is a recipient of the best paper award from a number of top venues, including ICDCS 2003, WWW 2004, 2005 Pat Goldberg Memorial Best Paper Award, IEEE Cloud 2012, and ICWS 2013. Prof. Liu is also a recipient of IEEE Computer Society Technical Achievement Award in 2012 and an Outstanding Doctoral Thesis Advisor award from Georgia Institute of Technology. In addition to services as general chair and PC chairs of numerous IEEE and ACM conferences in data engineering, very large databases and distributed computing fields, Prof. Liu has served on editorial board of over a dozen international journals. Currently Prof. Liu is the editor in chief of IEEE Transactions on Service Computing, and serves on the editorial board of half dozen international journals, including ACM Transactions on Internet Technology (TOIT), ACM Transactions on Web (TWEB), Distributed and Parallel Databases (Springer), Journal of Parallel and Distributed Computing (JPDC). Dr. Liu's current research is primarily sponsored by NSF, IBM, and Intel.



**Guang Yang** graduated from Huazhong University of Science and Technology with B.Sc. degree in Computer Science. Now he is a graduate student of School of Computer Science and Technology, HUST. His research focuses on query processing based on RDF data.



**Xuanhua Shi** received his Ph.D. degree in computer engineering from Huazhong University of Science and Technology (China) in 2005. From 2006, he worked as an INRIA Post-Doc in PARIS team at Rennes for one year. Currently he is an associate professor in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) at Huazhong University of Science and Technology (China). His research interests include cloud computing, data intensive computing, fault-tolerance, virtualization technology.

---

*Correspondence and offprint requests to:* Hai Jin, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. Email: [hjin@hust.edu.cn](mailto:hjin@hust.edu.cn)