Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Improving SPARQL query performance with algebraic expression tree based caching and entity caching*

Gang WU[1,2], Meng-dong YANG[3]
([1]*College of Information Science and Engineering*, *Northeastern University*, *Shenyang 110004*, *China*)
([2]*MOE Key Laboratory of Medical Image Computing*, *Northeastern University*, *Shenyang 110004*, *China*)
([3]*School of Computer Science and Engineering*, *Southeast University*, *Nanjing 210096*, *China*)
E-mail: wugang@ise.neu.edu.cn; mdyang@seu.edu.cn

Received Aug. 8, 2011; Revision accepted Jan. 27, 2012; Crosschecked Feb. 17, 2012

Abstract: To obtain comparable high query performance with relational databases, diverse database technologies have to be adapted to confront the complexity posed by both Resource Description Framework (RDF) data and SPARQL query. Database caching is one of such technologies that improves the performance of database with reasonable space expense based on the spatial/temporal/semantic locality principle. However, existing caching schemes exploited in RDF stores are found to be dysfunctional for complex query semantics. Although semantic caching approaches work effectively in this case, little work has been done in this area. In this paper, we try to improve SPARQL query performance with semantic caching approaches, i.e., SPARQL algebraic expression tree (AET) based caching and entity caching. Successive queries with multiple identical sub-queries and star-shaped joins can be efficiently evaluated with these two approaches. The approaches are implemented on a two-level-storage structure. The main memory stores the most frequently accessed cache items, and items swapped out are stored on the disk for future possible reuse. Evaluation results on three mainstream RDF benchmarks illustrate the effectiveness and efficiency of our approaches. Comparisons with previous research are also provided.

Key words: SPARQL, Resource Description Framework (RDF), Semantic caching, Algebraic expression tree (AET), Entity
doi:10.1631/jzus.C1101009 **Document code:** A **CLC number:** TP392

1 Introduction

Nowadays, the Semantic Web is becoming popular and has been recognized as a machine-understandable data Web where explicit semantics is specified to the information on the Web. Resource Description Framework (RDF) (Klyne and Carroll, 2004) defines the syntax for data representation and interchange on the Semantic Web. RDF triple is the smallest unit of describing a statement about a Web resource, which has the form of <subject, predicate, object> (also known as <subject, property, value>). For example, the fact that product1 with the label 'label1' is produced by producer1 could be expressed as two triples <product1, label, 'label1'> and <product1, producer, producer1>, where product1 and producer1 are uniform resource identifiers (URIs) that identify unique entities, and 'label1' is the literal representing the label. An RDF document containing multiple triples can be viewed as a directed labeled graph by simply taking a triple as a graph fragment that has an edge labeled with predicate and is directed from a subject node to an object node.

As RDF has a graph-based data model, it is more suitable for knowledge representation than the relational model (Wikipedia, 2012). However, to define queries over such a data model, a new query language

* Project supported by the National Natural Science Foundation of China (Nos. 60903010, 61025007, and 60933001), the National Basic Research Program (973) of China (No. 2011CB302206), the Natural Science Foundation of Jiangsu Province, China (No. BK2009268), the Fundamental Research Funds for the Central Universities (No. N110404013), and the Key Laboratory of Advanced Information Science and Network Technology of Beijing (No. XDXX1011)

should be proposed. W3C's SPARQL (Prud'hommeaux and Seaborne, 2008) is one of the RDF query languages. It takes graph pattern matching as the basic building block. To query products and their corresponding product labels and producers, one can write a SPARQL query by specifying a graph pattern as follows:

Query 1: Select the products and their labels and producers

```
SELECT   ?product      ?p        ?l
WHERE{  ?product   producer   ?p.
        ?product     label     ?l }
```

Here, a graph pattern consists of one or more triple patterns, which are delimited by period delimiter '.'. Identifiers starting with question mark '?' are variables, whose values are to be determined during query evaluation. An equal join on ?product is required to find triples with variable '?product' appearing as subject in both triple patterns <?product, producer, ?producer> and <?product, label, ?label>.

Although graph pattern matching facilitates a flexible expression of query semantics on RDF data, it makes the query evaluation of SPARQL different from that of SQL. Moreover, the existence of billions of RDF triples published on the Semantic Web also challenges the performance of SPARQL query engines. To obtain comparable high query performance on large scale RDF data, diverse typical database technologies like index structures, query optimizations, and parallelization have to be adapted. This is necessary to confront the complexity posed by both RDF data and SPARQL query. They are well studied and utilized in mainstream RDF stores, e.g., Sesame (Broekstra *et al.*, 2002), Jena (Wilkinson *et al.*, 2003; Owens *et al.*, 2008), Virtuoso (Erling and Mikhailov, 2007), RDF-3X (Neumann and Weikum, 2008), YARS2 (Harth *et al.*, 2007), and Oracle's RDF_MATCH (Chong *et al.*, 2005).

Database caching is another such kind of technique. It improves the performance of database with reasonable space expense based on the spatial/temporal locality principle. It has also been widely developed in the field of DB-based Web 2.0 applications. There are three kinds of caching granularity (Dar *et al.*, 1996), i.e., page level, tuple level, and semantic caching. Since data is usually organized in pages on the disk, designing a page level caching is intuitive but may be insensitive to the physical distribution of the data. The finest granularity level, tuples caching, can avoid the bad data clustering problem. However, it will produce more join operations. Semantic caching is believed to have better performance than the others (Dar *et al.*, 1996; Li *et al.*, 2001; Ren *et al.*, 2003). It caches semantic descriptions of previously evaluated queries and corresponding results. A new query can be efficiently answered if the whole or part of it has been previously cached. Such semantic locality does make sense for those queries with complex semantics.

Though page level and tuple (triple) level caching schemes have been naturally exploited in RDF stores together with other database technologies, they might not always be the best options. The Berlin SPARQL BenchMark (BSBM) illustrates a real-world application case and emulates "the search and navigation pattern of a consumer looking for a product" (Bizer and Schultz, 2009). A sequence of queries are issued to simulate a real-world workload. Since there are close associations between successive queries, semantic locality does exist. The intrinsic expressiveness of RDF data and SPARQL query language makes it possible to express rich semantics in Semantic Web applications, and hence to develop reasonable semantic caching approaches beyond low level page caching, individual triples caching, and specific application objects caching. Unfortuanately, little research has been done on this topic.

In this paper, we focus on improving SPARQL query performance with semantic caching approaches, i.e., SPARQL algebraic expression tree (AET) based caching and entity caching. Successive queries with multiple identical sub-queries and star-shaped joins can be efficiently evaluated with the two proposed approaches, respectively.

The main contributions of this paper include:

1. We develop an efficient and robust approach that caches intermediate query results based on the SPARQL AET. The serialization methods for identifying query and intermediate results are described to explain how the approach speeds up the evaluation of SPARQL queries with similar sub-queries. Operators including join, left outer join, union, and filter are supported.

2. We propose an entity caching approach which aggregates triples of the same entity (RDF resource) according to the ontology. This approach reduces star-shaped joins during the query evaluation phase and brings significant performance improvement.

3. We implement both semantic caching approaches on the Sesame RDF and provide the evaluation results of query performance on three benchmarks, LUBM (Guo *et al.*, 2005), SP^{2}Bench (Schmidt *et al.*, 2008), and BSBM (Bizer and Schultz, 2009), to illustrate the effectiveness and efficiency.

2 Related work

We are concerned about typical techniques of cache-conscious query processing (Ross, 2009), which focus mainly on designing efficient query processing algorithms. The approaches usually improve the performance of the query from four major aspects: spatial locality, temporal locality, prefetching, and sampling.

To improve spatial locality, data items accessed at the same time are organized in the same cache region. For example, data items can be arranged as column oriented (e.g., in column-wise databases) or with specific cache index structures. To improve temporal locality, techniques like data blocking, buffering, and cache-sized partitioning are employed. Prefetching data into a cache can hide the latency caused by cache miss. Cache sampling is used to choose appropriate algorithms for future query processing according to the statistical information obtained from the cached data items. In our work, query results are blocked in the same semantic region according to the semantic description of SPARQL AETs in our SPARQL AET caching. Our entity caching approach performs prefetching and data blocking for those RDF triples related to the same entity (RDF resource).

Semantic caching was first proposed by Dar *et al.* (1996). The approach was designed for a client-server database, including a client-side caching and replacement. The main idea of semantic caching is to maintain a semantic description of previously evaluated results in the cache for future reuse. Semantic caching is believed to outperform page caching and tuple caching approaches (Li *et al.*, 2001). Li *et al.* (2001) proposed two semantic caching strategies with different granularities to tackle the long latency problem caused by 'view-like' query behavior in the mediator-based database systems. Ren *et al.* (2003) provided a formal definition of the semantic caching model. They presented a query plan tree data structure to provide a detailed plan of a newly coming query from a cache. Simulation results showed advantages in "client-server environment, mobile computing, heterogeneous systems, Web applications, etc.".

Besides the relational data model, semantic caching has been studied in eXtensible Markup Language (XML) context as well. XCache is a semantic caching system for XQuery processing (Chen *et al.*, 2002).

Few of the existing RDF stores concentrate on RDF data caching design. As for Semantic Web applications, there are only a few researches on very similar topics. Abadi *et al.* (2007) proposed an approach to materialize path expressions like <?book, author, ?someone>, <?someone, wasBorn, '1860'> to a form like ?book.author:was Born='1860'. Oracle's RDF_MATCH system (Chong *et al.*, 2005) implements generic materialized join views to avoid self-join of a triple table. RDF_MATCH also applies subject-property matrix materialized join views to minimize the query processing overheads that are inherent in the canonical triple based RDF representation. Castillo *et al.* (2010) implemented RDF MatView which materializes frequent join patterns based on the analysis of SPARQL queries. Although semantic caching is an effective means to deal with view-like query behavior, it is beyond the materialized join views after all because it requires cache replacement strategies be cooperated.

To the best of our knowledge, the most relevant work in the Semantic Web is Martin *et al.* (2010), which introduced a proxy cache layer between a Web application and an RDF repository, and employed both triple level query result caching and application object caching. The caching scheme showed good results on BSBM (Bizer and Schultz, 2009). Considering the intrinsic rich semantics resided in Semantic Web applications, it is possible to design more meaningful semantic level caching approaches beyond low level individual triples caching and specific application objects caching.

3 Preliminaries

3.1 Definitions

RDF data is modeled as a directed graph with labeled nodes and edges, in which the basic element is the RDF triple. Definition 1 gives the formal definition.

Definition 1 (RDF triple) Assume U is the set of URIs, B the set of blank nodes, and L the set of literals. Triple

$$t=<s, p, o>\in(U\cup B)\times U\times(U\cup B\cup L)$$

is called an RDF triple. Here, subject s represents an entity (resource), while predicate p specifies a property of the entity and object o the property value. All subjects and objects constitute the labeled nodes of an RDF graph and predicate the labeled edges.

SPARQL is a pattern matching based RDF query language recommended by W3C. Definitions of the triple pattern and the basic graph pattern (BGP) are provided as follows.

Definition 2 (Triple pattern) Assume V is an infinite set of variables disjoint from U, B, and L. Then a triple

$$\text{tp}=<s, p, o>\in(U\cup B\cup V)\times(U\cup V)\times(U\cup B\cup L\cup V)$$

is a triple pattern.

Definition 3 (Basic graph pattern, BGP) Assume p_1, p_2, …, p_n ($n\geq1$) are all triple patterns. Then a set $G=\{p_1, p_2, …, p_n\}$ is a BGP.

In a BGP SPARQL query, one or more triple patterns are given to specify the graph pattern. During the query evaluation phase, a query engine searches the RDF graph for sub-graphs that match the graph pattern and returns them as query results. Besides triple selection, BGP also issues projection and join operations.

Definition 4 (Projection operation) Given a set of binding names, N, then the projection operation of a binding B with binding names set N on binding names N_p is

$$p(B, N_p)=\{v|v\in N\cap N_p\}.$$

Definition 5 (Join operation) Assume p_a and p_b are two triple patterns in one graph pattern. Then a join operation $j(p_a, p_b, v)=p_a\bowtie_{v=v}p_b$ occurs if and only if a variable v appears in both p_a and p_b (as any of s, p, and o).

The join operation connects multiple triple patterns and divides the evaluation of the graph pattern into multiple triple pattern evaluations. Besides the join operation (also known as equi-join), the left outer join is also supported in SPARQL, as defined in Definition 6.

Definition 6 (Left outer join operation) Assume p_a and p_b are two triple patterns in one graph pattern. Then a left outer join operation $\text{loj}(p_a, p_b, v)=p_a⟕_{v=v}p_b$ occurs if and only if a variable v satisfies either of the two conditions: (1) v appears in both p_a and p_b (as any of s, p, and o)—in this condition $\text{loj}(p_a, p_b, v)$ has the same output as $j(p_a, p_b, v)$; (2) v appears only in p_a—in this condition the variables in p_b only are all null.

Because a triple pattern cannot express quantitative criteria like the range query, regular expression matching, etc., the filter operation is introduced in SPARQL as well.

Definition 7 (Filter operation) A filter operation $f(p_{\text{in}}, p)$ is an operation that outputs the triples in p_{in} that satisfy the predicate criteria specified in p.

Definition 8 (Union operation) A union operation $u(p_a, p_b)$ is the union of the output of two input patterns p_a and p_b.

Definition 9 (Algebraic expression, AE) Given a SPARQL query Q, then expression $E=(P, O)$ is an algebraic expression (AE) of evaluating Q, where P is the set of triple patterns and O is the set of operations to be executed during query evaluation. In this paper, $O=\{j, \text{loj}, f, p\}$.

For instance, Query 1 has the corresponding AE: $p(j(<$?product, producer, ?$p>$, $<$?product, label, ?$p>$), {?product, ?p, ?l}).

In some SPARQL query engines, like Sesame, AE of a query is usually organized in a tree shaped structure to be efficiently evaluated. Hence, we call such a tree structure corresponding to the AE an AET. The AET of Query 1 is shown in Fig. 1.

As a projection operation is always performed automatically after a node has been evaluated, projections are omitted in AET. In this way, an AET can represent the evaluation process of a SPARQL query. In this study, we consider AETs of BGP SPARQL queries with left outer join, union, and filter operations.

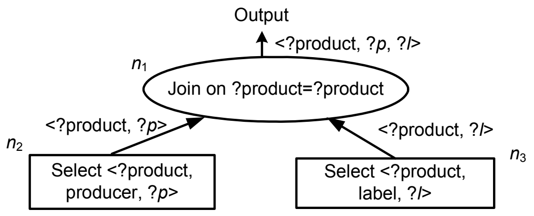
Output
<?product, ?p, ?l>
n1
Join on ?product=?product
<?product, ?p>
<?product, ?l>
n2
n3
Select <?product, producer, ?p>
Select <?product, label, ?l>

Fig. 1 The algebraic expression tree (AET) of Query 1
Leaf nodes (identified by rectangle) in an AET are always triple selection operations, where a triple selection is performed with the specified triple pattern. Non-leaf nodes (identified by ellipse) are other operations including join, left outer join, union, and filter

3.2 SPARQL query evaluation

The evaluation of a SPARQL query is similar to that of an SQL selection-projection-join query. To evaluate Query 1, an AET as shown in Fig. 1 may be generated at the query evaluation phase by query engines. For the convenience of discussion, we present only the SPARQL query evaluation process that is concerned with the join operation. Other operations involved in the evaluation, like left outer join, union, and filter, can be analyzed and applied similarly.

The join order may vary due to the optimization strategy selected and statistical information provided. For complex BGP queries, there could be much more triple patterns specified, which brings more join operations and makes optimal join ordering a challenge. The tree in Fig. 1 is only an example that helps describe our caching approaches. Those complex query optimization techniques beyond the scope of BGP are not discussed in this paper. Once a SPARQL query is translated into an AET, the query engine will perform a post-order traverse in the tree to ensure that all the sub-AET trees of a node have been evaluated before evaluating the node itself. Apparently, the output from the root node is the evaluation result of the query. Such an evaluation process is usually implemented as a recursion expansion and will not stop until the engine reaches leaf nodes. The recursion expansion is a top-down process, while recursion reduction is a bottom-up one.

As SPARQL queries can be very complex, the recursive evaluation of AET may be very complex and involves a lot of join nodes in its AET. Such queries are quite common in real-world Semantic Web applications. As join operations usually have heavy computation loads and consume a large amount of memory, they are the temporal/spatial bottleneck of the performance of RDF data management systems, especially for those dealing with large-scale RDF data.

4 AET based caching

BSBM demonstrates a scenario which consists of a sequence of queries mixed to form a typical query use case. Usually, in one query mix, successive queries probably share the partial/whole semantics. If previously evaluated query results are maintained in the cache, it is possible to speed up the query evaluation of future semantically related queries. Based on this idea, we present an AET query plan based caching approach that caches intermediate query results for possible reuse in the future SPARQL queries.

To illustrate our approach, we provide a simple query mix that consists of two successively issued SPARQL queries, both derived from Query 1.

```
Query 2
   SELECT ?product   ?l        ?p       ?c
   WHERE{ ?product   label              ?l.
          ?product   producer           ?p.
          ?product   comment            ?c }
Query 3
   SELECT ?product   ?l        ?p       ?prop
   WHERE{ ?product   label              ?l.
          ?product   producer           ?p.
          ?product   productProperty ?prop }
```

By comparison with Query 2, Query 3 has obviously redundant evaluations on triple queries <?product, producer, ?*p*> and <?product, label, ?*l*>, and the join <?product, producer, ?*p*>$\bowtie_{?\text{product}=?\text{product}}$ <?product, label, ?*l*>.

It is more evident when we translate the queries into AETs. Suppose the corresponding AETs of the two queries are as shown in Fig. 2. Apparently, they have a common sub-AET as annotated by the dashed line. The evaluation result obtained from the sub-AET is an intermediate query result. If the sub-AET can be detected and its result cached, redundancy can be avoided by reusing the intermediate query result.

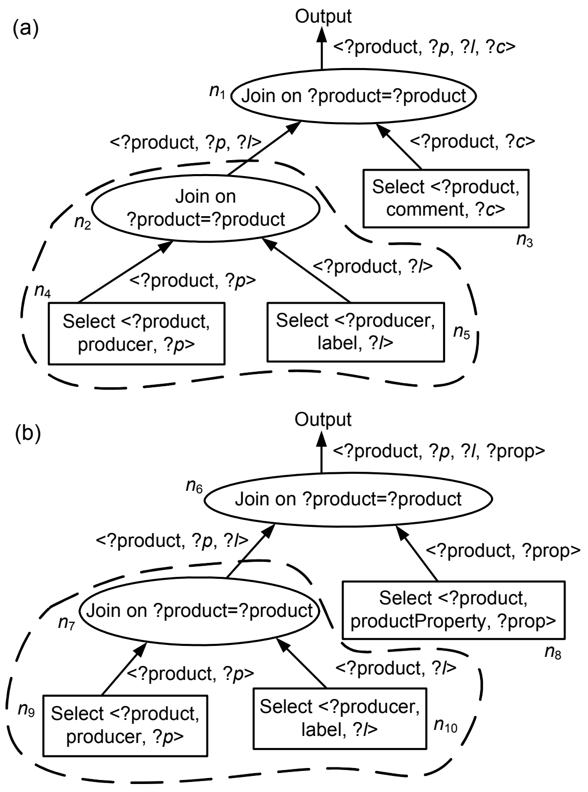
(a)
Output
<?product, ?p, ?l, ?c>
n1
Join on ?product=?product
<?product, ?p, ?l>
<?product, ?c>
n2
Join on ?product=?product
Select <?product, comment, ?c>
n3
<?product, ?p>
<?product, ?l>
n4
Select <?product, producer, ?p>
Select <?producer, label, ?l>
n5
(b)
Output
<?product, ?p, ?l, ?prop>
n6
Join on ?product=?product
<?product, ?p, ?l>
<?product, ?prop>
n7
Join on ?product=?product
Select <?product, productProperty, ?prop>
n8
<?product, ?p>
<?product, ?l>
n9
Select <?product, producer, ?p>
Select <?producer, label, ?l>
n10

Fig. 2 Algebraic expression trees (AETs) of Query 2 (a) and Query 3 (b)
These two queries have a common sub-AET as annotated by the dashed lines

Deduced from the example, more complex queries will lead to more complex AETs, and likely more sub-AET evaluation results will be cached. The cached sub-AET results can be reused if identical sub-AETs occur in the newly arriving query evaluation. Once the cache is hit, the result of matched sub-AET can be directly accessed, and hence repeated issuing of join can be avoided. In fact, queries having common sub-AETs in their corresponding AETs are very common in Semantic Web applications. Intuitively, caching the result of common sub-AETs can speed up the evaluation process, thus reducing the cost of running time and space overhead of the application.

4.1 Identifying AETs

There are multiple logically equivalent algebraic expressions for a given query. On one hand, this fact makes query optimization feasible in a query engine; on the other hand, it makes the AET based caching approach confront the challenge of identifying an AET (or part of AET) so that the query engine can recognize the identical AET (part) and reuse corresponding cached results. To solve this problem, we first define the equality of two AETs:

Definition 10 (Equality of two AETs) Assume R_A and R_B are the root nodes of two AETs T_A and T_B, respectively. Then we have

$$T_A=T_B \Leftrightarrow \begin{cases} (R_A.\text{subj}=R_B.\text{subj}) \wedge (R_A.\text{pred}=R_B.\text{pred}) \\ \quad \wedge (R_A.\text{obj}=R_B.\text{obj}), \\ \quad \text{when } (R_A \text{ is a leaf node}) \wedge (R_B \text{ is a leaf node}), \\ (R_A.\text{leftChild}=R_B.\text{leftChild}) \wedge (R_A.\text{rightChild} \\ \quad =R_B.\text{rightChild}) \wedge (R_A=R_B), \\ \quad \text{when } (R_A \text{ is a non-leaf node}) \\ \quad \wedge (R_B \text{ is a non-leaf node}), \\ \text{FALSE, otherwise.} \end{cases}$$

Considering the recursive definition of equality relation, we define the following recursive rule which helps to generate an identifier for each unique AET.

Definition 11 (Unique ID of AET) Given the root node R of an AET T, then T has a unique identifier:

$$\text{ID}(T)=\begin{cases} "<"\,R.\text{subj}\,","\,R.\text{pred}\,","\,R.\text{obj}\,">", \\ \qquad \text{when } R \text{ is a leaf node}, \\ R"("\text{ID}(R.\text{leftChild})","\text{ID}(R.\text{rightChild})")", \\ \qquad \text{when } R \text{ is a non-leaf node}. \end{cases}$$

According to Definition 11, two AETs can be identified regardless of the variables. However, there is still one case left to be considered. Suppose we use '?*p*' instead of '?product' to represent the variable of products in Query 2. The new query is lexically different from Query 2, though the two queries have the same semantics, and hence identical AETs. To solve the problem, AET is normalized as a preprocessing. The ID generated from the normalized AET can be used to identify AETs with the same query semantics.

The variable name normalization is implemented as a variable labeling method via a pre-order traverse of the AET. Each time the ID generator meets a triple pattern *p*, it assigns an ID id to each variable *v* in *p*. It first looks up the name *n* of *v* in a global map *m* which contains <name, ID> pairs. If a pair <*n*, id> does exist in *m*, the generator assigns the id found to *v*. Otherwise, the generator generates a new ID id′, and adds pair <*n*, id′> to *m* after assigning id′ to *v*.

In our example, the output of the normalization preprocessing is a new AET with expression <?v1, producer, ?v2>$\bowtie_{?v1=?v1}$<?v1, label, ?v3>. Therefore, the ID of such a normalized AET is *J*(<?v1, producer, ?v2>, <?v1, label, ?v3>) according to the rule defined in Definition 11. The ID generated can then be used to identify an AET with unique query semantics.

4.2 Replacement strategy and cache update

In AET-based caching a conventional version of the least recently used (LRU) replacement algorithm is employed. When a cache item is required to be inserted into the cache repository which is already full, the least frequently used cache item will be deleted from the cache repository to yield storage space for the new item.

As the construction of the AET-based cache repository is closely related to the query engine evaluation on the AET, computing an updated version of a cache item affected by triple insertion/removal is usually not more convenient than reconstructing the cache item. Since AET-based caching takes effect in a relatively small query context (the AET of the next coming query has an identical sub-AET to the previous one), taking cache update into account is costly and not worthwhile. In our implementation of the AET-based caching approach, all cache items immediately become invalid on triple repository update (triple insertion into or removal from the triple repository).

4.3 SPARQL query evaluation with AET based caching

To use the caching approach to improve the performance of SPARQL query evaluation, some modifications to the general query evaluation algorithm are needed. A cache lookup operation is inserted before evaluating a sub-AET; a cache writing operation is performed before returning the result of a sub-AET to its parent node in case of cache miss. With our caching approach, a SPARQL query engine works as in Algorithm 1.

Algorithm 1 AET evaluation with caching

```
Input: A normalized AET $T_n$
Output: Evaluation result $R$ of input
Global variable: current_node (the node whose result is being evaluated by the query engine)
Initialization: current_node←root node $N_r$ of $T_n$
Function evaluate($T$)
1  $R_T$←the root node of $T_n$
2  If $R_T$ is a leaf
3      Issue triple selection $R_T$, $R_t$←result
4      Return $R_t$
5  Else
6      Compute the ID of $T$ as id
7      If item <id, content> exists in the cache
8          Access the item, $R_c$←content
9          Return $R_c$
10     Else
11         $S_c$←{$N_1, N_2, \ldots, N_n$}, where $N_1, N_2, \ldots, N_n$ are
               direct child nodes of $R_T$
12         $S_{cr}$←{evaluate($N_1$), evaluate($N_2$), …,
               evaluate($N_n$)}={$R_1, R_2, \ldots, R_n$}
13         Execute operation on current_node with $S_{cr}$,
               $R_c$←result
14         If cache repository is full
15             Delete an item from the cache with LRU
16         Add item <id, $R_c$> to the cache
17         Return $R_c$
End Function
Algorithm: $R$←evaluate($T_n$)
```

The algorithm is recursive. The recursion expands at line 12 where the evaluation of a node depends on the results of all its children. Before the recursion, we add the cache lookup operation (lines 7–9): the query engine first computes the ID of the current sub-AET that takes current_node as its root node; then it looks up the cache repository with the ID. The evaluation result will be directly accessed and returned to the parent node if there is a matched cache item; in this way the evaluation from current_node down is avoided. Otherwise, the engine evaluates the result of the current sub-AET, and returns the result to the parent node after storing it in the cache repository (lines 10–17).

5 Entity caching

AET based caching improves the evaluation of SPARQL queries with an identical sub-AET structure.

However, for queries with the same semantics but different join ordering, AET based query caching does not work well since the corresponding AETs have different structures. Take the AETs shown in Fig. 3 as an example. Caching the output of n_1 will not be helpful for the evaluation of Query B because Query A and Query B have different join orders due to the query optimization strategies chosen.

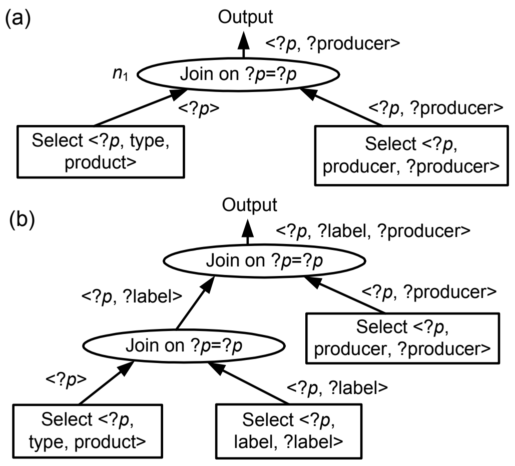

Fig. 3 Query A (a) and Query B (b) with different join orders

Moreover, selecting n properties of an entity (RDF resource) means $n-1$ joins are required in the corresponding AET. In the relational database, a join operation occurs only when there is an inter-relational condition restriction. In constant, no join operation is required when selecting properties within a relation. This inspired us to materialize those triples related to the same entity as a view to avoid join operations when retrieving the information of the entity. Fortunately, the RDF data model can simplify the materialization. We extract and aggregate triples having the same subject into a tuple containing all properties and corresponding property values of a specific entity. A table storing such tuples is shown in Table 1.

We employ a horizontal table schema (Sakr and Al-Naymat, 2010) to store such tuples for the following two reasons:

1. Flexible ontology representations are allowed with this table schema. First, for an entity, any possible property can be null. Second, an entity can have multiple types and hierarchical types.

2. An intra-entity selection can be conducted via one single access to the corresponding row in the table avoiding costly star-shaped joins.

5.1 Cache construction

We explain how cache items are constructed and filled into the cache repository on cache miss. The cache repository is initially empty. On evaluating a query, triple patterns specified in the BGP are grouped according to subjects. For example, Query 2 in BSBM retrieves basic information of a specific product by specifying a set of BGPs and groups (Table 2). For simplicity, left outer joins (optional) are omitted in this example.

Table 2 Grouped triple patterns and graph representation

ID	Triple pattern(s)
Product1	product1 rdfs:label ?label. product1 rdfs:comment ?comment. product1 bsbm:producer ?*p*. product1 bsbm:productFeature ?*f*. product1 bsbm:productPropertyTextual1 ?propertyTextual1. product1 bsbm:productPropertyTextual2 ?propertyTextual2. product1 bsbm:productPropertyTextual3 ?propertyTextual3. product1 bsbm:productPropertyNumeric1 ?propertyNumeric1. product1 bsbm:productPropertyNumeric2 ?propertyNumeric2.
?*p*	?*p* rdfs:label ?producer.
?*f*	?*f* rdfs:label ?productFeature.

Table 1 A sparse table storing aggregated RDF triples

ID	Type	Feature	Label	Producer	Description
Product1	{product}	{feature1, feature2}	{“p1”}	{producer1}	-
Feature1	{productFeature}	-	{“pf1”}	-	{“description1”}
Feature2	{productFeature}	-	{“pf2”}	-	{“description2”}
Producer1	{producer}	-	{“pp1”}	-	-

Triple patterns in which multiple properties of a specific entity are selected are considered as a group. Relations are also specified between corresponding groups. Once grouped, joins are categorized into two types, i.e., intra-entity (group) join and inter-entity (group) join. Hence, a query evaluation is divided into two steps:

Step 1: Entity selection. Each group, which corresponds to an entity, is issued independently. Entity caching is used to achieve speedup.

Step 2: Inter-entity joins, which are performed to generate the final result of the BGP.

After step 1, each group may have multiple entities (in case of a variable at the subject position) selected corresponding to the triple pattern specified in the group. Each entity corresponds to a tuple that includes the entity's URI and all its properties. The tuples are then added to the cache repository table for possible future reuse.

5.2 Query expansion and triple reducing

The method of constructing our entity cache ensures simultaneously selecting multiple properties of an entity, but an entity may be cached partially rather than totally. For instance, a product has the following properties: type, label, comment, producer, productFeature, and productProperty. Suppose a query selects type, label, comment, and producer of product1. Then there will be only one tuple <product1, type:product, label:"product1", comment: "comment1", producer:producer1> in the cache after the query is answered. Values of productFeature and productProperty are both null in the cached tuple. At this time, if there is another query selecting producer, productFeature, and productProperty of product1, accesses to both the entity cache and the triple store are inevitable. This will apparently affect the evaluation performance.

To solve this problem, we have to expand a triple pattern group to include all property values of an entity in the cache once the entity is accessed. By this expansion and prefetching, a single access to the entity cache item can fulfill entity selection no matter which properties are selected. Since the entire entity description has been reserved in the entity cache, there is no need to store those triples in the triple store as long as the cache holds the entity. Therefore, with query expansion, all triples describing the entity are removed from the triple store to reduce the store size. Furthermore, a triple write-back is performed when the entity is obsolete from the cache repository. Experimental results are provided in Section 6 to show the performance improvement of this technique.

5.3 Replacement strategy and cache update

The LRU algorithm and its variations have proved effective and efficient in cache replacement. In this work, we enhance the LRU with semantic information. In our replacement approach, if a cache item is hit, the cache item itself and all the cache items it refers to are moved to the head of the LRU list. For example, if item (product1, hasFeature:{feature1, feature2}, producer:{producer1}) is hit, then items (feature1, ...), (feature2, ...), (producer1, ...) should all be moved to the head of the LRU list. The principle behind this is the locality of data access. In practical applications, adjacent operations usually have semantics involvement (e.g., searching for products that fulfill specified restrictions, and then viewing product details of some of the search results). Thus, the data semantically related to the currently accessed data is more likely to be accessed in the near future. Compared to plain LRU, LRU with semantics has more cache hits during the warm cache phase (Fig. 4).

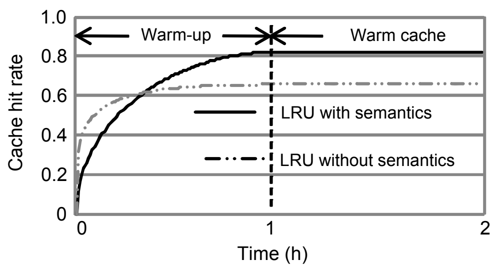

Fig. 4 Comparison of cache hit rate between LRU with semantics and plain LRU

Cache update is what to be concerned about when triple insertion/removal occurs. Our update strategy for entity caching is quite simple. Suppose the current triple to be inserted/removed is <s, p, o>. Then when entity s is in the triple store, the insertion/removal operation of <s, p, o> is performed on the triple store; when entity s is in the entity cache, the insertion/removal operation of property p:o is performed on the cache repository.

6 Implementation and evaluation

We use Sesame (Broekstra *et al.*, 2002), a Java-based RDF framework, to implement our caching approaches.

Fig. 5 shows the system architecture. A two-level storage is employed in both AET based caching and entity caching. The most frequently accessed cache items are stored in the main memory, and the swapped out items are stored on the disk for possible future use.

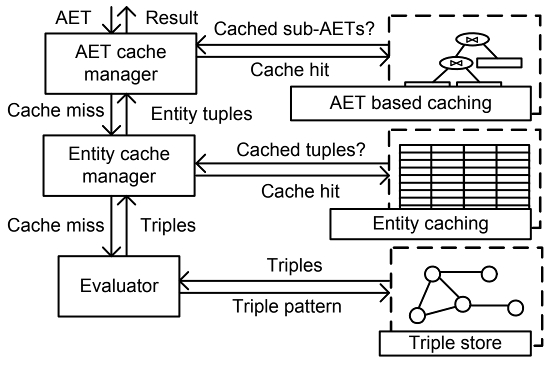

Fig. 5 Architecture of the caching system

SPARQL queries are translated into AETs and sent to the AET cache manager after parsing, and a lookup is performed in the AET cache repository. The result is directly returned to its requester on cache hit. Otherwise, the AET is passed to the entity cache manager, where AETs are processed and an entity cache lookup is performed in the entity cache repository. On cache hit, the results are directly sent to its requester, and the AET cache repository is updated. Otherwise, the AET is passed to the AET evaluator and evaluated in a conventional way. Results are returned to all requesters with both the AET cache repository and the entity cache repository updated.

AET based caching stores cache items in a custom binary format. It uses a deflation compression algorithm to reduce space consumption.

Entity caching stores tuples as key-value objects in the main memory. For tuple persistence, the MySQL (www.mysql.com) database is used. Tuples are serialized in JSON (www.json.org) and indexed with Apache Lucene (lucene.apache.org).

Three benchmarks, LUBM (Guo *et al.*, 2005), SP^{2}Bench (Schmidt *et al.*, 2008), and BSBM (Bizer and Schultz, 2009), are employed to evaluate our caching approaches. Here, we first present the metrics before introducing the evaluation results.

6.1 Evaluation metrics

1. Time improvement. We compare this metric from three aspects: (1) per-query-mix time consumption without our caching approaches; (2) per-query-mix time consumption with our caching approaches at the cache warm-up phase, when few cache hits occur; (3) per-query-mix time consumption with our caching approaches at the warm cache phase, when cache hits occur more frequently and the time consumption tends to be stable.

2. Cache hit. The count of cache hits is used to reflect to what extent the cached items are utilized.

3. Eliminated operations. This metric evaluates how many operations are eliminated.

6.2 Experiment setup

6.2.1 Datasets

LUBM: We tested our caching approaches on three datasets of different sizes: LUBM(10) (1.3M triples), LUBM(100) (13.8M triples), and LUBM (1000) (138M triples).

SP^{2}Bench: We used three SP^{2}Bench datasets of different scales: 2.5M triples, 10M triples, and 40M triples.

BSBM: We used three datasets of different sizes: BSBM(1000) (0.35M triples), BSBM(10000) (3.5M triples), and BSBM(100000) (35M triples).

6.2.2 Evaluation process

```
For $i$=1 to 10
    Construct condition $C$
    Issue query mix $Q$, which costs time period $T_i$
Next $i$
Output average value $T$ of $T_1, T_2, \ldots, T_{10}$
```

Here, condition C and query mix Q are set as follows:

1. In the evaluation of no-cache-query evaluation performance, $C=C_n$=disabling our caching approaches and using the original Sesame implementation; in the evaluation of query evaluation performance at the cache warm-up phase, $C=C_u$=enabling our caching scheme and clearing up the memory/disk cache repository; in the evaluation of query evalua-

tion performance at the warm cache phase, $C=C_W$= going on without clearing up the memory/disk cache repository after the evaluation under C_u.

2. In the evaluation of LUBM, $Q=Q_L$=14 queries provided by LUBM; in the evaluation of SP^{2}Bench, $Q=Q_S$=17 queries provided by SP^{2}Bench; in the evaluation of BSBM, $Q=Q_B$=25 queries provided by BSBM.

6.2.3 Runtime resources

The evaluation was performed on a HP dx2390 workstation with Q8400 CPU (2.66 GHz, quadruple core), 4 GB RAM, and 320 GB hard disk drive, running Windows Server 2008 R2 Enterprise Edition. JVM version is JDK 1.6.0 Update 12 for x64 architecture. Java programs are run with -Xmx1024M command argument.

6.3 Evaluation results

6.3.1 Time improvement and cache hit

Fig. 6 shows the experimental results of time improvement of our caching approaches on LUBM, SP^{2}Bench, and BSBM datasets.

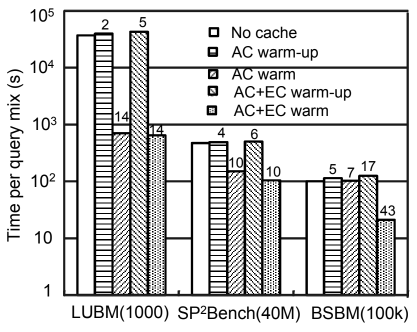

Fig. 6 Benchmark test results
The numbers above the columns are the average cache hit counts in each corresponding experiment configuration

Query evaluation at the cache warm-up phase costs more time with than without our caching approaches. This can be explained as follows:

1. There are few cache hits during the cache warm-up phase, so most queries are evaluated in a conventional way, like in the original Sesame implementation.

2. Cache building takes extra time.

The latter makes query evaluation at cache warm-up time a bit slower than in the original Sesame implementation without a caching scheme.

AET based caching contributes more to the performance improvement on LUBM and SP^{2}Bench. The reason is that LUBM and SP^{2}Bench provide static query mixes.

However, for BSBM performance improvement, entity caching contributes more to the performance improvement. The reason for BSBM having a different behavior is that BSBM generates queries dynamically, which makes queries in different query mixes have little in common. As a result, there are few cache hits in AET based caching. The entity caching caches entities and is more flexible, and hence helps a lot in reducing the workload of BSBM query evaluation.

As for cache hit, as LUBM and SP^{2}Bench provide static query mixes that contain fixed queries, both of the two caching approaches have reasonable cache hits on both benchmarks. Nevertheless, for BSBM, things become different. AET based caching leads to a small hit count. Entity caching in the warm cache phase leads to more hits.

6.3.2 Comparisons with existing approaches

The comparisons with the caching approaches in Martin *et al.* (2010) are shown in Fig. 7. In Martin *et al.* (2010), TQC stands for the triple-based query caching technique and OC stands for application object caching. For LUBM and SP^{2}Bench, TQC caches only triples, so operations still have to be reissued on cache hit. As a result, TQC does not work well on LUBM or SP^{2}Bench. OC is a caching mechanism designed for the application layer, where adjacent queries usually have sub-queries in common. But LUBM and SP^{2}Bench focus mainly on the evaluation of RDF stores' throughput with complex semantics. As neither benchmark is designed based on real Web applications, OC does not work well on LUBM or SP^{2}Bench. For BSBM, things become a little different: TQC does not perform well, while OC improves the performance to a considerable degree (about 50%). BSBM is a benchmark simulating the context of an electronic business application. Test queries in BSBM are designed intentionally to have coherent semantics in a small query context. Therefore, OC is applicable in improving performance on BSBM.

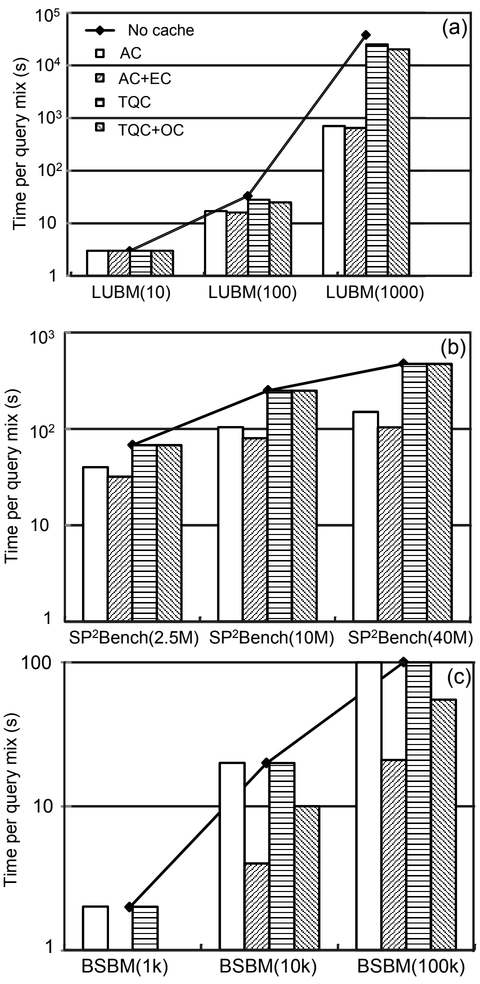

Fig. 7 Comparisons with existing approaches using datasets LUBM (a), SP^{2}Bench (b), and BSBM (c)
AC: algebraic expression tree (AET) based caching; EC: entity caching; TQC: triple-based query caching technique; OC: object caching

6.3.3 Scalability

The scalability is shown in Fig. 7 as well. The system with both of our caching approaches (AC+EC) had the best performance at the warm cache phase in all the nine datasets. Our caching approaches effectively increased the processing capacity of the Sesame RDF framework. Processing capacity on larger scale RDF datasets (LUBM(1000), SP^{2}Bench(40M), BSBM(100k)) was promoted more than on smaller scale datasets. This ensures that RDF stores with our caching approaches have higher upper bounds in data scaling.

6.3.4 Eliminated operations

Our caching approaches sped up query evaluation via elimination of joins, left outer joins, filters, and unions. According to Fig. 8, operations in LUBM and SP^{2}Bench were eliminated mainly by AET based caching because the test queries in these two benchmarks are static. For BSBM, queries are generated dynamically; thus, AET based caching did not work well and few operations were eliminated. However, entity caching provides a more flexible caching effect, and as illustrated, is able to eliminate more operations for BSBM test queries. This operation elimination ensures the speedup in BSBM query evaluation.

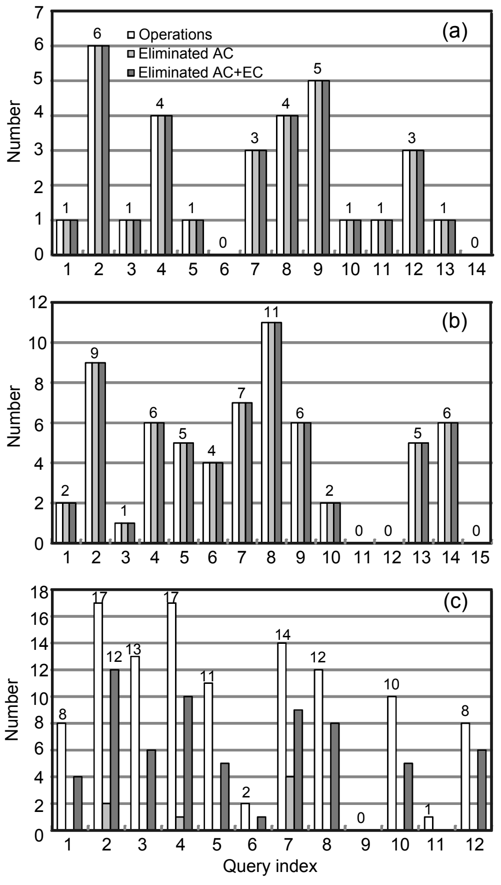

Fig. 8 Eliminated join operations for LUBM (a), SP^{2}Bench (b), and BSBM (c)

7 Conclusions and future work

In this paper, we present two semantic caching approaches that take full advantage of the semantics originated from both SPARQL queries and RDF data.

AET based caching caches intermediate results of SPARQL queries. Cache items are identified by the sub-AET structure, which ensures the identification ability of SPARQL query semantics. The AET based caching effectively improves the evaluation of SPARQL queries with an identical sub-query. It performs well on query context with similar adjacent queries.

Entity caching extracts BGP from SPARQL queries and groups triple patterns included in the BGP by the subject URIs/variables of the triple patterns. Triple pattern groups are expanded to fully select the property values of the object specified by the subject URI. The entity caching is adaptive and efficient. It works well on almost all query contexts.

Compared with existing work, both approaches and their combination can further speed up the evaluation of SPARQL query mix. The possible reason is that the proposed approaches perform semantic caching beyond low level page caching, individual triples caching, and specific application objects caching.

Our evaluations on three mainstream RDF benchmarks also reveal that AET based caching is suitable for the relatively static query mix, while entity caching plays a role in the dynamic query mix.

This work, however, is still at a very preliminary stage. Future work includes:

1. Distributed caching. Distributed caching has been widely applied in mainstream Web 2.0 applications. As the Semantic Web develops, centralized local caching will not work on very large scale Semantic Web data. Distributed semantic caching will therefore become a must in distributed Semantic Web data infrastructures.

2. Storage and indexing technique. Entity caching speeds up query evaluation via a new storage and indexing scheme of Semantic Web objects. This is applicable on the design of appropriate storage and index structure for RDF stores.

References

Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J., 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. 33rd Int. Conf. on Very Large Data Bases, p.411-422.

Bizer, C., Schultz, A., 2009. The Berlin SPARQL Benchmark. *Int. J. Semant. Web Inform. Syst.*, **5**(2):1-24. [doi:10.4018/jswis.2009040101]

Broekstra, J., Kampman, A., van Harmelen, F., 2002. Sesame: a generic architecture for storing and querying RDF and RDF schema. *LNCS*, **2342**:54-68. [doi:10.1007/3-540-48005-6_7]

Castillo, R., Leser, U., Rothe, C., 2010. RDFMatView: Indexing RDF Data for SPARQL Queries. Technical Report, Humboldt University, Berlin, Germany.

Chen, L., Rundensteiner, E.A., Wang, S., 2002. XCache: a Semantic Caching System for XML Queries. ACM SIGMOD Int. Conf. on Management of Data, p.618. [doi:10.1145/564691.564771]

Chong, E.I., Das, S., Eadon, G., Srinivasan, J., 2005. An Efficient SQL-Based RDF Querying Scheme. 31st Int. Conf. on Very Large Data Bases, p.1216-1227.

Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M., 1996. Semantic Data Caching and Replacement. 22nd Int. Conf. on Very Large Data Bases, p.330-341.

Erling, O., Mikhailov, I., 2007. RDF Support in the Virtuoso DBMS. First Conf. on Social Semantic Web, p.59-68.

Guo, Y., Pan, Z., Heflin, J., 2005. LUBM: a benchmark for OWL knowledge base systems. *Web Semant.*, **3**(2-3):158-182. [doi:10.1016/j.websem.2005.06.005]

Harth, A., Umbrich, J., Hogan, A., Decker, S., 2007. YARS2: a federated repository for querying graph structured data from the Web. *LNCS*, **4825**:211-224. [doi:10.1007/978-3-540-76298-0_16]

Klyne, G., Carroll, J.J., 2004. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation. Available from http://www.w3.org/TR/2004/REC-rdf-concepts-20040212/ [Accessed on Jan. 16, 2012].

Li, L., König-Ries, B., Pissinou, N., Makki, K., 2001. Strategies for Semantic Caching. 12th Int. Conf. on Database and Expert Systems Applications, p.284-298. [doi:10.1007/3-540-44759-8_29]

Martin, M., Unbehauen, J., Auer, S., 2010. Improving the performance of semantic Web applications with SPARQL query caching. *LNCS*, **6089**:304-318. [doi:10.1007/978-3-642-13489-0_21]

Neumann, T., Weikum, G., 2008. RDF-3X: a risc-style engine for RDF. *Proc. VLDB Endow.*, **1**(1):647-659.

Owens, A., Seaborne, A., Gibbins, N., Schraefel, M., 2008. Clustered TDB: a Clustered Triple Store for Jena. Available from http://eprints.ecs.soton.ac.uk/16974/1/www2009fixedref.pdf [Accessed on Jan. 16, 2012].

Prud'hommeaux, E., Seaborne, A., 2008. SPARQL Query Language for RDF. W3C Recommendation. Available from http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ [Accessed on Jan. 16, 2012].

Ren, Q., Dunham, M.H., Kumar, V., 2003. Semantic caching and query processing. *IEEE Trans. Knowl. Data Eng.*,

15(1):192-210. [doi:10.1109/TKDE.2003.1161590]

Ross, K.A., 2009. Cache-Conscious Query Processing. Encyclopedia of Database Systems, p.301-304. [doi:10.1007/978-0-387-39940-9_2151]

Sakr, S., Al-Naymat, G., 2010. Relational processing of RDF queries: a survey. *ACM SIGMOD Rec.*, **38**(4):23-28. [doi:10.1145/1815948.1815953]

Schmidt, M., Hornung, T., Lausen, G., Pinkel, C., 2008. SP^{2}Bench: a SPARQL Performance Benchmark. IEEE 25th Int. Conf. on Data Engineering, p.222-233. [doi:10.1109/ICDE.2009.28]

Wikipedia, 2012. Resource Description Framework. Available from http://en.wikipedia.org/wiki/Resource_Description_Framework [Accessed on Jan. 16, 2012].

Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D., 2003. Efficient RDF Storage and Retrieval in Jena2. First Int. Workshop on Semantic Web and Databases, p.131-150.

Recommended reading

Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M., 1996. Semantic Data Caching and Replacement. 22nd Int. Conf. on Very Large Data Bases, p.330-341.

Castillo, R., Leser, U., Rothe, C., 2010. RDFMatView: Indexing RDF Data for SPARQL Queries. Technical Report, Humboldt University.

Neumann, T., Weikum, G., 2008. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, **1**(1):647-659.

Martin, M., Unbehauen, J., Auer, S., 2010. Improving the performance of semantic Web applications with SPARQL query caching. *LNCS*, **6089**:304-318. [doi:10.1007/978-3-642-13489-0_21]

Broekstra, J., Kampman, A., van Harmelen, F., 2002. Sesame: a generic architecture for storing and querying RDF and RDF schema. *LNCS*, **2342**:54-68. [doi:10.1007/3-540-48005-6_7]