TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 01/14 pp613-622
Volume 15, Number 6, December 2010

Towards Efficient SPARQL Query Processing on RDF Data*

LIU Chang (刘 畅), WANG Haofen (王昊奋), YU Yong (俞 勇)**, XU Linhao (徐林昊)†

Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China;
† IBM China Research Lab, Beijing 100094, China

Abstract: Efficient support for querying large-scale resource description framework (RDF) triples plays an important role in semantic web data management. This paper presents an efficient RDF query engine to evaluate SPARQL queries, where the inverted index structure is employed for indexing the RDF triples. A set of operators on the inverted index was developed for query optimization and evaluation. Then a main-tree-shaped optimization algorithm was developed that transforms a SPARQL query graph into the optimal query plan by effectively reducing the search space to determine the optimal joining order. The optimization collects a set of RDF statistics for estimating the execution cost of the query plan. Finally the optimal query plan is evaluated using the defined operators for answering the given SPARQL query. Extensive tests were conducted on both synthetic and real datasets containing up to 100 million triples to evaluate this approach with the results showing that this approach can answer most queries within 1 s and is extremely efficient and scalable in comparison with previous best state-of-the-art RDF stores.

Key words: resource description framework (RDF) query engine; SPARQL; optimization

Introduction

With the fast growth of the semantic web, a large amount of resource description framework (RDF) data (e.g., DBpedia[1]) is being created and published for knowledge sharing and integration in web applications. Community efforts for interlinking the open RDF data sources are being actively pursued by the linking open data (LOD) project[2]. Data can be linked to produce the knowledge, such as “finding the relationship of any two people who worked with Paul Erdös”, by combining the desired RDF data sources together. Thus, the semantic web technologies open new ways to address complex information needs. However, an important problem that confronts current semantic web data management is efficient support for indexing and querying large-scale RDF data.

SPARQL[3] is the standard query language for accessing RDF data, where the basic access pattern is called the triple pattern. A triple pattern has the same form as an RDF triple, but with variables. For example, a customer may issue a SPARQL query such as shown in Fig. 1, which returns offers for a given product that fulfills the specific requirements. In the query, “?offer :vendor ?vendor” is a triple pattern. Like the counterpart of select-project-join queries in SQL, the

```
SELECT ?offer ?product
WHERE {
    ?offer ?product ?vendor.
    ?offer :vender ?vendor.
    ?offer rdf:type Offer.
    ?vendor rdf:type Vendor.
    ?vendor :country “cn”@en.
    ?product rdf:type Product.
}
```

Fig. 1 A SPARQL query

Received: 2010-09-16; revised: 2010-10-08
* Supported by the Shanghai Jiao Tong University and IBM CRL Joint Project

** To whom correspondence should be addressed.
E-mail: yyu@cs.sjtu.edu.cn; Tel: 86-21-54745879

SPARQL query supports both conjunctions and disjunctions of the triple patterns. For instance, in Fig. 1 the two triple patterns "?offer :vendor ?vendor" and "?offer rdf:type Offer" join on the variable ?offer. Furthermore, the predicates in the SPARQL query can also be variables (e.g., "?offer ?product ?vendor"), which allows "predicate-agnostic"queries.

Thus far, relational databases (RDB) have been widely used for RDF data storage with the SPARQL queries translated into SQL statements for evaluation. For example, Jena2[4] designs the property tables to cluster together subjects with the same properties; while SOR[5] uses multiple special-purpose tables, each of which keeps a type of RDF terms. However, these approaches involve many self-joins or cross-table-joins when processing queries, which greatly slows the query performance. For example, an equivalent SQL query of the SPARQL query shown in Fig. 1 includes five self-joins, as displayed in Fig. 2. To solve this problem, Abadi et al.[6] proposed a vertical partitioning based method to store and query RDF data. Though many joins are still required to answer queries over multiple predicates, linear merge joins can be used to alleviate the joining cost as each table is sorted by subject. However, predicate-unbound queries require a large number of union operations on all tables.

```
SELECT T1.sub as offer, T1.pre as product
FROM TRIPLE T1, TRIPLE T2, TRIPLE T3,
        TRIPLE T4, TRIPLE T5, TRIPLE T6
WHERE { T1.sub = T2.sub and T1.sub = T5.sub
    and T1.pre = T4.sub and T1.obj = T3.sub
    and T1.obj = T6.subj and T4.obj = 'rdf:type'
    and T2.obj = 'Offer' and T3.obj = 'Vendor'
    and T4.obj = 'Product' and T5.pre = 'vender'
    and T6.pre = 'country' and T6.obj = 'cn'
    and T2.pre = 'rdf:type' and T3.pre = 'rdf:type'}
```

Fig. 2 A translated SQL query (TRIPLE is the triple table)

Unlike the RDB-based solutions, native RDF stores, such as Hexstore[7], YARS2[8], and Semplore[9], are designed for efficient RDF query processing or semantic searches. Hexstore[7] uses a sixtuple indexing method for fast merge-joins while YARS2[8] designed the blocking index for fast triple retrieval and Zhang et al.[9] employed the IR technique to index RDF triples. However, compared with our approach, Zhang et al.[9] can only process a unary tree shaped search query, so it is not useful for SPARQL query processing. Furthermore, they do not do any query optimization.

More recently, Stocker et al.[10] borrowed the concept of the selectivity estimation from the traditional database community for optimizing RDF query processing. However, their method only takes the basic SPARQL graph patterns[3] into account and cannot improve complex SPARQL query patterns. RDF-3X[11] indexes RDF triples with a B^+-tree and answers SPARQL queries by transforming them into the equivalent relational algebra trees. As such, the traditional optimization-like algorithm can be used to identify the optimal joining order. Unlike Neumann and Weikum[11], the current study indexes the RDF triples by an inverted index structure as the IR engine organizes the same types of RDF data on the disk[9]. This greatly facilitates data retrieval and makes the implementation of the linear merge join easy and efficient. Therefore, the main object of the current work is how to efficiently optimize SPARQL query evaluations.

Query optimization is the most important research topic in the database area and has been well studied in past decades, with methods such as access path selection[12], histogram[13], and algebraic space searching[14]. However, query optimization for RDF query processing is still in its infancy. This paper presents an efficient RDF query engine for SPARQL query optimization and evaluation. A SPARQL query graph model is defined that can effectively expresses the query semantics. Then a set of operators is implemented on the indexed triples for transforming the SPARQL query graph into an optimal execution tree for evaluation. To achieve this goal, a main-tree-shaped optimization algorithm was developed to identify the optimal execution plan by effectively reducing the search space to determine the optimal joining order. The optimization uses a set of RDF statistics to estimate the execution cost of the query plan. Finally, the optimal execution plan is evaluated.The contributions of this work are as follows.

● An efficient RDF query engine is given which stores and indexes triples using inverted indices. Operators are developed for query optimization and evaluation to analyze search costs with their implementation significantly contributing to the query performance.

● A main-tree-shaped optimization algorithm was developed that transforms a SPARQL query graph into the optimal query plan by effectively reducing the search space. The optimization was a set of RDF statistics to estimate the query plan execution cost.

● Extensive tests on both synthetic and real datasets show that this approach is extremely efficient and scalable in comparison with state-of-the-art RDF stores.

1 Problem Statement

A SPARQL query graph, $G = \{V, E\}$, is composed of a vertex set, V, and an edge set, E. All vertices in V are divided into two disjoint subsets, V_n and V_t (i.e., $V = V_n \cup V_t$ and $V_n \cap V_t = \varnothing$). All vertices in V_n are named normal vertices and all vertices in V_t are called triple vertices (introduced in the next paragraph). Each normal vertex $v \in V_n$ represents a variable with one or several constraints. The constraints determine the answers bound with the variable node. A constraint could be either a concept constraint or a relationship constraint. A concept constraint is in the form of concept(:*c*) that represents the triple pattern <?*v*,rdf:type,:*c*>; while a relationship constraint could be in the form of R_s(:*p*,:*o*), R_p(:*s*,:*o*), and R_o(:*s*,:*p*) that represent the triple patterns <?*v*,:*p*,:*o*>, <:*s*,?*v*,:*o*>, and <:*s*,:*p*,?*v*> respectively. Notice that if one normal vertex or edge has multiple constraints, the relationship between these constraints is conjunctive according to the SPARQL semantic. In addition, a normal vertex $v \in V_n$ is identified as selected if it appears in the select clause of the query. Figure 3 shows four normal vertices with their constraints.

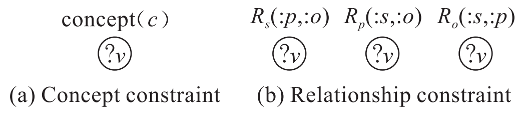

Fig. 3 Normal vertices with their constraints

All the edges in E are classified into two disjoint subsets, E_n and E_t (i.e., $E = E_n \cup E_t$ and $E_n \cap E_t = \varnothing$). All edges in E_n are named normal edges and all edges in E_t are called triple edges. Each normal edge $e \in E_n$ links two normal vertices and is associated with one or several constraints (i.e., constant values) which indicate the relationship between variables. For example, for <?*s*,?*p*,:*o*> shown in Fig. 4a, :*o* is represented as the constraint of an edge that connects two vertices ?*s* and ?*p*. Each triple edge $e \in E_t$ connects with a triple vertex $v \in V_t$ and a normal vertex $w \in V_n$. As such, three normal vertices linked by three triple edges and one triple vertex constitute a triple pattern <?*s*,?*p*,?*o*> whose subject, predicate, and object are all variables. Figure 4b shows how three triple edges and one triple vertex (dark node) connect with three normal vertexes. Note that both triple edges and triple vertices do not have any constraints.

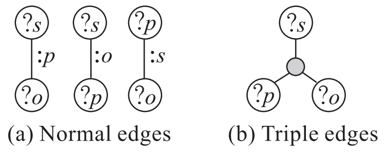

Fig. 4 Normal and triple edges

Example 1 Figure 5 shows the graphical representation of the query displayed in Fig. 1. In Fig. 5, the double-cycled nodes ?product and ?offer are selected since they appear in the SELECT clause.

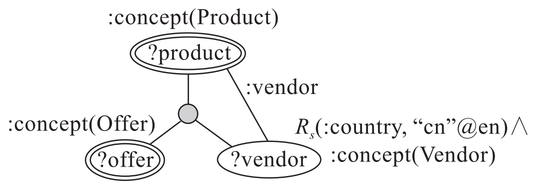

Fig. 5 A SPARQL query graph

Problem statement Given a SPARQL query graph and a large set of RDF triples, the problems are (1) how to efficiently index all triples and design operators for query optimization and evaluation, and (2) how to effectively transform the query graph into an optimal execution tree for processing.

2 Index Structure and Operators

This section first describes the index structure for indexing RDF triples and then defines the operators for query optimization and evaluation.

2.1 Index structure

Mapping RDF terms to IDs First assign a unique ID to each indexed RDF term because storing IDs occupies less disk space and comparison of IDs costs less time. Since the IR engine generates a unique docID for each document, create a document for each RDF term and then treat the docID as its ID. Specifically, first create the ID field for the inverted index; then for each RDF term, *t*, we insert *t* into the term list of the ID field and then we insert the docID into the document list of term *t*. An RDF term's ID is obtained using the term query <ID,term> to the IR engine, where term is the string value of the RDF term. Figure 6 shows the ID field optimization.

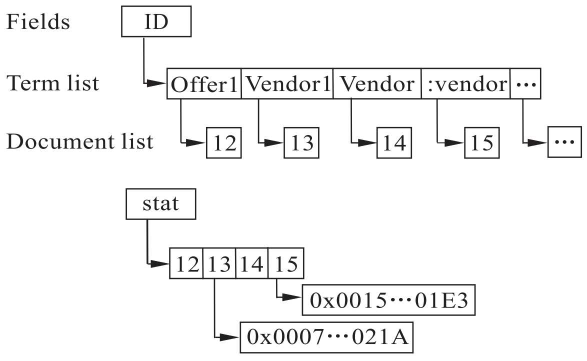

Fig. 6 ID and stat fields

Indexing statistics The system defines nine types of statistical information for each RDF term (see Section 4 for details). First encode the value of each type of statistic as a 4-byte little endian. The statistics of RDF term *t* are stored by creating a stat field for the inverted index and then inserting *t*'s ID into the term list of the stat field. Finally a 36-byte binary string is added into the document list of term *t*. The statistics of RDF term *t* are obtained using *t*'s ID with the document (i.e., the string value of the statistics) of the term then returned by a term query <stat,ID>. Figure 6 illustrates the organization of the stat field.

When new RDF triples are loaded, the statistics of all RDF terms appearing in these RDF triples should be updated. First obtain its statistics of each RDF term in the newly-loaded triples; then update the corresponding statistic and replace the old value with the new one. Similarly, when deleting RDF triples, the same method is used to update the statistics of all RDF terms appearing in the deleted RDF triples.

Indexing RDF triples Index any triple <*t*,rdf:type, *c*> whose predicate is rdf:type based on the type field. First insert *c*'s ID into the term list of the type field, then add a document whose ID is equal to *t*'s ID into the document list of term *c*. The term query <type,*c*> is used to retrieve all associated RDF terms that are the instances of the RDF concept *c*.

All triples whose predicates are not rdf:type are indexed with six fields: spo, sop, pso, pos, osp, and ops. Taking the pso field as an example, for each triple <*s*,*p*,*o*>, treat *s* as the document and *p* as the term, then put the term *p* into the term list of the pso field, and add *o* into the position list of document *s*. Then a term query <pso,*p*> can transverse all triples whose predicate is *p* in the order of *s* and *o*. The query returns the document list of *p* with each document representing a subject *s*. For each document *s*, the query then retrieves its position list and each position represents an object *o*. The other five fields are organized similarly and can be used to access all triples in different orders. Notice that, both the document list and the position list in these fields are sorted as the IR engine stores the IDs on the disk in ascending order. This allows implementation of a cheap linear merge join on the indexed triples. Table 1 lists the symbols.

Table 1 Symbols

Symbol	Description
$S=\{B_S,V_S\}$	A binding set
B_S,V_S	A binding list and a variable list
$\beta,\beta(v)$	A binding and a bound value of the variable v
t, $t\{\beta\}$	A triple pattern and an answer to the triple pattern
U	The whole triple set

2.2 Operators

Binding set is used to represent the intermediate and final results. A binding set $S=\{B_S,V_S\}$ is composed of a binding list, B_S, and a variable list, V_S. A binding $\beta\in B_S$ is a mapping from V_S to IDs that are the internal representation of the RDF terms. $\beta(v)$ is used to indicate the bound value of variable *v*. Let *U* be the whole triple set and *t* be a triple pattern, then replacement of all variables *V* in *t* with their bound values $\beta(V)$ is denoted as $t\{\beta\}$. If $t\{\beta\}\in U$ is true, then the bound value $\beta(V)$ is an answer of *t*. In addition, given two binding sets S_1 and S_2, and two bindings $\beta_1\in B_{S_1}$ and $\beta_2\in B_{S_2}$, $\beta_1(V)=\beta_2(V)$ if for each variable $v\in V$, $\beta_1(v)=\beta_2(v)$, where $V=V_{S_1}\cap V_{S_2}$. All symbols used in this section are listed in Table 1.

Index scan $\pi_t(f)$ Given a triple pattern *t* with only one variable *v* and an index field *f*, return $S=\{B_S,\{v\}\}$ where for each $\beta\in B_S, t\{\beta\}\in U$. As mentioned before, $\pi_t(f)$ can be easily implemented with the term query; thus, its cost is equal to the cost of sequentially reading the index.

Intersection $\lambda(S_1,S_2)$ Given two binding sets S_1 and S_2 whose $V_{S_1}=V_{S_2}=\{v\}$, return $S=\{B_{S_1}\cap B_{S_2},V_{S_1}\}$. As B_{S_1} and B_{S_2} are sorted, use the merge-sort algorithm to implement $\lambda(S_1,S_2)$ whose cost is $O(|S_1|+|S_2|)$.

Selection $\sigma_t(S)$ Given a binding set *S* and a triple pattern t, return a binding set $S'=\{B_{S'},V_S\}$ where for each $\beta'\in B_{S'}$, $t\{\beta\}\in U$. This system first enumerates each $\beta\in B_S$ and then checks if $t\{\beta\}\in U$, which requires an index lookup. Thus, its cost is $O(|S|+|S'|)$.

Aggregation $\mathbf{aggr}_V(S)$ Given a binding set S and a variable set $V\subset V_S$, return a binding set $S'=\{B_{S'},V\}$, where for each $\beta'\in B_{S'}$ there is $\beta\in B_S$ satisfying $\beta'(V)=\beta(V)$. This system first inserts all $\beta'(V)$ that equals to $\beta(V)$ into $B_{S'}$ for each $\beta\in B_S$; then sorts $B_{S'}$ and removes all duplicates in $B_{S'}$. Thus, its cost is $O(|S|\log|S|)$.

Binary join $\rtimes_t(S_1,S_2)$ Given two binding sets S_1 and S_2, and a triple pattern t with V the set of all variables in t, then return a binding set $S'=\{B_{S'},V_{S_1}\cup V_{S_2}\cup V\}$ satisfying the conditions: (1) $\forall\beta'\in B_{S'}$, $t\{\beta'\}\in U$, (2) $\exists\beta_1\in B_{S_1}$, $\beta_1(V_{S_1})=\beta'(V_{S_1})$, and (3) $\exists\beta_2\in B_{S_2}$, $\beta_2(V_{S_2})=\beta'(V_{S_2})$.

Triple join $\rtimes_t(S_1,S_2,S_3)$ Given three binding sets S_1, S_2, and S_3, and a triple pattern t with V the set of all variables in t, then return a binding set $S'=\{B_{S'},V_{S_1}\cup V_{S_2}\cup V_{S_3}\cup V\}$ satisfying the conditions: (1) $\forall\beta'\in B_{S'}, t\{\beta'\}\in U$, (2) $\exists\beta_1\in B_{S_1},\beta_1(V_{S_1})=\beta'(V_{S_1})$, (3) $\exists\beta_2\in B_{S_2},\beta_2(V_{S_2})=\beta'(V_{S_2})$, and (4) $\exists\beta_3\in B_{S_3}$, $\beta_3(V_{S_3})=\beta'(V_{S_3})$.

3 Query Optimization

Section 2 described how to transform a SPARQL query graph into an equivalent query tree. Efficiently evaluation of a SPARQL query graph involves finding a minimal cost execution tree for the query processing. The challenges are to effectively reduce the search space for determining the optimal joining order and to determine which type of statistics should be collected for estimating the cost of the execution tree. This section first formally defines the cost model and the statistics for the cost estimate, and then presents the optimization algorithm.

3.1 Cost model

Given a SPARQL query graph, the main goal is to find the optimal execution tree with the minimal cost. The execution cost is measured by defining a cost model of the execution tree. For simplicity, use Cost(op$_i$) to indicate the cost of the i-th operator op$_i$ in tree T, where op$_i$ is one of the operators listed in Section 3.2. Thus, the total cost of execution tree T is

$$\mathrm{Cost}(T)=\sum_{\mathrm{op}_i\in T}\mathrm{Cost}(\mathrm{op}_i).$$

Section 2.2 showed that the cost of each operator depends on the size of both the input and the output. However, the exact cardinalities of the input and output of each operator are not known beforehand. Relational databases use histograms to solve this problem[13]. The current system uses a similar idea for the cardinality estimation.

Definition 1 (Concept statistic) Let c be a concept, then its concept statistic, IND(c), is the number of distinct instances v where <v, rdf: type, c>$\in U$.

Definition 2 (Domain statistic) Let v be a subject, then its domain statistic, DOM$_s$(v), is the number of distinct predicates p where <v, p, *>$\in U$, where * indicates that the position can be replaced by any value. Let v be a predicate, then DOM$_p$(v) is the number of distinct subjects s where <s, v, *>$\in U$. Let v be an object, then DOM$_o$(v) is the number of distinct subjects s where <s, *, v>$\in U$.

Definition 3 (Range statistic) Let v be a subject, then its range statistics, RNG$_s$(v), is the number of distinct objects o where <v, *, o>$\in U$. Let v be a predicate, then RNG$_p$(v) is the number of distinct objects o where <*, v, o>$\in U$. Let v be an object, then RNG$_o$(v) is the number of distinct predicates p where <*, p, v>$\in U$.

These statistics are collected while loading the triples into the repository. While loading a triple, the type of each RDF resource is checked and its statistics are updated if it exists. Otherwise, the missing statistics are created. These statistics are used to build a set of small equal-width histograms for each type of statistical information. As new triples are added or existing triples are removed, the statistics will change (see Section 2.2). The histograms are refreshed, when the number of changes of the indexed statistics exceed a predefined threshold η.

Since all the leaf nodes are index scans, their cost can be estimated based on the input cardinality. For example, given a triple pattern <?s, rdf: type, c>, find the bucket that contains c's ID and then use the bucket frequency to estimate their cost. Similarly, for single triple patterns such as <?s, p, o>, identify the buckets that contain p and o and then choose the minimum frequency of these buckets as their costs. For complex triple patterns involving joining such as the binary join (its cost is $\mathrm{Cost}(|S_1|+|S_2|+|S'|+\sum_{\beta(x)}|L(\beta(x))|)$ where $|S'|$ is the output cardinality), their costs depend on both the input and output cardinalities. In general, $|S'|$ is estimated by $c_m \times |S_1||S_2|$ where $|S_1|$ and $|S_2|$ are the size of the input binding sets S_1 and S_2 and c_m is the merge factor. The many approaches[15] for estimating the cardinality of the joining results $|S'|$ are not covered here.

3.2 Optimization algorithm

There are two cases for query optimization of a SPARQL query. The first case has a tree query graph, while the second case has a graph query graph. The solution for the first case gives an optimal query tree while the second case uses a main-tree-shaped algorithm.

Case 1 A tree-shaped SPARQL query requires two steps to find the optimal execution tree. The first step translates every node into the execution subtree. However, if there are multiple index scans, the algorithm must determine their optimal joining order using a greedy algorithm which first estimates the cost of each index scan (see Section 3.1) and then joins the two index scans with the minimum costs by an intersection. Then, the intersection with the next lowest cost is joined with the index scan by another intersection until all the index scans are joined together.

The second step enumerates all the execution trees by treating each node as the root node and then choosing the tree with the minimum execution cost as the optimal execution plan. Specifically, let v be a root node with k subtrees $t_1,\cdots,t_k$ (i.e., the trees generated by the first step). Assume that the optimal execution tree of each subtree t_i is denoted as e_{t_i}. Then, recursively join the tree made of the root node v with the subtree tree_{t_i} with the minimum cost among tree_{t_1}, $\cdots,\mathrm{tree}_{t_k}$ until all the subtrees are joined together.

Case 2 The graph-shaped SPARQL query uses a heuristic approach based on the concept of the biconnected graph to produce the optimal tree. A biconnected graph[16] is a connected graph with no cutting vertices. A cutting vertex is a vertex whose removal increases the number of connected components. Here, each biconnected subgraph is treated as a virtual tree node which has two advantages for query optimization: (1) the query graph can be transformed into a tree (called main execution tree) and (2) identification of the optimal execution tree for each biconnected subgraph is cheaper than directly optimizing the entire query graph.

The optimization first uses a depth-first-search (DFS) algorithm to identify all the biconnected subgraphs. Then a dynamic programming algorithm is used for each biconnected subgraph to find its optimal subtree. Finally, all the optimal execution trees of the biconnected subgraphs are merged into the main execution tree by cutting vertexes. The greedy algorithm developed in the first case can be used to determine the optimal joining order between all nodes in the main execution tree (the cost of the entire optimal subtree for each biconnected subgraph is used as the cost of the virtual node). The key step in the second case is to transform a biconnected subgraph into an optimal execution tree.

For each biconnected subgraph, a dynamic programming (DP) based optimization algorithm is designed to calculate its optimal tree in $|V|$ phases, where $|V|$ is the number of vertexes in the query graph. Initially, the algorithm seeds the DP table with the variable set S containing only one variable (i.e., the vertex in the query graph). At the j-th phase, the algorithm supposes the variable set S includes j variables, and adds a variable $v \notin S$ into S, which is linked with some variable(s) in S by either normal or triple edges. Then compute the best execution tree for $S \cup \{v\}$. Let the edges linking v with S be E, the algorithm translates each edge $e \in E$ into a corresponding join operation depending on to the edge type that connects the execution tree of S and v together. The other edges $E-\{e\}$ are translated as selection operators and sequentially put on top of the join operation. The output binding set then includes all the variables $S \cup \{v\}$ and the selection only needs to be executed after join. The triple pattern for each selection is determined by the edge constraint. However, the order of the selection operations must also be considered. A greedy strategy is used to choose the selection with the smallest cost on the join and then the next lowest cost selection, and so on. Finally, if any variable can be removed at a tree node w, then an aggregation operator is added for node w. The algorithm repeats these steps until all variables in the subgraph are added into S. The optimal execution tree of the subgraph is obtained after the $|V|$-th variable is put into S.

This algorithm does not consider all possible query plans, but only the left-deep tree. This reduces the complexity of the optimization algorithm whose cost is $O(e2^v)$, where v is the number of nodes and e is the number of edges in the subgraph, and reduces the calculations for the selection on the join as discussed in Section 3.2 (the actual index lookup for the selection can be saved which the output binding set of the join is sorted).

Disjunctive queries This algorithm mainly focuses on how to optimize conjunctive SPARQL queries. However, SPARQL also support UNION and OPTIONAL clauses for disjunctive queries[3]. For UNION, the algorithm first treats all its triple patterns as nested sub-queries. Then the nested sub-queries are transformed into optimal subtrees and merged into the main execution tree by the UNION operator. For OPTIONAL, first all the triple patterns outside the OPTIONAL clause are copied into the OPTIONAL clause based on its semantic and then the optimal subtrees are selected for these triple patterns. Finally, the subtrees are merged into the main execution tree by the left outer join operator. The union and left outer join are described in detail in Liu et al.[17]

4 Performance Study

A prototype system named Airstore was built using Lucence 2.4[18] to evaluate this approach. The algorithm was implemented in the Java SDK 1.6 and compared with the state-of-the-art RDF stores AllegroGraph[19], Sesame[20], YARS2[8], SOR[5], and RDF-3X[11]. The first four systems were implemented in Java, while RDF-3X was developed using C++. The IBM DB2 v9.1 was used as the backend storage of the SOR. All tests were ran on a PC with an Intel Duo Core 6700@2.66 GHz processor, 2.0 GB RAM, and a 140 GB 7200 RPM IDE driver, while the operating system is Ubuntu 8.0.4. The synthetic and real datasets, LUBM[21] and DBpedia[1], were used for performance testing and all tests were repeated 10 times to calculate the average results.

Since the LUBM benchmark queries are designed for ontology reasoning, they were not used as benchmark queries. Instead SPARQL queries (Q1-Q7) were developed with five typical access patterns, point, star, tree, graph, and hybrid. Table 2 lists all the benchmark queries. Each access pattern tested the efficiency of a particular access method with Q1 and Q2 used to evaluate the efficiency of data retrieval with vertex constraints, while Q3 and Q4 were designed for testing joinings with edge constraints. Two graph-shaped queries, Q5 and Q6, were used to study the effectiveness of the cost estimation algorithms. Finally, Q4 and Q6 were combined using the union operation to evaluate the efficiency of different RDF stores on multiple query graphs. Seven other SPARQL queries (Q8-Q14) were designed for the DBpedia dataset[17].

Table 2 SPARQL benchmark queries

ID	Query	Shape
Q1	SELECT distinct ?*x* WHERE { ?*x* ub:researchInterest. ub:Research18. ?*x* rdf:type ub:FullProfessor.}	Point
Q2	SELECT distinct ?*x* WHERE { ?*x* rdf:type ub:UndergraduateStudent.}	Point
Q3	SELECT distinct ?*x* ?*y*1 ?*y*2 ?*y*3 WHERE { ?*x* ub:worksFor ub:University0. ?*x* ub:name ?*y*1. ?*x* ub:emailAddress ?*y*2. ?*x* ub:telephone ?*y*3. }	Star
Q4	SELECT distinct ?*x* ?*z* WHERE { ?*x* ub:advisor ?*y*. ?*y* ub:worksFor ?*z*. ?*a* ub:memberOf ?*z*. ub:Publication5 ub:publicationAuthor ?*a*.}	Tree
Q5	SELECT distinct ?*x* ?*y* ?*z* WHERE { ?*x* rdf:type ub:GraduateStudent. ?*y* rdf:type ub:University. ?*x* ub:memberOf ?*z*. ?*z* ub:subOrganizationOf ?*y*. ?*z* rdf:type ub:Department.?*x* ub:undergraduateDegreeFrom ?*y*. }	Graph
Q6	SELECT distinct ?*x* ?*z* WHERE { ?*x* rdf:type ub:Department. ?*x* ?*y* ?*z*. }	Graph
Q7	Q4 union Q6	Hybrid

The system performance was evaluated based on metrics for the time for data loading and indexing, accuracy of cost estimation and query time. The first metric evaluates the efficiency of the RDF stores for loading and indexing triples. The accuracy of cost estimation was used for testing the effectiveness of the query optimization algorithm (i.e., to what degree the optimization algorithm can speed up SPARQL query processing). The query time is of most interest to end-users since it determines both the usability and scalability of the RDF stores.

4.1 Time for data loading and indexing

The first test evaluates the time used for loading and indexing the LUBM, datasets LUBM20, LUBM40, LUBM60, LUBM80, and LUBM100, where the number of triples varies from 2.8 million to 14 million. All the baseline systems were also ran on these LUBM datasets to measure the scalability.

The results in Fig. 7 show that the performances of airstore, AllegroGraph, RDF-3X, and SOR were similar with increasing data volume with all four systems significantly outperforming Sesame and YARS2. The results show that the IR indexing technique can be used to manage the RDF data since the scalability of airstore is as good as that of the B^+-tree-based RDF stores, such as RDF-3X and SOR.

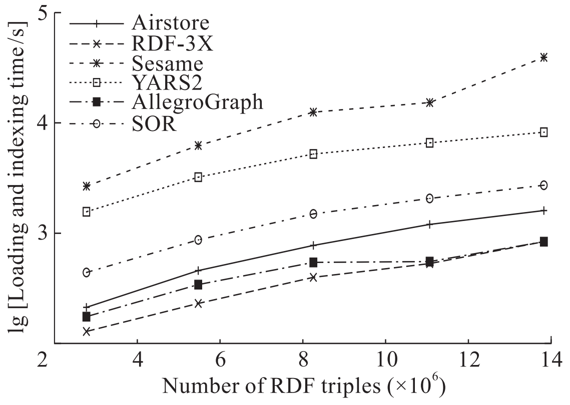

Fig. 7 Time for data loading and indexing

4.2 Accuracy of cost estimation

The accuracy of the cost estimates of the query optimization algorithm was then evaluated by calculating the costs of all possible query plans (including the actual optimal plan) and then comparing their execution times. The accuracy of the cost estimate, ACE, is defined as $\mathrm{ACE}=\frac{\mathrm{qtime}_c}{\mathrm{qtime}_o}$, where qtime_c indicates the execution time of a candidate query plan and qtime_o refers to the execution time of the optimized query plan selected by the algorithm. Thus, larger ACE indicates slower execution trees and, therefore, more accurate execution cost estimates by the algorithm. Only the result of Q5 on LUBM1000 that contains more than 100 million triples is reported due to space limitations[17].

The results in Fig. 8 show that the execution times for the first four query plans are close to the chosen plan, while those for the next seven query plans are 1.2-7 times slower. The execution times of the last seven query plans were 1200 times slower than for Airstore. In the figure, the first plan is the actual optimal plan. These results show that this algorithm very effectively identifies the optimal plan in the restricted searching space.

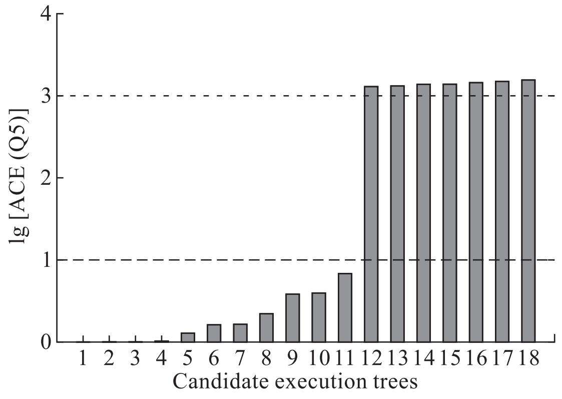

Fig. 8 Accuracy of cost estimation

4.3 Query time

Finally, the query times for the different RDF stores were measured for both the LUBM and DBpedia datasets. The query time for airstore includes both the query optimization time and the query execution times. Six indices were used for Sesame with the full 24 indices used for AllegroGraph with the SOR's performance tuned with the IBM DB2 design advisor to get the best results. The two sets of tests on the LUBM datasets tested the query time for different RDF stores and the scalability for different data volumes (LUBM20, LUBM40, LUBM60, LUBM80, and LUBM100). Then airstore was evaluated using the two large RDF datasets LUBM1000 and DBpedia.

Performance on the LUBM datasets Figure 9 illustrates the query times for all the RDF stores on the LUBM100. The results on LUBM20, LUBM40, LUBM60, and LUBM80 were similar. The results in Fig. 9 show that the query times for Airstore and RDF-3X were the best with very efficient performance in comparison with AllegroGraph, Sesame, YARS2, and SOR. Airstore outperformed RDF-3X on all the queries except for Q5. Airstore outperforms RDF-3X on most queries mainly because the sequential index reading on the inverted index is more efficient than the scanning B^+-tree. For example, for Q2, Airstore only needs to read all the documents with the term UndergraduateStudent to obtain all the answers and with the inverted index, all the documents with the same term are placed near to each other on the disk. For Q5, RDF-3X pipelines the operators to produce the final result while Airstore does not use such a pipeline and has to maintain the binding sets for the intermediate results of the binary and triple joins, which costs more time. In this case, RDF-3X outperforms airstore.

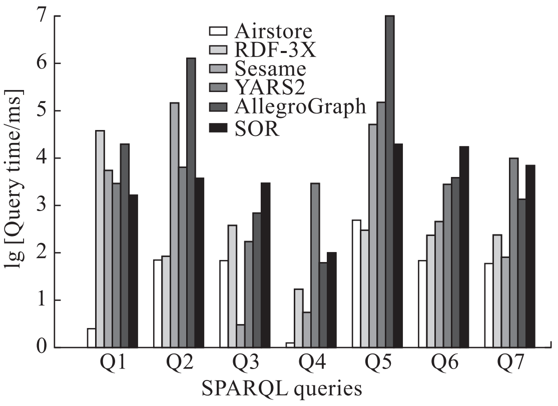

Fig. 9 Query time on LUBM datasets

Thus the Airstore performance on most queries is better than that of RDF-3X, even though RDF-3X was implemented using C++. Therefore, the results show that this RDF storage scheme greatly improves query performance and the IR technique very efficiently indexes the RDF triples.

The scalability of Airstore as indicated by the query time was then evaluated by varying the data volume. The results in Fig. 10 show that the query time for each SPARQL query increases linearly with increasing amount of the triple, which shows the good scalability of Airstore relative to the RDF data population.

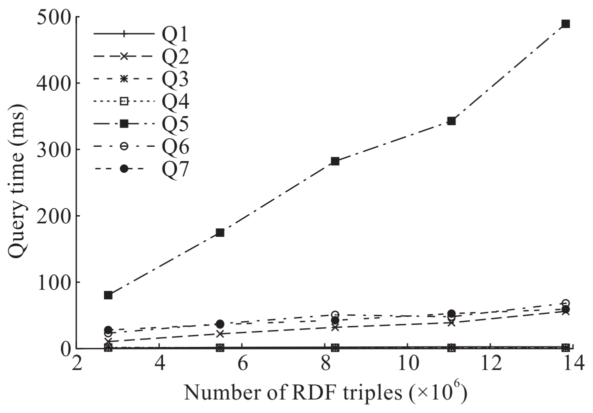

Fig. 10 Airstore query time for large datasets

Performance on large RDF datasets The final tests evaluated the Airstore query performance on two large RDF datasets LUBM1000 with 140 million triples and DBpedia with 120 million triples.

The results in Fig. 11 show that Airstore took only 4.4 s to execute the most expensive query, Q5, with all other queries finishing within 1 s. The results in Fig. 12 show that the slowest query time on the DBpedia dataset was less than 120 ms. Thus, these results show that airstore is scalable and efficient in terms of the data population to be queried. Comparison of Figs. 11 and 12 also show that the performance difference of airstore between the LUBM1000 and DBPedia datasets is mainly caused by the difference in their data distribution and the query selectivity.

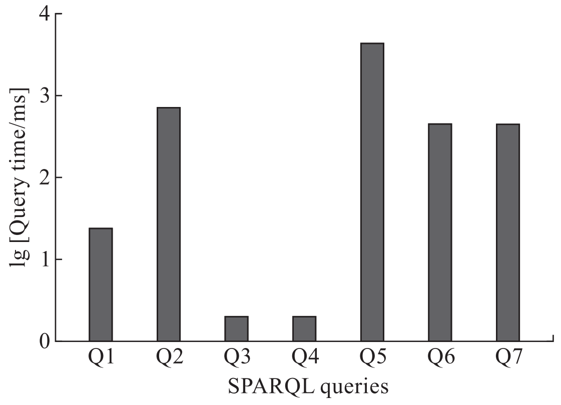

Fig. 11 Airstore query time on LUBM1000

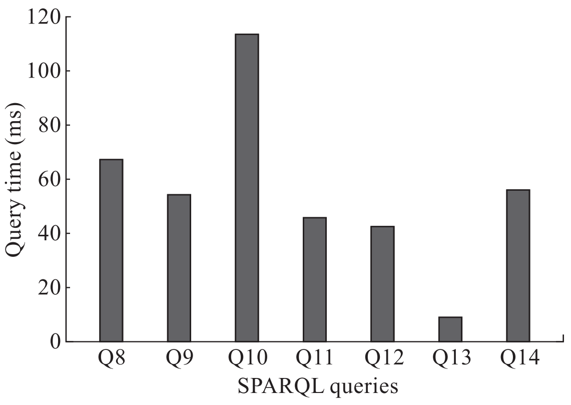

Fig. 12 Airstore query time on DBPedia

5 Conclusions

This paper presents an RDF query engine for efficient SPARQL query processing. The algorithm includes an IR based solution for indexing triples and a set of highly-efficient operators for query optimization and evaluation, a set of RDF statistics for estimating the execution cost of the query plan, and a main-tree-shaped optimization algorithm for identifying the optimal query plan. Tests show that this approach is very efficient and scalable for querying large-scale RDF triples. Current work focuses on the query optimization problem on SPARQL basic graph patterns, and union and optional patterns. Future work will extend this approach to support the filter clause and named graphs in the SPARQL[3] by extending the existing statistics, indexing scheme and operators.

References

[1] DBpedia. http://dbpedia.org/, 2010.

[2] Bizer C, Heath T, Idehen K, et al. Linked data on the web. In: Proc. of World Wide Web. Beijing, China, 2008: 1265-1266.

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

[4] Wilkinson K. Jena property table implementation. In: Proc. of SSWS. Athens, Georgia, USA, 2006.

[5] Ma L, Wang C, Lu J, et al. Effective and efficient semantic web data management on DB2. In: Proc. of SIGMOD: International Conference on Management of Data. Vancouver, Canada, 2008.

[6] Abadi D J, Marcus A, Madden S R, et al. Scalable semantic web data management using vertical partitioning. In: Proc. of Very Large Database. Vienna, Austria, 2007.

[7] Weiss C, Karras P, Bernstein A. Hexastore: Sextuple indexing for semantic web data management. In: Proc. of Very Large Database. Auckland, New Zealand, 2008.

[8] Harth A, Umbrich J, Hogan A, et al. Yars2: A federated repository for querying graph structured data from the web. In: Proc. of International Semantic Web Conference. Pusan, South Korea, 2007.

[9] Zhang L, Liu Q, Zhang J, et al. Semplore: An IR approach to scalable hybrid query of semantic web data. In: Proc. of International Semantic Web Conference. Pusan, South Korea, 2007.

[10] Stocker M, Seaborne A, Bernstein A, et al. SPARQL basic graph pattern optimization using selectivity estimation. In: Proc. of World Wide Web. Beijing, China, 2008.

[11] Neumann T, Weikum G. Rdf-3x: A RISC-style engine for RDF. In: Proc. of Very Large Database. Auckland, New Zealand, 2008.

[12] Selinger P G, Astrahan M M, Chamberlin D D, et al. Access path selection in a relational database management system. In: Proc. of SIGMOD: International Conference on Management of Data. Boston, Massachusetts, USA, 1979: 23-34

[13] Ioannidis Y. The history of histograms. In: Proc. of Very Large Database. Berlin, Germany, 2003.

[14] Ioannidis Y E. Query optimization. *ACM Computing Surveys*, 1996, **28**(1): 121-123.

[15] Florin R, Alin D. Sketches for size of join estimation. *ACM Transaction on Database System*, 2008, **33**(3): 1-46.

[16] Biconnected graph. http://mathworld.wolfram.com/BiconnectedGraph.html, 2010.

[17] Liu C, Xu L, Wang H, et al. Towards efficient SPARQL query processing on RDF data. http://apex.sjtu.edu.cn/apex wiki/Papers, 2009.

[18] Apache Lucene. http://lucene.apache.org/, 2010.

[19] AllegroGraph RDFStore. http://agraph.franz.com/allegrograph/, 2010.

[20] OWLIM semantic repository. http://ontotext.com/owlim/, 2010.

[21] Guo Y, Pan Z, Heflin J. LUBM: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 2005, **3**(2): 158-182.