Semplore: An IR Approach to Scalable Hybrid Query of Semantic Web Data

Lei Zhang[1], QiaoLing Liu[2], Jie Zhang[2], HaoFen Wang[2],
Yue Pan[1], and Yong Yu[2]

[1] IBM China Research Lab
Beijing, 100094, China
lzhangl,panyue@cn.ibm.com
[2] Department of Computer Science & Engineering
Shanghai Jiao Tong University, Shanghai, 200240, China
{lql,zhangjie,whfcarter,yyu}@apex.sjtu.edu.cn

Abstract. As an extension to the current Web, Semantic Web will not only contain structured data with machine understandable semantics but also textual information. While structured queries can be used to find information more precisely on the Semantic Web, keyword searches are still needed to help exploit textual information. It thus becomes very important that we can combine precise structured queries with imprecise keyword searches to have a hybrid query capability. In addition, due to the huge volume of information on the Semantic Web, the hybrid query must be processed in a very scalable way. In this paper, we define such a hybrid query capability that combines unary tree-shaped structured queries with keyword searches. We show how existing information retrieval (IR) index structures and functions can be reused to index semantic web data and its textual information, and how the hybrid query is evaluated on the index structure using IR engines in an efficient and scalable manner. We implemented this IR approach in an engine called Semplore. Comprehensive experiments on its performance show that it is a promising approach. It leads us to believe that it may be possible to evolve current web search engines to query and search the Semantic Web. Finally, we breifly describe how Semplore is used for searching Wikipedia and an IBM customer's product information.

1 Introduction

With more and more structured and semantic information made avaiable on the Semantic Web, structured queries such as SPARQL can be used to find information more precisely. At the same time, current web search is dominated by the form of keyword searches. Although precise structured queries generally produce far better results than imprecise keyword searches, keyword search capability is still needed in Semantic Web because: (1) The huge amount of textual information in the (Semantic) Web will remain to be a valuable source of information that need be exploited using keyword searches; (2) Users often have vague information needs that can hardly be expressed as formal queries; and (3) keyword

searches are extremely simple to use. It thus becomes very important that we can combine precise structured queries with imprecise keyword searches to have a hybrid query capability. On the other hand, as an extension to the current Web, Semantic Web will have an even larger volume of data and textual information. Hybrid queries against such large volume of web-scale data must be evaluated in a very scalable way.

Current research on searching or querying Semantic Web uses either an IR-based (e.g. [1–3]) or a DB-based (e.g. [4–7]) approach. The IR-based work does not provide structured query capability and the DB-based work lacks support to keyword searches. Few work [8] tries to combine them to achieve hybrid query capability. DB-based work pays more attention to the scalability of query-answering and it relies on database's various indices and query optimization algorithms to support efficient evaluation of complex queries. However, taking DB engines to support complex queries on web-scale data is still a big challenge. In contrast, IR engine is a special-purpose engine supporting only keyword searches but has proven to be able to scale to the size of the Web. Current web search engines have developed scalable method to process keyword searches using classic IR techniques with distributed and parallel backend infrastructure [9]. This inspires us to trade query capability for scalability and to try an IR approach of indexing, querying and searching of semantic web data. The problem then breaks down to the following points:

– What is the hybrid query capability ?
– How to index semantic web data using existing IR index structures which are designed for textual information ?
– How to use IR engines to answer the hybrid queries and maintain the efficiency and scalability ?

In this paper, we show how we tried to solve these problems and how we implemented the solution in Semplore using a popular open-source IR engine – Lucene. Our experiments show that the IR approach is promising and it leads us to believe that it may be possible to evolve current web search engine's powerful backend IR infrastructure for querying and searching the Semantic Web and ultimately evolve them to web-scale semantic web search engines.

The paper is organized as follows. Section 2 defines the hybrid query capability. Section 3 describes in detail the methods of indexing and querying of semantic web data using IR index structure and engine. Section 4 then reports the experiment results of Semplore and briefly describes its two applications. We discuss related work in Section 5 and conclude the paper in Section 6.

2 Hybrid Query Capability

[10] introduced a DL-based formal conjunctive query language for Semantic Web. Conjunctive queries are also the formal core of SPARQL query language [11]. We thus use conjunctive queries as the basis of the hybrid query. We then make an extension to ordinary conjunctive queries to combine keyword search for

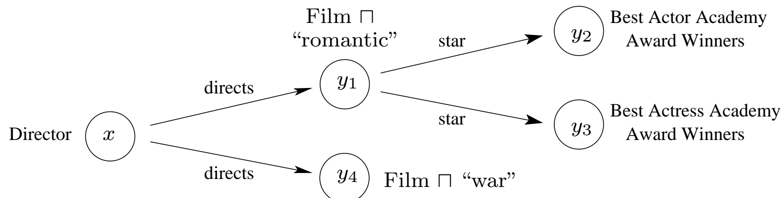

Fig. 1. An Example Query Graph

semantic web data as well. The idea is that we view the searched keywords as a "virtual" concept called keyword concept W. An individual will be regarded as an instance of a keyword concept W if the textual content of any of its datatype properties contains the searched keywords in W (i.e., we adopt a boolean IR model). This idea derives from our previous work in [8].

Based on that, we can formally define the hybrid query capability of Semplore. A *unary hybrid conjunctive query* q over a knowledge base K is a query expression of the form

$$q(x) \leftarrow \exists \overrightarrow{y}.conj(x, \overrightarrow{y})$$

where x is called the target variable, $\overrightarrow{y}$ are existentially quantified variables called non-distinguished variables, and $conj(x, \overrightarrow{y})$ is a conjunction of terms of the form $C(z)$, $R(z_1, z_2)$, or $R^-(z_1, z_2)$. z, z_1, z_2 are individuals in K or variables in x or $\overrightarrow{y}$. R is a role/relation name and C (or D) is a concept expression that is a boolean combination of concept name A and keyword concept W:

$$C, D := \top \mid \bot \mid A \mid W \mid \{i\} \mid C \sqcap D \mid C \sqcup D \mid \neg C$$

The answer to the query w.r.t K is the set defined by $\{a \in O \mid K \models q[x/a]\}$, where O denotes the set of all individual names in K, and $q[x/a]$ denotes the query q with all occurrences of variable x substituted by the individual name a.

The above query can be depicted as a directed graph, where the nodes are variables or individual names and the edges are relations connecting them. Concept expressions and relation names provide labels for nodes and edges respectively.[3] Fig.1 shows an example of such a query graph and [10] has details about the definition of a query graph.

In this paper, we restrict to queries whose graph patterns are trees, as in [10]. We also restrict the query result to be unary (i.e., a single target variable x in the query). These restrictions lead to a much more simplified and hence efficient query evaluation procedure, while a large portion of information needs can still be expressed. This is the major place where we trade query capability for scalability.

The query capability of Semplore can then be defined primarily as *unary tree-shaped hybrid query*. It's not hard to see that the unary tree-shaped hybrid

[3] If the node is an individual i, the concept expression is $\{i\}$. If a node have no label on it, we add the concept expression $\top$ as its label.

Table 1. Translating Semantic Web Data to "Documents" in IR

Document	Field	Term
concept C	subConOf	super-concepts of C
	superConOf	sub-concepts of C
	text	tokens in textual properties of C
relation R	subRelOf	super-relations of R
	superRelOf	sub-relations of R
	text	tokens in textual properties of R
individual i	type	concepts that i belongs to
	subjOf	all relations R that $(i, R, ?)$ is a triple in data
	objOf	all relations R that $(?, R, i)$ is a triple in data
	text	tokens in textual properties of i

queries without keyword concepts is a strict subset of SPARQL queries. In the next section, we show that the hybrid query capability can be achieved using IR engines on semantic web data.

3 Semplore Engine

3.1 Index Structure

IR indexing is based on the concepts of documents, fields(e.g. title, abstract, etc.), and terms. Based on the classic inverted index structure, IR engines can efficiently retrieve documents given a boolean combination of pairs of (field, term) as a query. Current web search engine has implemented this technique on the web scale.

Our intuition is that if we treat individuals as documents and their associated concept names as terms, we can then retrieve all individuals of a given concept by inputing the concept name as a query term into the IR engine. Extending this intuition, we realize that we can answer many kinds of semantic queries using IR engines if we translate semantic web data into documents, fields and terms in a proper way as shown in Table 1. After the translation, IR engine can index semantic web data in its inverted index structure and provide retrieval functions over the data. In addition, in order to return inferred data in query result, we require that the semantic web data be preprocessed by a reasoning engine to contain all inferred data.

The idea of treating concepts as terms to index individuals is not new. Previous work [12] proposed and analyzed more complex labeling schemes for indexing semantic web data. What's new in our work, however, is on how to index relation instances (s, R, o). The index should enable us to find all the objects of a relation with a given set of subjects (e.g. find all the films directed by some Chinese director: $\{f \mid \texttt{directs}(d, f) \wedge \texttt{ChineseDirector}(d)\}$) and vice versa.

We propose an approach called PosIdx to index a relation instance (s, R, o). Recall that in the classic inverted index structure, for each term there is a posting list of documents that contain the term. In addition, for each document in the

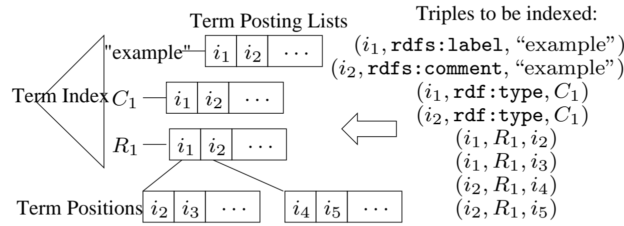

Fig. 2. PosIdx Index Structure Example

posting list, there is also a list of positions showing where the term appears in the document. In the PosIdx method, subject s is treated as a document with a field named `subjOf` and a term R in the field. Object o is stored as a "position" of the term R in the document s. We use the position list to store the objects of relation R under subject s. An example index structure is depicted in Fig.2 (field information is omitted for brevity). The `objOf` field is a symmetric case of `subjOf` and is also not shown in Fig.2. If we see `subjOf` as a field for indexing instances of relation R, the `objOf` field can then be seen as indexing instances of relation R^-.

As to the physical storage and access of the logical inverted index structure , it has been thoroughly studied in the IR field, which results in many optimized methods, such as byte-aligned index compression [13] and self-indexing [14]. Furthermore, in the proposed PosIdx method, relation objects enjoy the benefit of spatial locality for fast access, because positions of a term are usually physically stored together and continuously in modern IR engines. In fact, in our Semplore implementation, we enjoy all the above-mentioned optimizations and benefits because they are alreay built into the underlying IR engine Lucene.

The use of IR approach also enables us to leverage the heavy optimizations on the architectures and algorithms for building index in IR field (e.g.[15]). In a real (Semantic) Web search engine, advanced techniques such as MapReduce [16] can be applied on a cluster of machines to speed up the index building process, thanks to the simplicity of the IR index structure.

3.2 Query Evaluation

Based on the above index, Semplore reuses IR engine's merge-sort based boolean query evaluation method and extends it to answer the unary tree-shaped hybrid query defined in Section 2. In the following, we first introduce and explain some basic operations and their corresponding notations, then we describe the query evaluation algorithm of Semplore.

Basic Operations We generalize the notion of a posting list to an Ascending Integer Stream (AIS) which can be accessed from the smallest integer to the largest one. By adding additional indexing structures to the inverted index (e.g.,

self-indexing [14]), modern IR engines can supply a very efficient stream reader for a posting list AIS.

(1) Basic retrieval: (f, t)

Given a field f and a term t, (f, t) retrieves the corresponding posting list from inverted index. This is a standard IR operation. The output of this operation is an AIS. For example, under Table 1's structure, $(\texttt{type}, \texttt{ChineseDirector})$ will retrieve all individuals of the `ChineseDirector` concept as an AIS.

(2) Merge-sort: $m(S_1, op, S_2)$

S_1 and S_2 are two AISs and op is a binary operator which can be $\cap$, $\cup$ or $-$. Merge-sort computes $S_1\, op\, S_2$ and returns a new AIS. Merge-sort can be nested to compute boolean combinations of multiple AISs. IR research has developed efficient algorithms to do nested merge-sort on AISs.

(3) Concept expression evaluation: $\curlywedge(C)$

The input of this operation is a concept expression C as defined in Section 2 which is a boolean combination of concept name A and keyword concept W. The output of this operation is an AIS containing all the IDs of the individuals of C. This operation can be implemented using basic retrievals and nested merge-sort operations, thus is also readily available in modern IR engines. For example, if C is $\texttt{Film} \sqcap$ "war", the $\curlywedge(C)$ operation can then be achieved through two basic retrievals and one merge-sort: $m(\,(\texttt{type}, \texttt{Film}), \cap, (\texttt{text}, \text{"war"})\,)$

(4) Relation expansion: $\bowtie(S_1, R, S_2)$

The input of the operation is a relation R and two AISs S_1 and S_2 that contain individual IDs. The operation computes the set $\{y \mid \exists x : x \in S_1 \wedge (x, R, y) \wedge y \in S_2\}$ and returns it as an AIS. For example, $\bowtie$ $(\curlywedge(\texttt{ChineseDirector}), \texttt{directs}, \curlywedge(\texttt{DocumentaryFilm}))$ can be used to find all documentary films directed by some Chinese director. This operation is not available in a classic IR engine. We will show later how it can be computed on the index structure we proposed.

Evaluation Algorithm Based on the above four basic operations, Algorithm 1 shows how a *unary tree-shaped hybrid query* defined in Section 2 can be evaluated. The algorithm can be visually imagined as traversing the query tree in the depth-first order. It evaluates the concept expression of each vertex when moving forward and uses results of children to constrain the results of parent when moving backward. It will terminate in $2 * E$ steps each of which is either a $\curlywedge(C)$ operation or a $\bowtie(S_1, R, S_2)$ operation.

Taking Fig.1 as an example with target variable x, we first reach to leaf vertex y_2 via x and y_1, and compute their results $S[x], S[y_1], S[y_2]$. Then $S[y_1] = \bowtie (S[y_2], \texttt{star}^-, S[y_1])$ is computed when we move backward from y_2 to y_1. Similarly, y_3 is traversed and its result also adds constraint to result of its parent y_1. Next, result of root vertex x is updated by $S[x] = \bowtie (S[y_1], \texttt{directs}^-, S[x])$. Finally, y_4 is traversed and the result of root x is updated again, which is the final answer.

Algorithm 1: Query Evaluation Algorithm

Input : A unary tree-shaped hybrid query $Q(t)$ with graph $G = (V, E)$ and target variable t on the vertex $v_t \in V$; Each vertex $u \in V$ has a concept expression C_u as its label and each edge $(u, v) \in E$ has a relation $R_{(u,v)}$ as its label. (Note that $R_{(u,v)}$ is equal to $R^-_{(v,u)}$.)

Output: An AIS containing the IDs of individuals in the answer set of $Q(t)$

1 **foreach** *vertex* $u \in V$ **do**
2 $\quad$ $checked[u] = false$; $S[u] = null$;
3 $DFS(v_t)$;
4 **return** $S[v_t]$

Procedure `DFS(`u`)`

1 $checked[u] = true$;
2 $S[u] = \curlywedge(C_u)$;
3 **foreach** *vertex* v *such that* $(u, v) \in E$ *or* $(v, u) \in E$ **do**
4 $\quad$ **if** $(checked[v] == true)$ **then continue** v;
5 $\quad$ $DFS(v)$;
6 $\quad$ **if** $((v, u) \in E)$ **then** $S[u] = \bowtie (S[v], R_{(v,u)}, S[u])$;
7 $\quad$ **else** $S[u] = \bowtie (S[v], R^-_{(u,v)}, S[u])$;

Relation Expansion Among the four basic operations used for query evaluation, relation expansion $\bowtie (S_1, R, S_2)$ is the one that is not directly supported by current IR engines. However, it can be evaluated in four steps using additional operations. Fig.3 shows how this is done on the PosIdx index structure.

First, we compute the valid subjects that have R relations: $S = m(S_1, \cap, (\texttt{subjOf}, R))$.[4] Second, we find the objects for each valid subject $s \in S$ using an operation called $\texttt{getObjects}(s, R)$ which returns the object set $\{o \mid (s, R, o)\}$ as an AIS, given s and R. Third, we union all these object sets and sort the result set to obtain a new AIS S_O. This step is encapsulated in an operation called $\texttt{massUnion}(S, R) = S_O = \bigcup_{s \in S} \texttt{getObjects(s, R)}$. Finally, we do a merge-sort $m(S_O, \cap, S_2)$ to obtain the final result. This final step can be integrated into the massUnion operation as we will show later. In short, relation expansion can be completed using basic retrieval, merge-sort and two additional operations: getObjects and massUnion.

When the number of valid subjects in S is large, the $\texttt{massUnion}(S, R)$ operation becomes expensive because it has to union and sort a large number of sets stored on disk, which is an I/O bound operation. In Fig.3, these sets are the shaded segments in term R's position list. If we still do a streaming merge-sort on all the sets to obtain all the objects, it would incur a prohibitive I/O cost because it will lead to a large number of back-and-forth disk seeks. One way to save I/O cost is to selectively union a subset of all the sets that can still cover

[4] If the relation is R^-, replace `subjOf` with `objOf`. Similarly, in the following steps `getSubjects`(o,R) is used instead of `getObjects`(s,R).

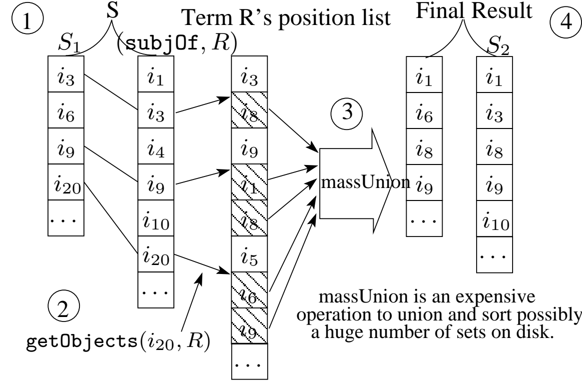

Fig. 3. Relation Expansion on PosIdx

Algorithm 3: BVI for $\bowtie (S_1, R, S_2)$

---

1 Do a basic retrieval $O_R = (\texttt{objOf}, R)$; Let $N = |O_R|$ and $T = 0$;
2 Allocate two bit vectors B_1 and B_2 of size N and initialize them to all 0s;
3 Do $m(O_R, \cap, S_2)$. During the merge-sort, set $B_2[i] = 1$ for the ith element in O_R that is also in S_2. Let M be the number of 1 bits in B_2;
4 **foreach** $s \in m(S_1, (\textit{subjOf}, R))$ **do**
5 $\quad O_s = \texttt{getObjects}(s, R)$;
6 $\quad$ **foreach** *sequence number* $i \in O_s$ **do**
7 $\quad\quad$ **if** $(B_2[i] == 1)$ **then**
8 $\quad\quad\quad B_2[i] = 0;\ T++;\ B_1[i] = 1;$
9 $\quad\quad\quad$ **if** $(T == M)$ **then** goto 10;
10 Construct the result AIS G from the O_R stream using the filtering condition: the ith element of O_R is in G if $B_1[i]$ is 1;
11 **return** G *as an AIS*

---

all the objects. However, selecting such a subset is the Set-Cover problem which is NP hard [17].

We use a simple yet effective approach for the massUnion operation: read in all the sets one by one using the $\texttt{getObjects}(s, R)$ operation and use a bit vector to track the union result. For convenience, we use sequence numbers to identify the objects in the position list. Suppose the set of all distinct objects of relation R is $O_R = \{o \mid \exists s : (s, R, o)\}$ and $N = |O_R|$. Sorting the set O_R on object IDs ascendingly gets a list of objects $o_1, o_2, \ldots, o_N$. The sequence number of object o_i is i under relation R. We thus can allocate a limited size bit vector to track which object is in the result of the relation expansion. The algorithm is called Bit Vector Intersection (BVI) and shown in Algorithm 3.

Note that $N = |O_R|$ in line 1 can be directly obtained from inverted index as the document frequency of the term R without any computation at run time. Line 4 and 5 can be implemented together on the PosIdx index very efficiently

using a sequential scan on the position list during merge-sort, with the help of self-indexing [14].

The worst case time complexity of the algorithm is linear to the number of all objects of valid subjects but it will stop when all the possible results have been found in line 9. Meanwhile, because of the stop condition, the execution time of the algorithm won't increase definitely with the size of the valid subjects $S = m(S_1, \cap, (\texttt{subjOf}, R))$. In our experiments, we are surprised to find that the time even decreases when the size of valid subjects $|S|$ exceeds an threshold. This is because in one disk I/O more O_s sets (i.e, the shaded blocks in Fig. 3) can be read in due to the increased locality of these sets when $|S|$ becomes large. It shows the benefit of the PosIdx index structure in which relation objects enjoy the spatial locality for fast access.

The space requirement of Algorithm 3 is linear to N which may be quite large. But in practice, a 256MB memory can already hold the two bit vectors for N as large as 1 billion and the memory can be reused across multiple executions of the algorithm.

3.3 Implementation

We implemented all the above index and query evaluation algorithms in the Semplore engine. It uses the popular open source IR engine Lucene to do classic inverted indexing and perform all the basic IR operations. V2.0.0 Java version of Lucene is used which implements byte-aligned index compression and self-indexing. We use the COLT package[5] for fast bit vector operations in Semplore.

4 Experiment and Application

4.1 Experiment Setup

Both synthetic and real world semantic web data are used in our experiment. We use LUBM [18] benchmark data sets, from LUBM(5,0) to LUBM(400,0), to test Semplore's scalability. To evaluate Semplore's performance on hybrid queries, we use Wikipedia content as the real world knowledge base because it contains both rich textual properties and relationships between entities. We combine TBox from YAGO [19] and ABox from DBpedia [20] as our Wikipedia dataset. Some simple heuristic rules are used for data cleaning.

All the data sets are preprocessed by Minerva [7] to do reasoning. IBM DB2 v8.1.7 is used as the backend database of Minerva. Table 2 shows the number of triples of each data set after reasoning. We then index the inferred data set in Semplore, by extracting the triples from Minerva. The index time of Semplore excludes the time of extracting triples from Minerva. Table 2 shows Semplore's index performance w.r.t different data sets using a single index thread. We can see that both the index time and space of Semplore increase basically linearly with the size of dataset. The following query evaluation experiments are carried

[5] http://dsd.lbl.gov/∼hoschek/colt/

Table 2. Datasets and Index Performance

Dataset	Triples After Reasoning	Index Time(s)	Index Space(MB)
LUBM(5,0)	729,253	198	25.6
LUBM(10,0)	1,485,029	1379	52
LUBM(20,0)	3,135,033	2712	110
LUBM(50,0)	7,763,131	9699	272
LUBM(400,0)	62,233,512	49783	2150
Wikipedia	13,991,407	25873	1320

Table 3. Query Response Time for LUBM Datasets (ms) ('Sem': Semplore, 'Min': Minerva, 'Ses': Sesame; As in [21], we omit Sesame's performance on LUBM(50) and LUBM(400) due to its excessive loading time.)

Query	LUBM(5)			LUBM(10)			LUBM(20)			LUBM(50)		LUBM(400)	
	Sem	Min	Ses	Sem	Min	Ses	Sem	Min	Ses	Sem	Min	Sem	Min
BQ1	0	203	47	0	359	16	16	703	16	16	1407	94	10890
BQ3	0	156	47	0	203	31	0	360	31	0	719	0	5391
BQ4'	0	47	0	0	31	0	0	47	15	0	63	0	297
BQ5	0	156	218	0	156	250	0	312	204	0	578	16	4235
BQ6	0	109	1000	0	234	1953	16	484	2968	32	1000	203	7750
BQ7'	0	266	47	0	375	32	0	765	32	0	1813	0	13750
BQ8'	0	203	2781	0	234	4468	0	1078	8874	16	2609	0	19688
BQ10	0	250	0	0	344	0	0	734	32	0	1406	0	10750
BQ11	0	47	94	0	47	110	0	47	109	0	62	0	266
BQ12'	16	62	125	0	78	703	0	109	2625	0	218	15	1313
BQ13	0	94	16	0	31	15	0	32	78	15	31	47	63
BQ14	0	93	578	0	156	1125	0	360	2672	31	797	156	5922
NQ1	47	453	4890	78	1046	12594	156	2016	53468	360	5438	2938	67078
NQ2	31	2625	281	78	4688	609	172	7094	532	406	17922	3297	523815

out on these indices. For Sesame [4], we used the 1.2.6 version with MySQL 5.0.21 as its backend RDBMS. All the experiments are conducted on a normal desktop PC with Pentium 4 CPU of 3.2 GHz and 2G memory, running Microsoft Windows Server 2003 with Sun Java JRE 1.5.0. The Wikipedia data set and the test queries used in the following experiments are all available from our Semplore web site[6] which also contains live demos.

4.2 Query Evaluation

First, we consider the 14 LUBM benchmark queries(BQ) in [18]. Some modifications are applied to the queries due to the unary tree-shaped query capability of Semplore. Queries with multiple target variables (BQ4, BQ7, BQ8, BQ12) are modified to unary ones, in which we all choose x as target variable. We also remove two cyclic queries (BQ2, BQ9). The remaining 12 queries are all path

[6] http://apex.sjtu.edu.cn/apex_wiki/Demos/Semplore

Table 4. Hybrid Queries and Their Response Time on Wikipedia Dataset (ms)

Query Set	Set Size	Pattern	Example Query	Ave Time	Max Time
QS1 (TBox)	10	node	Find all concepts with label containing the term "chinese"	8	16
QS2 (ABox)	10	node	Find documentary films about "world war"	7	16
QS3 (ABox)	20	path ave_len:2.3 max_len:3	Find films reaching "Academy Award" and starring a "James Bond" actor; Find artists origining from New York City and having "hip hop" albums	11	94
QS4 (ABox)	10	tree ave_dep:2.2 max_dep:3	Find directors who have directed "romantic" films starring Best Actor Academy Award Winners and Best Actress Academy Award Winners and also films about "war"	25	101

queries and primarily designed to test reasoning capability. We therefore add two new tree-shaped queries (NQ1, NQ2) to test Semplore.

We compare the query evaluation performance on pure structured queries of Semplore with the DB-based ontology stores Minerva and Sesame. Table 3 shows the response time of the three systems for all the 14 queries. The query time includes traversing the whole result set of each query. All the three systems have inferred data materialized and do not have run-time reasoning. Thanks to the benefit of spatial locality for fast access using PosIdx index, Semplore achieves very good scalability on all the 14 queries. Its maximal query processing time of the 12 BQs is less than 0.25 second even for the largest LUBM(400) data set, which is much better than that of Minerva. Considering the two NQs, although Semplore needs more relation expansion operations, it manages to return answers within 4 seconds and still keeps orders of magnitude faster than Minerva and Sesame. One important reason is that while Minerva and Sesame depends on complex nested table joins, Semplore turns to relation expansions along the edges of query graph, which are more lightweight operations. This advantage comes from Semplore's designed trade-off between query capability and scalability. Certainly, we note that Minerva is well-designed to deal with reasoning, whereas Semplore focuses more on indexing and querying. The comparison here is somewhat unfair on this aspect, but it certainly shows that IR-based approach is also promising for querying semantic web data.

What we test above is on queries without keyword searches. Since there are no benchmarks for hybrid queries yet, we created four sets of queries for the Wikipedia dataset with increasing complexity in query patterns, from node, path to tree. The queries include 10 TBox queries and 40 ABox queries. All the queries contain one or more keyword searches to express vague information needs. The details of the hybrid queries are shown in Table 4.

The table also summarizes the response time of Semplore w.r.t the four query sets. The query time includes traversing the whole result set of each query. By providing sub-second query respond time on the 1.3GB Wikipedia data set

index , Semplore provides good scalability on hybrid queries as well. This mainly benefits from its unified index method for both semantic information and textual information.

4.3 Applications

In this section, we very briefly describe two applications of the Semplore engine. The first one is a search application for the Wikipedia data set we used in the experiments. To provide both structured queries and keyword seaches in the user interface (UI), we adopted and extended the faceted search paradigm [22]. Users can mix browsing, querying, searching and discovering in the search application easily. User's actions in the UI are translated to hybrid queries to the Semplore engine. Our users reported that it improves access to the Wikipedia's rich structural and textual information. The application can be accessed from our Semplore web site.

In another application, we exploit the use of semantic web technologies for product information management (PIM) as reported in [23]. Modern enterprises have very complex product information that contains a large variety of categories, attributes, relationships and rich textual descriptions. Hybrid query capability and easy-to-use UI is needed for the PIM user to browse, query and search all the products. We modeled a real IBM customer's product information in OWL and converted a subset of the real data into RDF. We then use the Semplore engine and the extended faceted search developed above to help user browse, query and search the product information. Semplore successfully delivered answers to hybrid queries with sub-second performance on the 2GB index of the RDF data set. It thus augments the semantic PIM system in [23] with a powerful search engine.

5 Related Work

Existing work on querying and searching semantic web data can be roughly divided into two categories: IR-based and DB-based.

Swoogle [3] is a crawler-based indexing and retrieval system for the Semantic Web. It uses IR engine to index the crawled semantic web documents using either n-gram or URIrefs as terms and computes a ranked list of these documents given the search terms. [1] augments keyword search with semantic information collected from different sources while [2] uses keyword search results as a seed to do spread activation on semantic networks. These work takes a pure IR approach and does not support structured queries on the semantic web data. [24] combines keyword searches and structured queries but focuses on result ranking.

Most DB-based work such as [4–7] rely on relational database engines for indexing and querying. The primary database schema used for storing triples is a vertical schema with various optimizations. Query answering is achieved primarily through self-join on the vertical table. Indices are built on combinations of subjects, predicates and objects. These DB-based approaches have no or weak

support to keyword searches. YARS [25] does not rely on a relational database but uses similar index structures (i.e. B+ trees). Both YARS and Kowari[7] support keyword searches but they use IR engine separately for that purpose only. BigOWLIM[8] is a scalable repository supporting structured queries but uses its own proprietary storage and index format. LUBM [18] benchmark are developed alongside those work to evaluate semantic web knowledge base systems [21].

Our work is certainly related to the cross field of DB and IR. Actually, the integration of IR and DB is a long desired research goal in the information management area. IR methods have been successfully borrowed to DB area for many tasks such as keyword searches (e.g. [26]) and searching XML (e.g. [27]). At the same time, work in IR area is also adding more structures and semantics to the keyword search. [28] borrows XML Fragment query language to express more semantic-rich queries for searching a semantically annotated text corpora. These work does not address the hybrid query capability for semantic web data (i.e. the RDF triples) but most of them supports ranking of results which is currently lacking in our method.

6 Conclusion

Having hybrid query capability and scalable query evaluation algorithm is important in querying semantic web data. In this paper, we define such a hybrid query capability that combines structured queries with keyword searches. Unlike most of the current work that uses DB engines, our work uses existing IR index structure and engine to support the hybrid query capability. Our work shows that this IR approach not only is possible but also achieves good scalability due to its trade-off on query capability and reuse of existing IR optimizations. It leads us to believe that it may be possible to evolve current web search engine's powerful backend IR infrastructure for querying and searching the Semantic Web and ultimately evolve them to web-scale semantic web search engines. Our future work includes the evaluation of the effectiveness of the hybrid query capability and the support of ranking of results.

References

1. Guha, R., McCool, R., Miller, E.: Semantic search. In: Proc. of the 12th Intl. Conf. on World Wide Web. (2003)
2. Rocha, C., Schwabe, D., Aragao, M.P.: A hybrid approach for searching in the semantic web. In: Proc. of the 13th Intl. Conf. on World Wide Web. (2004)
3. Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R.S., Peng, Y., Reddivari, P., Doshi, V., Sachs, J.: Swoogle: a search and metadata engine for the semantic web. In: Proc. of the 13th ACM CIKM Conf. (2004)
4. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. In: Proc. of 1st Intl. Semantic Web Conference (ISWC2002). (2002)

[7] http://www.kowari.org
[8] http://www.ontotext.com/owlim/big/

5. Pan, Z., Heflin, J.: DLDB: Extending relational databases to support semantic web queries. In: Workshop on Practical and Scalable Semantic Systems. (2003)
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient SQL-based RDF querying scheme. In: Proc. of the VLDB 2005. (2005)
7. Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y., Pan, Y.: Minerva: A scalable OWL ontology storage and inference system. In: Proc. of the 1st Asian Semantic Web Conference (ASWC2006). (2006)
8. Zhang, L., Yu, Y., Zhou, J., Lin, C., Yang, Y.: An enhanced model for searching in semantic portals. In: Proc. of the 14th Intl. World Wide Web Conf. (2005)
9. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks **30**(1-7) (1998)
10. Horrocks, I., Tessaris, S.: Querying the semantic web: A formal approach. In: International Semantic Web Conference. (2002) 177–191
11. Siberski, W., Pan, J.Z., Thaden, U.: Querying the semantic web with preferences. In: International Semantic Web Conference. (2006)
12. Christophides, V., G.Karvounarakis, D.Plexousakis: Optimizing taxonomic semantic web queries using labeling schemes. Journal of Web Semantics **1**(2) (2004)
13. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: Proc. of the 25th ACM SIGIR Conf. (2002)
14. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Transaction on Information Systems **14**(4) (1996) 349–379
15. Melink, S., Raghavan, S., Yang, B., Garcia-Molina, H.: Building a distributed full-text index for the web. ACM Trans. Inf. Syst. **19**(3) (2001)
16. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proc. of the 6th Symp. on Operating System Design and Implementation. (2004)
17. Cormen, T.H., et al: Introduction to Algorithms. MIT Press (2001)
18. Y.Guo, Z.Pan, J.Heflin: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2) (2005)
19. Suchanek, F.M., Kasneci, G., Weikum, G.: YAGO: A core of semantic knowledge unifying wordnet and wikipedia. In: Proc. of WWW 2007. (2007)
20. Auer, S., Lehmann, J.: What have innsbruck and leipzig in common? extracting semantic from wiki content. In: Proc. of ESWC 2007. (2007)
21. Y.Guo, Z.Pan, J.Heflin: An evaluation of knowledge base systems for large OWL datasets. In: Proc. of the 3rd Intl. Semantic Web Conf. (2004)
22. Yee, K.P., Swearingen, K., Li, K., Hearst, M.: Faceted metadata for image search and browsing. In: Proc. of CHI 2003. (2003)
23. Brunner, J.S., Ma, L., Wang, C., Zhang, L., Wolfson, D.C., Pan, Y., Srinivas, K.: Explorations in the use of semantic web technologies for product information management. In: Proc. of WWW 2007. (2007)
24. Nejdl, W., Siberski, W., Thaden, U., Balke, W.T.: Top-k query evaluation for schema-based peer-to-peer networks. In: Proc. of ISWC 2004. (2004)
25. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. In: Proc. of the 3rd Latin American Web Congress, IEEE (2005)
26. Liu, F., Yu, C., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In: Proc. of SIGMOD 2006. (2006)
27. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: Proc. of the 2004 ACM SIGMOD Conference. (2004)
28. Chu-Carroll, J., Prager, J., Czuba, K., Ferrucci, D., Duboue, P.: Semantic search via XML Fragments: a high-precision approach to IR. In: Proc. of the 29th ACM SIGIR Conference. (2006)