RESEARCH ON OPTIMIZING TOP-K JOIN QUERIES FOR RDF DATASETS BASED ON SPARK
Jinguang Gu1,2, Hao Dong1,2, Zhao Liu1,2, Fangfang Xu1,2,*
1 College of Computer Science and Technology,
Wuhan University of Science and technology, Wuhan, 430065, China
2 Hubei Province Key Laboratory of Intelligent Information Processing and
Real-time Industrial System, Wuhan, 430065, China
xuff@wust.edu.cn

Abstract
In recent years, the scale of RDF datasets is increasing rapidly, the query research on RDF datasets in the transitional centralized environment is unable to meet the demand of data query field, especially the top-k query. Based on the Spark distributed computing system and the HBase distributed storage system, a new method for the storage of large-scale RDF data and the top-k query is designed in this paper. A top–k query plan STA (Spark Threshold Algorithm) query algorithm is proposed to reduce the connection operation of RDF data during the running process. On this basis of the STA query algorithm, the STA algorithm and a new top-k query algorithm SSJA (Spark Simple Join Algorithm) are improved to reduce the sorting related operations for the intermediate data with the help of Spark distributed system. The experimental results show that the SSJA algorithm is better than the STA algorithm in reducing the cost and has a stronger applicability.

Keywords: Spark; Top-k query; distributed; RDF data

1. INTRODUCTION

RDF (Resource Description Framework) (Decker, 2000) is a data model for data interchange on the web, more and more fields begin to use RDF to represent data for knowledge sharing. SPARQL (Simple Protocol and RDF Query Language) (Guha, 2003) is a standard RDF query language. In many cases, users are more interested in the most valuable results in the huge datasets, we call this top-k query (Ilyas, 2008). Top-k join query is a kind of top-k query which involves multiple tables or a number of datasets and the results are sorted by the aggregation score. In the actual query, ORDER BY and LIMIT phrases can easily complete the extraction of top-k results. However, they are often used as a result modifier and completed at the last stage of the query in the SPARQL algebra expression. This kind of sort mechanism is always insufficient.

In recent years, research of the query optimization on SPARQL top-k join is on the rise. Because the query process involves a large number of complex connections and sort operations, the cost of time for an accurate top-k join query result is high. With the rapid development of semantic web technology, the amount of RDF datasets continues growing which is a huge challenge to query performance. Traditional methods of top-k join query on RDF datasets are designed on single machine, which have bottlenecks in memory space and computing performance that may affect the further development and application of semantic web query technology.

Cloud computing has become one of the most popular research fields due to its high performance and easy extension of mass data storage and computing power. SPARK, developed by AMP lab of Berkeley, is a new generation of distributed processing framework for big data process. We use Spark to process the SPARQL top-k join query on large scale RDF data, which is significant to solve the bottlenecks.

SPARQL top-k query optimization is a relatively comprehensive research area. The existing methods are mainly focus on the optimization of top-k join query algorithms (Liu, 2006; Hwang, 2007; Martinenghi, 2012) and relational algebra (Pérez, 2006; Schmidt, 2010; Nikolov, 2013; Bozzon, 2012). The optimization methods based on relational algebra are mainly embodied in the expansion of SPARQL query language or SPARQL algebra and are used to meet the various needs of top-k query. Paper (Chang, 2002; Ilyas, 2004; Martinenghi, 2012; Schnatter, 2010) proposed an efficient rank-aware operator to deal with other operators in order to evaluate the top-k query on the relational database. Paper (Li, 2005) introduces the top-k query system RANKSQL which has achieved the effective top-k query on the relational database through the extended relation algebra. Optimization of top-k join algorithm is another key technology. The typical example is HRJN (Ilyas, 2004) algorithm which can only support accessing tuples sequentially and it needs additional hash table to store the input tuples and the calculation of threshold is quite complex. RSEQ (Rank Sequence Operator) algorithm (Magliacane, 2012) optimized the HRJN algorithm which uses the single ordered set to support random access and use the least sequential access to enhance the efficiency. Wagner introduced the PBRJ (Wagner, 2012) algorithm which includes boundary pattern B and tuple access strategy

DOI: 10.29268/iciot.2016.0018
P. P is used to select which set should be chose to read data and B is used to calculate the threshold.

Recently, researchers have proposed some methods to calculate top-k queries in distributed environment for different types of large scale datasets. TP TRACK (Three-Phase Uniform-Threshold algorithm) (Cao, 2004) is a method which used in the distributed environment to calculate top-k queries. TP TRACK reduces network bandwidth consumption by pruning away ineligible objects and terminates in three rounds-trips regardless of data input. Paper (Lu, 2013) introduced the rank calculation method of uncertain data for parameterized ranking functions (PRF) and put forward a method to compute the upper bound of PRF on MapReduce framework to cut the tuples which may not belong to the top-k query result. Most research work on Hadoop, but Spark has a higher performance on handling large-scale data iterations. This paper proposed a new method for top-k query on Spark which decreases the time cost by reducing the connection operation. Its experimental result demonstrates that this strategy has eminent query performance.

2. SYSTEM STRUCTURE

We use HBase to store RDF data and process them on Spark. Figure 1 shows the system architecture of the scalable top-k query on large-scale RDF data. The system receives RDF data and loads them into HBase. Firstly, SPARQL query is parsed into query tree by Jena’s SPARQL parser, to ensure syntax correctness and create the query tree. The top-k planner iterates over the query tree and greedily chooses the ‘join’ that needs to be executed, according to the selectivity and the cost of all possible joins. With the help of SPARK, the query operators are distributed executed by the top-k executor module. For those operators, the top-k executor uses the distributed M/R algorithm which launches M/R jobs to process them. Next up we will describe each module in more detail.

2.1 Storage Mechanism

Traditionally, several big clustered B+ trees are used to store different orders of six permutations of subject (S), predicate (P), and object (O): SPO, SOP, PSO, POS, OPS, and OSP. During the incremental update procedure, large number of triples will be inserted into (deleted from) the B+ trees which causes heavy maintenance burden of B+ trees. If the number of insertions (deletions) could be decreased, it could improve the performance of incremental update. Based on this consideration, three different indices orders are adopted: SP, O, OS, P. The basic idea is to separate a triple into a two-atom tuple, which is called binary tuple. Take SP, O order for example, the triple (subject (S), predicate (P), and object (O)) is separated into a binary tuple (KEY, O). The ‘KEY’ is the combination of S and O separated by the string ‘|’. In the current implement, the order of binary tuples is sorting by KEY first and then O. The establishment of the triple index for the other two orders (PO, OS, P) is similar to that for SP, O order. Therefore, the indices can be stored in HBase table in the form of key-value pairs.

![Figure 1. Architecture of the query system](image)

Here, we use an example to demonstrate SP, O order index. Assume we need to insert the following 5 triples into HBase

(0 user5, science, 79979592)
(0 user6, chinese, 6464163)
(0 user6, math, 63951451)
(0 user7, chinese, 70491811)
(0 user7, math, 11182584)

Then, the structure of indices is shown in Figure 2.

Due to our index architecture, the index scan operation is different from that in previous work. As our indices contain three different orders and index scan for the three orders is similar, we take the SP, O order as an example to illustrate the operation. There are three kinds of SPARQL query algebra triple patterns (SPAP) related to SP, O order: (s, p, ?o), (s, p, ?o), (s, p, o), and (?, p, ?o).

For SPAP like (s, ?p, ?o), we firstly look up the SP, O index using atom s. But s is not completely matched the KEY, so we filter those data whose KEY contains the string of s. The bindings for ?p and ?o can be naturally retrieved. For (s, p, ?o), we also look up the SP, O index using s|p. Compared with (s, ?p, ?o), the difference is that among all the bindings of (s, ?p, ?o), we must get the bindings that satisfy KEY = s|p. Here, we adopt an atom filter to accomplish this task. Before a new segment is read from disk, the atom filter will judge if p belongs to the range of the segment. If p does not belong to the segment, we skip reading the segment and continue to the next segment. In this way, useless segments will not be read thus eliminating redundant I/O operations. Similarly to (s, p, ?o), for (s, p, o),
the atom filter filters not only the ?p value but also the ?o value. At last, for the special case (?s, ?p, ?o), i.e., get all the triples from the RDF storage system, we get all the bindings of ?s by performing a full scan on the keys that represent SP, O order in the index.

In order to clearly describe how the atom filter and equivalent class work, strategy is given in Table 1.

Table 1. The strategy of index scan

<table>
<thead>
<tr>
<th>SQAP</th>
<th>Index Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s,p,o)</td>
<td>any</td>
</tr>
<tr>
<td>(?s,p,o)</td>
<td>PO_S</td>
</tr>
<tr>
<td>(s,?p,o)</td>
<td>OS_P</td>
</tr>
<tr>
<td>(s,?p,?)</td>
<td>SP_O</td>
</tr>
<tr>
<td>(?s,?p,o)</td>
<td>OS_P</td>
</tr>
<tr>
<td>(s,?p,?)</td>
<td>SP_O</td>
</tr>
<tr>
<td>(?s,?p,?)</td>
<td>PO_S</td>
</tr>
<tr>
<td>(?s,?p,?)</td>
<td>any</td>
</tr>
</tbody>
</table>

Table 1 shows the eight different types of SQAP, corresponding to all combinations of bindings in a triple. For each triple pattern, the table indicates the index that can be used to retrieve the corresponding data efficiently. For example, the triple pattern (?s, p, o) can be answered using the PO_S index, as it has bound predicate and object. The triple patterns having all positions bound, or all unbound (?s, ?p, ?o) and (s, p, o), can be answered by any indices. For (s, p, o), we can select the index having the smallest B+ tree depth, which is usually OS_P. For (?s, ?p, ?o) we choose an index considering any joins that must be performed on the triple pattern.

2.2 SPARQL Parser

The SPARQL query is parsed into a standard algebraic query plan by the Jena. In this paper, the top-k SPARQL query is the main research goal. In most of the algebraic representation of SPARQL, the algebraic operators that evaluate the ORDER BY and LIMIT clauses are result modifiers which alter the sequence of solution mappings after the full evaluation of the graph pattern in the WHERE clause.

For example, the following top-k SPARQL query needs to be executed:

`SPARQL 1: SELECT ?user ?name (f1(?chinesegrade) + f2(?mathgrade) + f3(?factor)) AS ?score
WHERE {
  ?user haschinesegrade ?chinesegrade. //P1
  ?user hasmathgrade ?mathgrade. //P2
  ?user hasname ?name. //P3
  ?name hasfactor ?factor //P4
}
ORDER BY DESC(?score)
LIMIT 10`

\[
\pi (\text{User}, \text{Name}, \text{Score})
\]

\[
\text{Limit}(0,10) \quad \text{Order}(?\text{Score})
\]

\[
\text{Function}(f1(?\text{chinesegrade}) + f2(?\text{mathgrade}) + f3(?\text{factor}))
\]

\[
\text{?user haschinesegrade ?chinesegrade.}
\text{?user hasmathgrade ?mathgrade.}
\text{?user hasname ?name.}
\text{?name hasfactor ?factor}
\]
SPARQL 1 retrieves the best 10 names of users ordered by a function of grades and factor: f1, f2 and f3 are normalization functions. The bold letters (P1, P2, P3, P4) represent the abbreviations of the corresponding triple patterns. The algorithm to parse the SPARQL 1 into a standard algebra expression is as follow: firstly, the WHERE clause are drawn from the storage until the whole result set is materialized, secondly, the project expression score is evaluated by the EXTEND operator on each solution and used to order the result set; finally the top 10 results are returned. The Figure 3 shows the standard algebraic query plan of SPARQL 1.

The standard algebraic query plan is equivalent to the top-k SPARQL query. So how to execute the WHERE clause optimally is important. That will be detailed in the next section.

2.3 Top-k Planner

Most research on top-k SPARQK query adopts the materialize-rank scheme. But a SPARQL engine process thousands of matching solutions, compute the score for each of them and order the result set. In addition, the ranking factor can be expensive to compute and should be evaluated only when needed. This scheme is more possible to block the performance of top-k SPARQK query. Every SPARQL query has several different query plans. Finding out an optimized query plan is critical important for improving the performance of top-k SPARQK query. Therefore, a new algorithm to generate the top-k Plan is proposed in this paper.

The following steps show the details of this algorithm:

a) Create the connection graph of the WHERE clause. If the SPARQL query includes nested WHERE clause, transfer it into equal Aggregate and Join operation first. Then create the connection graph of the WHERE clause: the node represents subject(S) or object(O), the line represents predct(P). Figure 4 shows the connection graph of the WHERE clause about SPARQL 1.

b) Label relationship of ranking factors. According to the ranking function, the relationships between ranking factors are clear at a glance. Some triple patterns in the WHERE clause can join each other directly, but others are not. Such as (?user hasmathgrade ?mathgrade) and (?name hasfactor ?factor), they don’t have common property names. When Join operation is made between (?name hasfactor ?factor ) and (?user hasname ?name), the operating result has common property name with (?user hasmathgrade ?mathgrade). The relationship of ranking factors of SPARQL 1 is shown in Figure 5.

c) Establish the executing plan of top-k SPARQK query. By using the relationship of ranking factors, some WHERE clauses with ranking factor is contained in the plan. But those triple patterns without ranking factor are left. So we must take triple patterns without ranking factor into consideration to make completely top-k SPARQK query.

Figure 5. The relationship of ranking factors of SPARQL 1

Figure 6. the executing plan of SPARQL 2

Figure 4. the connection graph of the WHERE clause about SPARQL 1
plan. Because the number of top-k executed results is less than that without top-k executing. So those triple patterns without ranking factor and without relationship of ranking factors are executed after top-k execution to decrease the expensive of Join operation. Assume that there have other triple patterns in the SPARQL 1 which have no ranking factors and without relationship of ranking factors, named it SPARQL 2. Then the executing plan of SPARQL 2 is shown in Figure 6.

In order to highlight the main query plan, Figure 6 omits the Aggregate operation and Order operation.

2.4 Top-k Execution

2.4.1 STA algorithm

Assume S is a graph pattern, var(S) denotes the set of variables occurring in S. Traditionally, S consists of triples PT and operators O.

\[ PT = \{P1, P2, P3, ..., Pn, Other1, Other2, ..., Othern\} \]

\[ O = \{Join, Union, Left Join, Difference, ...\} \]

Pn denotes the triple with ranking factors and Othern denotes the triple without ranking factors and without relationship of ranking factors. RT = (RT(i), 1 ≤ i ≤ n) represents the executive results of Pn and has been ordered by ranking factors. The formula top m results of RT is RTm = (RT(i), 0 ≤ m ≤ n), 1 ≤ i ≤ n, m is a constant. The algorithm is as follows:

1. For arbitrary triple patterns in S, execute them in the database and get RT and ROT which indicate the executive result of Pn and Othern respectively.
2. For every result set in RT, calculate the score of each item according to the related score function and get the top m items of each set as RTm. If there is a set which has less than m items, end the algorithm otherwise calculate the Tu = f(p1, p2,..., pn), where pn is the minimum score of each set in RTm.
3. Access the RTm(i) sequentially and use the tuple to take join execution with every other result set in RT and finally get the result:
   \[ \text{PRT}_m = (\text{PRT}_m(i) = (\text{RT}(i)_m \Join \text{RT}(1)) \Join \text{RT}(2) ... \Join \text{RT}(i-1) \Join \text{RT}(i+1) ... \Join \text{RT}(n)) \text{, } i \in [1-n], m \in \Bbb{N} \]
4. Calculate the score of each tuple in PRTm get the top k tuples, and get the minimum score of the k tuples as T1.
5. If T1 > Tu join the k tuples with ROT and get the final result. If T1 < Tu, repeat the second step until a result set in RT has been read completely or T1 > Tu.

The implementation of the algorithm is shown in Algorithm 1.

In the algorithm, we need to set a constant m and use the m tuples to join with other tuples, greatly reducing the Shuffle operation. However, it is not easy to set the optimal value of m because it is determined by k and the datasets. If the value of m is set too small, the number of iterations required increases, while if m is set too large, the time required for each iteration increases.

### Algorithm 1 STA algorithm

PT: \{P1, P2, P3, ..., Pn, Other1, Other2, ..., Othern\}

RO: the result of Other

K: Top-k, m: the amount of tuples each loop read

F: top-k function

Num: minimum(amount of each results in the RT)

\[
\begin{align*}
\text{If } & \text{Num}>0 \\
\text{RT}_m & \leftarrow (\text{RT}(i)_m, \text{min}(m+m-1,n-1)), i \in [1-n], \text{m=m}\text{, n}\text{,} \text{m}\text{=constant} \\
\text{Tu} & \leftarrow \text{f}(\text{min}(\text{RT}_m(i))) \\
\text{PRT}_m & \leftarrow (\text{RT}(i)_m \Join \text{RT}(j), 1 \leq j) \\
\text{UPRT}_m & \leftarrow \text{Union}(\text{PRT}_m) \\
\text{Temp}_k & \leftarrow \text{Top}_k(\text{F}(\text{UPRT}_m)) \\
\text{T1} & \leftarrow \text{the minimum score of Temp}_k \text{ Top-k R} \\
\text{If } & \text{T1} > \text{Tu} \\
\text{Out} & \leftarrow \text{Temp}_k \text{ R Join ROT} \\
\text{stop loop end if} & \text{ else} \\
\text{If } & \text{T1} < \text{Tu} \\
\text{Top-k} & \leftarrow \text{Top}_k(\text{F}(\text{Temp}_k \text{ R Union Othern})) \\
\text{T1} & \leftarrow \text{the minimum score of Top-k R} \\
\text{i} & \leftarrow \text{i+1} \text{, m = min(m+,i-1,n-1)}; \\
\text{Num} & \leftarrow \text{minimum(amount of RT(i))-m} \\
\text{end if} & \text{ end if} \\
\text{End loop} &
\end{align*}
\]

2.4.2 SSJA algorithm

In order to solve the limitation of the STA algorithm, we propose the SSJA algorithm.

1. Execute arbitrary triple patterns in S to get RT and ROT which indicate the execute result of Pn and Othern respectively.
2. Get the amount of every result set in RT and choose the minimum value. Then the appropriate result set is RT(min).
3. Access the RT(min) sequentially and use the tuple to take join execution with every other result set in RT and finally get the result:
   \[ \text{PRT}_\text{min} = (\text{RT}(\text{min}) \Join \text{RT}(1)) \Join \text{RT}(2) ... \Join \text{RT}(\text{min}-1) \Join \text{RT}(\text{min}+1) ... \Join \text{RT}(n)) \text{, } \text{min}\text{=constant} \]
4. Use the sorting function to calculate the scores and get the largest k tuples PTK.
5. Join the PTK and ROT and the final result set.

The implementation of the algorithm is shown in Algorithm 2.

The main idea of SSJA algorithm is to reduce sorting time when the query is processed. The algorithm makes best use of the computing capability of Spark platform and
makes effective use of the characteristics of the data processing. Besides, it can reduce the traversal related operations in the Spark environment as much as possible.

We don’t need to set the m may be for one iteration SSJA needs more time, but compared the whole algorithm with STA, it needs less time. It is more general and has lower complexity as well.

#### Algorithm 2: SSJA algorithm

PT: [P1, P2, P3, ..., Pr, Other1, Other1, ..., Otherm]  
K: Top-k value  
F: top-k function  
ROT: the query result of Otheri

Start

\[
\begin{align*}
\text{CRT} & \leftarrow \text{Count}(\text{ROT}) \\
\text{min} & \leftarrow \text{Min}(\text{CRT}) \\
\text{PRT} & \leftarrow (\text{RT}(\text{min}) \cup \text{RT}(j), j! = \text{min}, 0 \leq j < n) \\
\text{SPRT} & \leftarrow F(\text{PRT}) \\
\text{Top-k PR} & \leftarrow \text{Top-k}(\text{SPRT}) \\
\text{Top-k R} & \leftarrow (\text{Top-k PR} \cup \text{ROT}) \\
\end{align*}
\]

End

3. Experiments

The algorithms proposed in this paper are implemented on 10 computers with Ubuntu operating system, each computer has 8 core AMD Opteron(TM) Processor 6274 processors, 32GB main memory and 250GB hard disk. We use the jdk-1.8.0, hadoop-2.6.0, hbase-1.1.2 and spark-1.5.2.

In this paper, we use a self-developed data generator to generate a triple dataset for experiments. The dataset contains a number of user entities, each of them has a set of attributes and corresponding scores. Three datasets with different sizes which are used in the experiment is showed in table 2.

<table>
<thead>
<tr>
<th>Table 2. Datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
</tr>
<tr>
<td>number of triples</td>
</tr>
</tbody>
</table>

The plan for testing the STA and SSJA algorithms is as follows:

1. For the STA algorithm, we need to set a constant m, the m affect the time required for the same query, we do experiments on D3 with different k and m and every experiment had been done several times. The result is the average value.
2. Get the optimal m in step one and compare STA algorithm with SSJA algorithm and traditional top-k query methods.
3. Do experiments on different size of the datasets with the same query.
4. Use query statements with two query attributes and three query attributes respectively to do the experiments, compare the cost of time.

Table 3 shows the query time on the D3 dataset of STA algorithm with two sorting attributes and different k. The result shows that the query time increases with the increasing k on the same dataset. But in the case of same k, the query time of STA algorithm will increase first and then decrease with the increase of m. The reason is that the number of iterations required increase when m is small. As m increases, the time required for each iteration increases. So there is an optimal m for each query.

<table>
<thead>
<tr>
<th>Table 3. Query time with different arguments (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
</tr>
<tr>
<td>m=500</td>
</tr>
<tr>
<td>m=1000</td>
</tr>
<tr>
<td>m=5000</td>
</tr>
<tr>
<td>m=10000</td>
</tr>
</tbody>
</table>

With the result of above experiments, we get the optimal query time for different k. Then we use the optimal k to compare with the SSJA algorithm and traditional top-k query method. The result showed in the Table 4. The traditional top-k query method sorts the whole set then get the largest k results. So the value of k does not have much impact on the query time. But it’s the least efficient method. SSJA algorithm takes less time than STA algorithm even through STA algorithm has the optimal m and k has little effect of SSJA algorithm and the query time is relatively stable.

<table>
<thead>
<tr>
<th>Table 4. Query time of three algorithms (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query time</td>
</tr>
<tr>
<td>STA</td>
</tr>
<tr>
<td>SSJA</td>
</tr>
<tr>
<td>Traditional method</td>
</tr>
</tbody>
</table>

Table 5 shows the time consumed by the three algorithms under four special conditions: three different size datasets, the value of k is 100, the query statement has two query attributes and m is the optimal value. Because the size of D1 is too small, we keep two decimal places. For the three kinds of methods, the query time increases with the increase of the dataset size. But the traditional method has the highest growth rate. The SSJA always has the least query time.

<table>
<thead>
<tr>
<th>Table 5. Query time under different dataset while k=100 (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query time</td>
</tr>
<tr>
<td>STA</td>
</tr>
<tr>
<td>SSJA</td>
</tr>
<tr>
<td>Traditional method</td>
</tr>
</tbody>
</table>
Finally, we compare the three methods with different number of sorting attributes. We use the D2 and k=100, Q1, Q2, Q3 represent 2, 3, 4 sorting attributes respectively. The result is showed in the Table 5, we can see the time of STA algorithm needed grow fastest with the number of sorting attributes increase. Its query efficiency is even worse than traditional top-k query when the number of sorting attributes grow large. But the SSJA algorithm still has the minimal query time.

| Query time under different query statement while k=100 (second) |
|-------------|-------------|-------------|-------------|
| Algorithm   | Q1 | Q2 | Q3 |
| STA         | 23 | 49 | 62 |
| SSJA        | 15 | 27 | 32 |
| Traditional | 17 | 31 | 45 |

In summary, the SSJA algorithm we proposed in this paper has the best query efficiency on the cost of time and the best adaptability under the Spark distributed computing environment when it is faced with complex query conditions compared with the other two algorithms.

4. CONCLUSION

In this paper, we propose two algorithms for top-k query. The STA algorithm decreases the time by reducing the number of tuples that need for each iteration. But it is difficult to select the optimal number of tuples, the SSJA algorithm solves the problem. In order to confirm the performance of our algorithms, we design four experiments with different parameters and compare with the traditional method. The result shows SSJA algorithm has higher efficiency than STA algorithm and the traditional method.

5. ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (61272110, 61602350), the Key Projects of National Social Science Foundation of China(11&ZD189), the Natural Science Foundation of Hubei Province (2013CFFB334), the Natural Science Foundation of Educational Agency of Hubei Province (Q20101110), the State Key Lab of Software Engineering Open Foundation of Wuhan University (SKLSE2012-09-07), the Teaching Research Project of Hubei Province (2011s005), the Wuhan Key Technology Support Program(2013010602010216) and NSF of Wuhan University of science and technology Of China under grant number 2016xz2016.

6. REFERENCES


Authors

Jinguang Gu is currently a Professor in college of Computer Science and Technology of Wuhan University of Science and technology. His main research interests include Semantic WEB, New network computing, Intelligent Information Processing.

Hao Dong is a graduate student in college of Computer Science and Technology of Wuhan University of Science and technology. Her main research interest is Semantic WEB.

Zhao Liu is currently a Professor in college of Computer Science and Technology of Wuhan University of Science and technology. His main research interests include Intelligent Information Processing, computer vision.

Fangfang Xu is currently an assistant engineer in college of Computer Science and Technology of Wuhan University of Science and technology. She received her M.Sc degree from Wuhan University of Science and technology. Her main research interest is Semantic WEB.