Semplore: A scalable IR approach to search the Web of Data

Haofen Wang, Qiaoling Liu, Thomas Penin, Linyun Fu, Lei Zhang, Thanh Tran, Yong Yu, Yue Pan

Abstract

The Web of Data keeps growing rapidly. However, the full exploitation of this large amount of structured data faces numerous challenges like usability, scalability, imprecise information needs and data change. We present Semplore, an IR-based system that aims at addressing these issues. Semplore supports intuitive faceted search and complex queries both on text and structured data. It combines imprecise keyword search and precise structured query in a unified ranking scheme. Scalable query processing is supported by leveraging inverted indexes traditionally used in IR systems. This is combined with a novel block-based index structure to support efficient index update when data changes. The experimental results show that Semplore is an efficient and effective system for searching the Web of Data and can be used as a basic infrastructure for Web-scale Semantic Web search engines.

1. Introduction

More and more structured data in the form of RDF becomes available on the Web. This Web of Data bears enormous potential for supporting Web users in accomplishing more complex tasks and ultimately, will bring about new possibilities for commercial exploitation. Several initiatives have been started to deal with and to promote this Web of Data, noticeably the Linking Open Data (LOD) project [5] and the Billion Triple Challenge.1 Through these projects, a large number of datasets have become freely available. For instance, the amount of RDF triples promoted and maintained by the LOD project is in the order of tens of billions and keeps increasing rapidly. The popularity of these activities clearly indicates the strong interest in making use of the Web of Data—to improve the Web usage and to go beyond the applications possible so far.

However, the effective exploitation of the Web of Data bears a number of challenges:

• Usability: Typically, the user needs to specify a structured query (expressed in a formal language like SQL or SPARQL2) to search the Web of Data. However, the end user often does not know the query language and the underlying data schema.
• Scalability: As the amount of available data is ever growing, the ability to scale becomes essential.
• Imprecise information needs: The information needs expressed by the user might be imprecise. An effective search solution should be able to consider this aspect to deliver relevant results.
• Data change: The Web of Data is continuously changing. Thus, efficient mechanisms for index update at the Web scale are needed when data changes.

In this paper, we present Semplore (Semantic Explorer), a system for Web data search which addresses these challenges in an integrated way.

Usability is concerned with searching and exploring the Web of Data by end users. Formal query languages such as SPARQL are available to developers for accessing the data. However, Web users may prefer to use keyword search, a paradigm that has been made popular by major Web search engine providers. In this paper, we address this challenge by proposing a hybrid query formalism which has the power of structured queries and keeps the benefits of keyword queries (i.e., ease for use and expressing vague information


Keywords:
Scalable query processing
Inverted index
Faceted search
Search result ranking
Index update
needs). Furthermore, we elaborate on the use of faceted browsing, which when combined with hybrid querying, allows users to start with imprecise keywords, and to iteratively compose complex and expressive structured queries via operations on facets.

Scalability has always been a major challenge in Web search, especially when more complex queries are taken into account. While Information Retrieval (IR) approaches [16, 32] have proven to scale to the Web, the query capability supported by IR search engines is restricted to keywords. Clearly, this query capability is insufficient to fully exploit the expressive structure and semantics exhibited by the Web of Data. Database (DB) approaches [12, 28], on the other hand, provide expressive query capabilities. Some systems like the one reported in [1] also embrace the good usability of keyword search. However, scaling databases to Web data search remains an open challenge. In order to provide scalable hybrid query processing which combines keyword search with expressive structured query answering, we build upon IR technologies that have been successfully applied to the Web. In particular, we propose an extension of the inverted index to offer a scalable solution that can handle both text and structured data in the form of RDF triples.

To deal with the imprecision inherent in keyword parts of hybrid queries, we provide a ranking scheme that aims to return relevant results. This scheme takes both the imprecise keywords and the precise structure constraints of the hybrid query into account. In particular, scores are obtained for the imprecise matching of keywords against the data. The structure of the query and the underlying data is used to propagate and aggregate these scores. For efficiency and scalability, this ranking support is tightly embedded into the basic operations of hybrid query processing.

To support the change of the Web of Data, we propose a mechanism for incrementally updating the data indexes to reflect changes in the data. A block-based structure is proposed as our extension to the inverted index, which allows for incremental index update instead of re-building it from scratch when data changes.

Our experiments show that Semplore is an efficient and effective solution for hybrid query processing, and in particular, exhibits fast response time to compute facets and ranked results. Semplore also improves the state of the art in structured query processing. It outperforms the state-of-the-art triple stores w.r.t. unary tree-shaped queries. Overall, the results show that Semplore can serve as the basic infrastructure for querying and exploring the Web of Data.

This paper is organized as follows. Section 2 will give an overview of Semplore. In Section 3, we will investigate Semplore’s core features, its index structure, and in particular, its search capacities. Section 4 discusses extensions to these core features, i.e., ranking, faceted search, and index updates. In Section 5, we will present the evaluation results, before moving to the related work in Section 6. Conclusions along with future work will be presented in Section 7.

2. System overview

2.1. Hybrid query formalism

The Web of Data is essentially a collection of resource descriptions that contain relations, attributes, literal values, and text such as comments. Querying these descriptions in an effective way requires a mechanism that can handle both structured and textual information. To this end, hybrid query is a formalism that combines imprecise keyword search and precise structured query.

As a use case to illustrate the capability of such queries, let us assume that a user called Alice wants to find information about action films directed by Hong Kong directors and starring Chinese martial art actors.

Fig. 1 depicts the corresponding hybrid query as a directed labeled graph. The ability to use a combination of keywords like “action film” and precise relations like “starring” facilitates the user to express such a complex information need.

Semplore builds on the work of [22] that focuses on conjunctive queries, and extends this structured query capability with keywords to support hybrid queries. The key idea is to view keywords as “virtual” concepts called keyword concepts. A resource (will be referred to as an individual) will be regarded as an instance of a keyword concept $W$ if the textual content of any of its attributes contains all the keywords in $W$. The definition of a hybrid query $q$ over a knowledge base $K$ is an expression of the form $q(x) \leftarrow \exists \, \mathcal{F} \cdot \text{conj}(x, \mathcal{F})$

where $x$ is called the target variable, $\mathcal{F}$ is a vector of non-distinguished variables [22], and $\text{conj}(x, \mathcal{F})$ is a conjunction of terms in the form of $C(z), R(z_1, z_2)$, or $R(\neg z_1, z_2)$, and $z, z_1$, and $z_2$ are individuals in $K$ or variables in $x$ or $\mathcal{F}$. $R$ is a relation, $R^{-1}$ is its inverse relation, and $C$ (or $D$) represents a concept expression that is a Boolean combination of a normal concept $A$, a keyword concept $W$, and an enumerate class $[i]$ with individual $i$:

$$C, D := \top \mid \exists A W([i])C \land D \lor D \land \neg C$$

According to the above definition, the example query in Fig. 1 can be represented as: $q(x_0) \leftarrow \text{"action film"} (x_0) \land \text{starring}(x_0, x_1) \land \text{directedBy}(x_0, x_2) \land \text{ChineseActor}(x_1) \land \text{"martial"} (x_1) \land \text{HongKongFilmDirector}(x_2)$.

The answer to the query $q(x)$ w.r.t. $K$ is the set defined by $\{a \in O \mid K \vdash q[a/\alpha]\}$, where $O$ denotes the set of all individuals in $K$, and $q[a/\alpha]$ denotes the query $q$ with all occurrences of variable $x$ substituted by the individual $a$. Back to our example query, Heart of Dragon, a 1985 Hong Kong action film directed by Sammo Hung, is a relevant result.

In this paper, we focus on queries with tree-shaped structures. Also, queries are unary, i.e., it contains only a single target variable. That is, a list of resources will be returned as results just like current Web search engines do—as opposed to database results that might be tuples. These restrictions enable efficient query processing, while a large portion of typical information needs can still be expressed.

2.2. System components

The architecture of Semplore is depicted in Fig. 2. It is composed of three main components: (1) a front-end interface addressing the usability challenges by making various search functionalities of Semplore accessible to the user, (2) a search component in charge of retrieving and ranking the results as well as computing related facets, and (3) a component for indexing the Web of Data in a way that enables efficient search and index update.
Transforming the Web of Data into "Documents", "Fields", and "Terms" in IR.

Table 1
Transforming the Web of Data into "Documents", "Fields", and "Terms" in IR.

<table>
<thead>
<tr>
<th>Document</th>
<th>Field</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept C</td>
<td>subConOf</td>
<td>Super-concepts of C</td>
</tr>
<tr>
<td>Concept C</td>
<td>superConOf</td>
<td>Sub-concepts of C</td>
</tr>
<tr>
<td>Concept C</td>
<td>text</td>
<td>Tokens in textual properties of C</td>
</tr>
<tr>
<td>Relation R</td>
<td>subRelOf</td>
<td>Super-relations of R</td>
</tr>
<tr>
<td>Relation R</td>
<td>superRelOf</td>
<td>Sub-relations of R</td>
</tr>
<tr>
<td>Relation R</td>
<td>text</td>
<td>Tokens in textual properties of R</td>
</tr>
<tr>
<td>Individual i</td>
<td>type</td>
<td>Concepts that i belongs to</td>
</tr>
<tr>
<td>Individual i</td>
<td>subjOf</td>
<td>All relations R that (i, R, ?) is a triple in data</td>
</tr>
<tr>
<td>Individual i</td>
<td>objOf</td>
<td>All relations R that (? , R, i) is a triple in data</td>
</tr>
<tr>
<td>Individual i</td>
<td>text</td>
<td>Tokens in textual properties of i</td>
</tr>
</tbody>
</table>

3. Semplore core features

3.1. Data indexing

IR indexing is based on the concepts of documents, fields (e.g., title, abstract), and terms. Using inverted indexes, IR engines can efficiently retrieve documents for a given query consisting of a Boolean combination of (field, term) pairs. Current Web search engines have proven that this technique scales to the large quantity of documents on the Web.

Intuitively speaking, if we treat resources as documents and their associated concepts as terms, we can retrieve all individuals of a given concept by inputting the concept name as a query term. Extending this intuition, we can answer many types of semantic queries using IR engines, when we transform the Web of Data into documents, fields, and terms in a proper way, as shown by Table 1. After the transformation, the Web of Data is stored in inverted indexes of an IR engine. The data is then retrieved using the engine. In particular, the retrieval of resources based on sub-concepts, super-concepts, sub-relations, super-relations, type, and text are supported.

Besides these lookup functionalities, we propose an approach called PosIdx (position-based index) to index, retrieve, and ultimately join relation triples for supporting the unary tree-shape hybrid queries defined in Section 2.1. In an inverted index structure, each term is associated with a posting list of documents containing it. In addition, for each of these documents, there are a list of positions (stored in a position list) showing where the term appears in it. In the PosIdx method, relation names are indexed as terms and the subjects are stored as documents. The objects of a relation are stored in the position list. In other words, given the triple (s, R, o), the object o is stored in the position list of the term R in the document s. An example index structure is depicted in Fig. 3, where person2 and person3 are objects stored in the position list of subject film1 for relation directedBy. We treat directedBy as a term, film1 as a document, and person2 and person3 as the positions directedBy appears in the document. The index structure is symmetric since the objects of a relation represent the subjects of the inverse relation. That is, subjOf is treated as a field used for indexing instances of a relation R. Likewise, objOf is a field used for indexing instances of the relation R−1.

The physical storage of the inverted index and the retrieval of documents on top of it have been thoroughly studied in the IR community. Many optimized methods have been developed to improve the efficiency of index management, such as byte-aligned index compression [34] and self-indexing [29]. These optimizations can be leveraged for the efficient retrieval of relations. Furthermore, in the proposed PosIdx method, relation objects enjoy the benefit of spatial locality for fast access, since positions of a term are usually physically placed together and stored continuously in an inverted index used by modern IR engines.

3.2. Search functionalities

Based on the proposed index, Semplore reuses the IR engine’s merge-sort-based Boolean query evaluation method and extends it to answer unary tree-shaped hybrid queries. We will now introduce and explain some basic operations and their corresponding notations, before describing the query evaluation procedure implemented for Semplore.

3.2.1. 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 structures to the inverted index (e.g., self-indexing [29]), modern IR engines supply a very efficient stream reader for an AIS.

(1) Basic-retrieval b(f, t): given a field f and a term t, b(f, t) retrieves the corresponding posting list from the inverted index. The output of this operation is an AIS. For example, under the struc-
ture described in Table 1, \( b(\text{type, ChineseActor}) \) will retrieve all individuals of the \( \text{ChineseActor} \) concept as an AIS.

2. Merge-sort \( m(S_1, o_1, S_2, o_2) \): \( S_1 \) and \( S_2 \) are two AISs and \( o \) is a binary operator which can be \( \land \), \( \lor \) or \( \neg \). Merge-sort computes \( S_1 \oplus S_2 \) 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. Mass-union \( u(S, R) \): Given a set of subjects \( S \) and a relation \( R \), this operation returns the union of object sets \( \{ o(s, R, o) \} \) over every subject \( s \) in \( S \), i.e., \( u(S, R) = \bigcup_{s \in S} \{ o(s, R, o) \} \). It also sorts the union set to ensure the returned result is an AIS.

3.2.2. Concept expression evaluation

As defined in Section 2.1, a concept expression \( C \) is a Boolean combination of normal concepts, keyword concepts, and enumerated classes. Evaluating concept expressions is an important part of hybrid query processing. For this, we introduce an operation for concept expression evaluation that is denoted as \( \lambda(C) \). This operation takes a concept expression \( C \) as input and outputs an AIS containing the IDs of all individuals of \( C \). It can be implemented using the operations for basic-retrieval combined with nested merge-sort. For example, given the concept expression \( C = \text{ChineseActor} \cap \text{"martial"} \), \( \lambda(C) \) is performed via two basic-retrieval operations and one merge-sort operation: \( m \left( b(\text{type, ChineseActor}), \land, b(\text{text, "martial"}) \right) \).

3.2.3. Relation expansion

The \( \lambda(C) \) operation concerns the evaluation of the concept expression \( C \) on query vertexes. To evaluate a tree-shaped query, we need another additional operation for evaluating query edges. The relation expansion operation \( \oplus(s, R, S_2) \) is defined for this purpose. The input to this operation is a relation \( R \) and two AISs \( S_1 \) and \( S_2 \) that contain individual IDs for relation subjects and objects respectively. The operation computes the set \( \{ y \mid \exists x : x \in S_1 \land (x, R, y) \land y \in S_2 \} \) and returns it as an AIS. For example, \( \oplus(\lambda(\text{"action film"}), \text{directedBy}, \lambda(\text{HongKongFilmDirector})) \) is used to find all Hong Kong film directors who have directed some action film. This DB-like operation (i.e., join) is not directly supported by current IR engines. We propose to evaluate it in four steps through a combination of the basic operations discussed above.

First, we compute the subjects that have \( R \) as relation, i.e., \( S = m(S_1, \land, b(\text{subjOf}, R)) \). Second, we find the set of objects for each subject \( s \in S \), i.e., \( g(s, R) = \{ o(s, R, o) \} \). Third, we union the object sets for these subjects and sort the result set to obtain a new AIS \( S_0 \) where \( S_0 = u(S, R) = \bigcup_{s \in S} g(s, R) \). Finally, we do a merge-sort \( m(S_0, \land, S_2) \) to obtain the final result. Fig. 4 illustrates these four steps to calculate \( \oplus(\lambda(\text{"action film"}), \text{directedBy}, \lambda(\text{HongKongFilmDirector})) \).

When the number of subjects in \( S \) is large, the mass-union \( u(S, R) \) operation becomes expensive, as it requires to union and sort on a large number of sets stored on disk. In Fig. 4, these sets are represented by the orange segments in term \( \text{directedBy} \)'s position list. In particular, retrieving all the objects using a streaming merge-sort on these sets from disk, might lead to prohibitive I/O as it requires a large number of back-and-forth disk seeks. One way to save I/O cost is to select a subset of all the sets that can still cover all the objects. However, this is the Set-Cover problem which is NP-hard.

We use a simple yet effective approach for the mass-union operation: for every subject \( s \), the set \( \{ o(s, R, o) \} \) is retrieved and a bit vector is used to track the union of the results. 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 \( O_R = \{ o(s) \mid (s, R, o) \} \) and \( N = |O_R| \). A list of objects \( o_1, o_2, \ldots , o_N \) are returned after sorting the set \( O_R \) on object IDs in ascending order. The sequence number of object \( o_i \) is \( i \) under relation \( R \). Thus, a bit vector of a limited size is allocated to track which object is in the result of the relation expansion. Note that the size of the bit vector, i.e., \( N = |O_R| \), can be directly obtained from the inverted index as the document frequency of the term \( R \), without any computation at run time. Retrieving the object sets of all subjects can be implemented very efficiently using a sequential scan on the position list during merge-sort, with the help of self-indexing [29]. In this way, the effectiveness can be guaranteed by the fact that sequential accesses are less expensive than random accesses, which reduces the I/O cost.

The worst case time complexity of this operation is linear to the number of objects that have to be retrieved for these subjects using merge-sort, since the object IDs are sorted in the position list. The operation terminates when all the possible results are found. Thus, the execution time will not necessarily increase with the

1 If the relation is \( R' \), replace \( \text{subjOf} \) with \( \text{objOf} \). Similarly, in the following steps \( g(o, R') \) is used instead of \( g(s, R) \).
size of the valid subjects $S = m(S_1 \cap, b(\text{subj}) \exists t, R)$. In our experiments, we find that the time even decreases when the size of subjects $|S|$ exceeds a threshold. This is because through one single disk I/O operation, more $g(s, R)$ sets (the orange blocks in Fig. 4) can be retrieved due to the increased locality of these sets when $|S|$ becomes large. The space requirement of the bit vector is linear to the number $N$ of objects associated with a relation, which may be quite large. But in practice, 256 MB memory can already hold a bit vector for $N$ as large as 1 billion, and this memory can be reused for multiple executions of the mass-union operation.

3.2.4. Evaluation algorithm

Based on the operations of concept expression evaluation and relation expansion, Algorithm 1 shows how a unary tree-shaped hybrid query is evaluated. The algorithm traverses the query tree in the depth-first search (DFS) order. It evaluates the concept expression of each vertex when moving forward, and uses the results of the children to constrain the results of the parent when moving backward. It terminates in $2 \times |E|$ steps, each of which is either a $\lambda(C)$ operation or a $\exists(S_1, R, S_2)$ operation.

In fact, traversing the query tree in the DFS order is a kind of planning strategy for query execution. From our experimental results presented later in Section 5.2, Semplore is usually able to quickly deliver answers for those complex queries. We have now considered finer grained query optimization technologies which make use of the characteristics of RDF data to provide optimal query execution trees. In a nutshell, we build additional indexes to record the statistical distribution for the subject part, the predicate part, the object part, or their combinations for cost estimation. We then use a dynamic programming algorithm to find the (near) optimal execution tree as fast as possible. Since it is still in its early stage and only some preliminary results are available, we do not integrate it in Semplore.

Algorithm 1. Query evaluation algorithm

1. **Input**: A unary tree-shaped hybrid query $Q(t)$ with graph $G = (V, E)$ and target variable $t$ on the vertex $V \in V$: Each vertex $v \in V$ has a concept expression $C_v$ 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 $R_{u,v}^o$.)
2. **Output**: An AIS containing the IDs of individuals in the answer set of $Q(t)$.
3. For $\forall u \in V$, set checked[$u$] = false and $S_u$ = null;
4. DFS($u$);
5. return $S_u$;

**Procedure. DFS**($u$):

1. checked[$u$] = true;
2. $S_u$ = $\lambda(C_u)$;
3. for each vertex $v$ such that $(u, v) \in E$ or $(v, u) \in E$ do
4. if checked[$v$] = true then
5. Continue;
6. end if
7. DFS($v$);
8. if $(v, u) \in E$ then
9. $S_u$ = $\bowtie(S_1, R_{u,v}, S_v)$;
10. end else
11. $S_u$ = $\bowtie(S_1, R_{v,u}, S_v)$;
12. end if
13. end for

Taking Fig. 1 as an example with $x_0$ as the target variable, we first reach $x_1$ via $x_0$, and compute the results $S_{x_0}$ and $S_{x_1}$ for these two vertices. Then $S_{x_0} = \bowtie(S_{x_2}, \text{starring}^{-1}, S_{x_1})$ is computed when we move backward from $x_1$ to $x_0$. Similarly, $x_2$ is traversed and the results of the root is updated by $S_{x_0} = \bowtie(S_{x_2}, \text{directedBy}^{-1}, S_{x_0})$, which is the final answer.

4. Semplore extension

4.1. Relation-based ranking

For effective search, it is necessary to rank the results according to their relevance to a hybrid query. One main measure of relevance is the score obtained from matching keywords against data, which is computed for a result of the basic-retrieval operation supported by the underlying IR engine. Other metrics representing the quality or popularity (e.g., as computed via PageRank [6] or simple frequency-based TF/IDF-like approaches [2]) can be considered as additional factors that determine the relevance score of a resource, i.e., a graph element. In this paper, we focus on the scores returned by IR engines as the main measure for ranking.

Note that according to our query model, only the bindings to the query root vertex, i.e., the target variable are returned to the user. Query constraints such as keywords, however, might be applied to other resources representing bindings to other query vertexes, i.e., non-distinguished query variables. Intuitively, the scores of these resources should have an impact on the ranking of the final results because they are related, i.e., they are connected over the structure specified in the query. Based on this notion, we formulate the following two principles to guide the design of the ranking mechanism.

- **Quality propagation.** The score associated with an element can be seen as a measure of quality. In quality propagation, this score is updated to reflect also the quality of its neighbor elements. As a result of quality propagation, elements are assigned a higher rank if they are connected with higher quality neighbors. For instance, when looking up the successors of US presidents who match the keyword “war”, J.F. Kennedy should be ranked high because his predecessor Eisenhower is tightly related to “war”.
- **Quantity aggregation.** In addition to the quality, the number of neighbors is taken into account. As a result, elements are ranked higher if they have a larger number of neighbors. In the query “Find institutions that Turing Award winners work at”, CMU, UC Berkeley, and IBM are the top 3 institutions because they have the largest number of Turing Award winners.

Moreover, monotonicity should be respected by the ranking scheme such that the higher the quality of its neighbor and the higher the number of its neighbors respectively, the higher the rank of an element. We will now elaborate a ranking scheme based on these principles.

First, we extend the AIS data structure with scores to obtain a Scored Ascending Integer Stream (SAIS). Given a SAIS $S' = \{d_1, 0.4 \}, \{d_1, 0.8 \}, \{d_4, 0.6 \}, \ldots$, we represent the score of an element $d_1$ by $S'[d_1] = 0.4$ and treat the score as a probability value, i.e., we adopt the probabilistic semantics for scores attached to elements. Then, we generalize the $b$, $m$, and $u$ operations discussed previously to $b'(f, t)$, $m'(S_1, r, S_2)$, and $u'(S', R)$ for SAISs:

For basic-retrieval $b'(f, t)$, we compute the score of an item as follows:

$$b'(f, t)[d] = \begin{cases} \text{RVS}(t, d) & \text{if } f = \text{text} \\ 1 & \text{otherwise} \end{cases}$$

where $\text{RVS}(t, d) \in [0, 1]$ represents the IR relevance of a document $d$ (a resource in our approach) with respect to term $t$ according to the TF-IDF principle. Note that $t$ can be either a textual token or an ontological term. If $t$ describes information about some concept or property, $b'$ just returns 1 when $d$ matches such a constraint. This relevance score is returned by the IR engine as a result of keyword-based retrieval.
For merge-sort \( m(S_1 \cap S_2)[d] = S_1[d] \cdot S_2[d] \)

For mass-union \( u(S', R) \), we compute the score of an item by using the following equation:

\[
u(S', R)[o] = 1 - \prod_{s \in S' \cap (s, R, o)} (1 - S'[s])
\]

Note that both merge-sort and mass-union are based on the assumption that the scores of the elements are probabilistically independent. \( m(S_1 \cap S_2)[d] \) calculates the joint probability of \( S'_1[d] \) and \( S'_2[d] \) while \( u(S', R)[o] \) propagates and aggregates all contributions of \( s \in S' \) to the score of \( o \) through a relation \( R \).

By applying the scoring functions to the three operations, we associate the results of a distribution query \( v \) with scores, i.e., element \( d \) representing a binding of \( v \) is scored by

\[
res^v[d] = b(\text{text}, W(v))[d] \cdot b(\text{type}, C(v))[d] \cdot \prod u(res'^v[u]).
\]

\[
R(u, v)[d]
\]

Note that this formula combines not only the scores for the keyword and the concept constraints on \( v \) but also those from neighbors connected through the different relation edges.

Furthermore, as the scores are combined with the basic operations, ranking is seamlessly integrated into query processing. During query processing, these scores are propagated along the edges from leaf vertexes to the root vertex, i.e., to the node corresponding to the target variable.

### 4.2 Faceted search and browsing

Faceted browsing (or synonymously, faceted search) \([19]\) has been recognized as an intuitive yet effective way for the user to express complex information needs. Starting from a keyword query, the user gets the ranked results as well as the corresponding facets. These facets enable the user to better understand the results and further refine the current search by adding facet constraints to the query. In the context of searching the Web of Data, we leverage concepts and relations as facets to facilitate users to construct hybrid queries. In particular, given a list of results, i.e., resources represent bindings to the target variable, the corresponding facets are all concepts and relations associated with these resources.

To support hybrid query formulation in such an interactive way, there remains the problem of efficient facet calculation and ranking. We will now present an approach for computing and ranking facets based on the counts of results that correspond to them.

We abstract this problem as the problem of distribution query answering. Essentially, a distribution query is a special tree-shaped query, i.e., a single-atom query, which asks for the associated concepts and relations (both incoming and outgoing) of a result set \( D \) (e.g., bindings to \( x_i \) as shown by Fig. 5). In particular, given a set \( D \), the concepts and relations where elements in \( D \) appear as individuals, objects, and subjects respectively, are computed. Furthermore, a count is returned for every concept \( C \) indicating how many elements in \( D \) belong to \( C \), i.e., \( \text{count}(D, C) = |\{d | d \in D \wedge (d, \text{type}, C)\}| \). Similarly, two counts are returned for each relation \( R \) indicating how many elements in \( D \) are the subjects and objects of \( R \) respectively, i.e., \( \text{count}_s(D, R) = |\{s | s \in D \wedge \exists o : (s, R, o)\}| \) and \( \text{count}_o(D, R) = |\{o | o \in D \wedge \exists s : (s, R, o)\}| \).

More precisely, we formulate these distribution queries using three preserved relations \( \text{type}, \text{subjOf}, \) and \( \text{objOf} \) as predicates and transform them into the mass-union operations \( u'(\text{type}), u'(\text{subjOf}), \) and \( u'(\text{objOf}) \) respectively. These queries are further answered based on an additional PostIdx index using the subjOf field only. \( \text{type}, \text{subjOf}, \text{objOf} \) are treated as terms; individuals are regarded as documents; and their associated concepts and relations are stored in the corresponding position lists. For obtaining the counts, we use a special scoring function, which calculates the number of results correspond to a given facet (a concept or a relation):

\[
u'(S', R)[o] = \sum_{s \in S' \cap (s, R, o)} 1
\]

Once the results of a distribution query are returned, we sort the concepts and relations by their counts in the descending order and pick out the top- \( k \) as the suggested facets.

### 4.3 Index update

Traditional optimizations for document index update \([36,25]\) cannot be directly applied to the support of updates for relation triples: when new triples arrive, their subjects and objects are inserted into the posting lists of the corresponding relations and inverse relations, which will move the original positions of some individuals behind. It might be a heavy cost since inserting one individual may sometimes leads to reconstructing the whole posting list. We present here a block-based index structure which is able to reduce the update cost. Our proposed index update mechanism is designed for the incremental crawling scenario, which mainly handles the insertion of triples. It extends the index structure of Semplore to support efficient incremental updates so that we can handle the ever growing Web of Data.

#### 4.3.1 Block-based index structure

To minimize changes in the position lists when inserting triples, we split posting lists into blocks. Taking the first individual in a block as a landmark, the local position of another element in this block is defined by a pair \( \langle \text{LandmarkID}, \text{offset} \rangle \), where the former identifies the landmark and the latter is the offset of the element’s position relative to that of the landmark. An auxiliary table is used to store the real position of the landmarks. The real position of the element is obtained by adding the offset to the real position of its block’s landmark. In Fig. 6, for example, in the posting list of relation \( \text{directedBy} \), individuals \( \text{film}_1 \) and \( \text{film}_2 \) are the landmarks of \( \text{Block}_1 \) and \( \text{Block}_2 \) respectively. The local position of \( \text{film}_5 \) is represented by...
(lm1, 1) where lm1 is the landmark ID of Block1 and 1 represents
the offset. Its real position can be obtained by summing the real
position of lm1 and the offset value 1.

Note that the block-based index structure is only used for the
storage of relation triples. For concepts, relations or concept indi-
viduals, which are only stored in posting lists, the index structure
is the same as that is defined in Semplore Core Features.

4.3.2. Single update operation

For a single triple insertion (see Algorithm 2), we first insert, if
needed, the subject into the relation’s posting list and the object
into that of the inverse relation. We can now add the local position
of the subject (object) to the position list of the object (subject) in
the posting list of the reverse relation (relation) respectively.

Algorithm 2. Single update algorithm

1: Input: A triple (s, R, o) to be inserted
2: if s ∈ Posting(R) then
3: insert(s, R);
4: end if
5: if o ∈ Posting(R) then
6: insert(o, R);
7: end if
8: Add LocalPosition(s, R) to PositionList(o, R);
9: Add LocalPosition(R, s) to PositionList(o, R);
10: Update the landmark table;

Procedure. insert(s, R)

1: Find block B that s should be inserted into;
2: for each instance i ∈ B ∧ i > s do
3: for each (p, p), (p, p) ∈ PositionList(i, R) do
4: o = skipTo((p, p), Position(R));
5: Find (n, n) in PositionList(o, R) that skipTo(n, n, Position(R)) == i;
6: ns = no + 1;
7: end for
8: end for
9: Add s to Posting(R);
10: Update the landmark table;

The insertion of an individual s into the posting list of a relation
R is done by insert(s, R). For each individual i following the insert
position of s in a block, the corresponding object o is found in the
inverse relation’s posting list. skipTo is used to find the appropriate
position of an element to be inserted in a posting list given its local
position. After s’s local position is found in the position list of o from
R’s posting list, its offset shall increase by one. Finally, s is inserted
and the landmarks of all the blocks following the current one are updated
in the landmark table.

Fig. 7 shows what happens when a single triple
(film2, directedBy, person2) is inserted into the index. Since
the subject film2 does not exist in the posting list of relation
directedBy, it is inserted into Block1 of the relation’s posting list. All
the offset values of the subsequence individuals (i.e., film2’s offset value)
in this block shall be increased by one. The local position of
film2 is updated and stored in the position list of its corresponding
objects. By reading the position list of film2 in the relation’s posting
list, we can easily get the corresponding objects (i.e., person2) and
skip to their positions in the posting list of directedBy. Then we
can update the position of film2 found in person2’s position list
through increasing the offset values by one. Similarly, the object
person2 is inserted into Block2 of directedBy’s posting list and all
the local positions of subsequent individuals in this block have to
be updated. As a result, their local positions stored in the position
list of the corresponding subjects in directedBy’s posting list are
changed.

Deleting is easier: the local positions of both the subject and the
object are deleted in each other’s position lists. Due to the deletion
cost of an individual in the posting list, we do not delete an
individual with an empty position list in the prospect of a further
insertion.

4.3.3. Batch update operation

With a batch update, all the individuals belonging to the same
block are inserted into the posting list in a single operation, as
explained by Algorithm 3. Once the new individuals in the posting
list are inserted, the local position of each original individual in
the block that is behind the minimum inserted individual should be
moved backward. These local positions are stored in the posting lists
of the corresponding objects in the inverse relation’s posting list.
The offset value of such an individual is updated according to the
number of individuals that will be inserted in front of it. The
batch insert operation is more efficient since it reduces the num-
ber of position updates when inserting individuals belonging to
the same block. After the batch update, a block whose size exceeds
the threshold will be splitted into two smaller blocks.

Algorithm 3. Batch update algorithm

1: Input: Triples (s, R, o1), (s, R, o2) ... (s, R, on) to be inserted
2: for each block Bi ∈ Posting(R) do
3: Ssub = {s1, S1 | Posting(R) ∧ s1 ≤ Landmark(Bi) ∧ s1 < Landmark(Bi+1)};
4: batchInsert(Ssub, Bi, R);
5: end for
6: for each block Bi ∈ Posting(R) do
7: Sobj = {s1, S1 | Posting(R) ∧ s1 ≥ Landmark(Bi) ∧ s1 < Landmark(Bi+1)};
8: batchInsert(Sobj, Bi, R);
9: end for
10: for each (s, R, o) do
11: Add LocalPosition(R, s) to PositionList(o, R);
12: Add LocalPosition(R, o) to PositionList(s, R);
13: end for

Procedure. batchInsert(S, B, R)

1: for each instance i ∈ B ∧ i ≥ min(S) do
2: for each (p, p), (p, p) ∈ PositionList(i, R) do
3: o = skipTo((p, p), Position(R));
4: Find (n, n) in PositionList(o, R) that skipTo(n, n, Position(R)) == i;
5: ns = no + 1;  \{\{s ∈ S ∧ s < i\} \}
6: end for
7: end for
8: Add each s ∈ S to Posting(R);
9: Update the landmark table;
4.3.4. Performance analysis

To ensure the efficiency of index update, the block size should be restricted in a certain range. Given blocks with larger sizes, a larger number of individuals are affected in terms of their local positions during insertions. Thus, Thus, a block with large size needs a large number of required update operations to be performed on the position list. On the other hand, blocks with too small sizes will decrease the efficiency of both index update and query processing. This is because the real positions of the individuals are computed by looking up the landmark table. Small blocks will increase the size of the landmark table, and thus slow down the lookup operation. Small blocks also produce many fragments on the disk, which has a negative effect on disk access time. For a properly chosen block size, the landmark table is usually very small so that it can be completely loaded in the main memory during index update and query processing. The impact of the block size is further discussed and demonstrated in our previous work [26].

The performance of query evaluation on top of the block-based index is not significantly different from that of the procedure elaborated in Section 3.1. For concept individuals, which are stored in a traditional inverted index without blocks, the IR engine provides fast processing time. For relation triples, the main difference is that the real position of each individual in the posting list has to be computed by seeking the real position of the landmark in the landmark table and adding the individual’s offset. In the landmark table, all landmarks in a certain relation’s posting list are sorted by their real positions. Using binary search, the time complexity of seeking the real position of a landmark is \(O(\log (L/B))\), where \(L\) is the length of the posting list and \(B\) is the average block size.

5. Evaluation and discussion

We built Semplore on top of Lucene 2.3.4 The prototype was developed using Java 1.6. We compared Semplore with the prevalent state-of-the-art triple stores: the native RDF repository RDF-3X [30] and the DB-based system SOR [28] built on IBM DB2 V9.1. The block size of Semplore’s index was set to 1000. Both synthetic and real datasets, LUBM [17] and DBpedia5, were used for the experiments. We created a dataset for 1000 universities LUBM(1000) with more than 137 million triples to assess the scalability of the system. All experiments were conducted on a 64-bit PC with a 2.66 GHz Intel Dual Core processor and 4 GB memory.

5.1. Index building and update

We first compared the time to build the data indexes from scratch for all three systems on the LUBM datasets. We varied the number of universities from 20 to 100 (20 universities as a unit step) to obtain five datasets with different sizes, ranging from 2.8 million triples to 14 million triples. As shown by Fig. 8(a), we can see that with the increase in data volume, the time for loading and indexing RDF triples needed by Semplore is much better than that needed by SOR. As Semplore used additional indexes for textual indexing RDF triples needed by Semplore is much better than that of RDF-3X. Note that RDF-3X was developed using C++ while the other systems were implemented using Java. RDF-3X uses six B+-trees to index all combinations of \(b\), \(p\), and \(o\). We tuned SOR to its best performance with the help of IBM DB2 Design Advisor. Each query was repeated 10 times to calculate the average time for all three systems. Before each run, the program copied an arbitrary big file having the same size as the main memory to clean the operating system cache. As for SOR, we did not only record the query time

update mode, Semplore achieved significant improvements compared with approaches that build indexes from scratch. It is even better than that of RDF-3X, especially for large datasets. The main reason is that RDF-3X does not support incremental update and has to re-index for the data change.

Table 2 shows the index size and index building time of the three systems on DBpedia and LUBM(1000). Among all the three systems, SOR requires the largest index space as the size of the stored triples is also counted. Both the index size and the index building time of Semplore and RDF-3X are comparable while SOR takes much more time. The reason is that for fast query processing, SOR makes use of additional indexes which have to be built on such large scale data. Note that RDF-3X has to map the triples into an address space comparable to the scale of the imported data so that it fails to perform data loading and indexing on a 32-bit machine because the size of the indexed triples exceeds its address space. This experimental result suggests that using IR indexing technique to manage RDF data is applicable to the Web of Data, i.e., Semplore scales no worse and in some cases, is even better than B+-tree-based RDF stores (e.g., RDF-3X and SOR).

5.2. Structured query capability

Structured queries in real world scenarios are of different complexities and represent different access patterns. We aim to capture these differences by proposing queries constructed from various combinations of query variables, shapes, and lengths. In particular, we have defined 15 conjunctive queries for each dataset, resulting in a total of 30 queries. The 15 queries can be decomposed into five classes, each representing a particular query shape. Each class comprises three queries that vary in length and in the number of variables. We will discuss the query classes with some examples and refer interested readers to http://apex.sjtu.edu.cn/apex wiki/Semplore_QS for the whole set of queries:

- **Single-atom Queries** (QC1) consist of exactly a single query atom. Query 1 on DBpedia (\(Q_{DBpedia}\)), for instance, simply asks for all living people.
- **Path Queries** (QC2) consist of several connected query atoms that together form a path. The example query \(Q_{LUBM}5\) retrieves all students taking courses that are lectured by full professors.
- **Star Queries** (QC3) are composed of more than two single-atom and path queries. These parts share exactly one common node, i.e., the center node of the star. The query example \(Q_{LUBM}9\) retrieves the students who take a course taught by a specified full professor, have published a given publication, and belong to a certain organization having a particular telephone number.
- **Entity Queries** (QC4) are formed by several single-atom queries that share exactly one common node. Intuitively speaking, this node stands for an entity, and edges represent entity properties and attributes respectively. \(Q_{DBpedia}\) asks for locations with a specified population and area.
- **Tree-shaped Queries** (QC5) consist of nodes and edges that form a tree. \(Q_{LUBM}15\) for instance, retrieves students from Department 3 having an advisor who has authored *Publication0* and has a telephone number “xxx-xxx-xxxxx”.

Note that RDF-3X was developed using C++ while the other systems were implemented using Java. RDF-3X uses six B+-trees to index all combinations of \(b\), \(p\), and \(o\). We tuned SOR to its best performance with the help of IBM DB2 Design Advisor. Each query was repeated 10 times to calculate the average time for all three systems. Before each run, the program copied an arbitrary big file having the same size as the main memory to clean the operating system cache. As for SOR, we did not only record the query time.

Fig. 8. (a) Data loading and indexing time. (b) Index update performance.

Table 2
Index size and indexing time on DBpedia and LUBM(1000).

<table>
<thead>
<tr>
<th></th>
<th>DBpedia</th>
<th>LUBM (1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index space (MB)</td>
<td>Indexing time (s)</td>
</tr>
<tr>
<td>Semplore</td>
<td>5,546</td>
<td>33,688</td>
</tr>
<tr>
<td>SOR</td>
<td>39,797</td>
<td>65,595</td>
</tr>
<tr>
<td>RDF-3X</td>
<td>5,960</td>
<td>21,841</td>
</tr>
</tbody>
</table>

Table 3
Response time for different query classes on three systems (ms).

<table>
<thead>
<tr>
<th>Query</th>
<th>LUBM(1000)</th>
<th>DBpedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Semplore</td>
<td>RDF-3X</td>
</tr>
<tr>
<td>QC1</td>
<td>1485</td>
<td>2409</td>
</tr>
<tr>
<td>QC2</td>
<td>277</td>
<td>999</td>
</tr>
<tr>
<td>QC3</td>
<td>619</td>
<td>1161</td>
</tr>
<tr>
<td>QC4</td>
<td>556</td>
<td>1163</td>
</tr>
<tr>
<td>QC5</td>
<td>463</td>
<td>844</td>
</tr>
<tr>
<td>QC1</td>
<td>550</td>
<td>646</td>
</tr>
<tr>
<td>QC2</td>
<td>36</td>
<td>419</td>
</tr>
<tr>
<td>QC3</td>
<td>107</td>
<td>52</td>
</tr>
<tr>
<td>QC4</td>
<td>173</td>
<td>299</td>
</tr>
<tr>
<td>QC5</td>
<td>105</td>
<td>907</td>
</tr>
</tbody>
</table>

Table 3 shows the response time of the three systems for all the five classes of queries. Thanks to spatial locality when using the PosIdx method, Semplore is very scalable for all query types. In particular, it outperforms RDF-3X in most cases although the latter employs several compression and optimization techniques. Semplore’s performance is comparable to that of SOR with cache and is one or two orders of magnitude faster than SOR when performing cold start. Benefitted from a well-designed cache system, the response time of SOR on all query sets is significantly shortened. For QC3, SOR even does better consistently on both datasets. When comparing the comparative performance of these systems between LUBM and DBpedia, we find that the performance of SOR is relatively stable on both datasets and even better on the artificial data set while the response time of Semplore and RDF-3X always tends to be much shorter on DBpedia. It indicates that DB-based systems are more robust and they always build sufficient indexes to speed up query processing for all cases while Semplore and RDF-3X simplify their designs and implement efficient query processing based on assumptions on the data distribution, which seems to fit better on the real-world dataset. With tree-shaped queries, although Semplore needs more relation expansion operations, it manages to return answers within one second and the performance gains are obvious compared with the other two systems. One important reason is that Semplore’s relation expansions are lightweight compared with complex nested table joins. This advantage comes from Semplore’s design on the trade-off between query capabil-

Fig. 9. (a) Precision and recall for QS3. (b) Precision performance for QS4.
ity and scalability. The comparison results indicate that IR-based approaches are promising for querying Semantic Web data.

5.3. Hybrid search

To test the effectiveness and efficiency of Semplore on hybrid queries, we created four query sets, i.e., QS1 through QS4, and tested query response time on all of them. QS3 and QS4 were further used for effectiveness test:

- **QS1.** It consists of 500 queries, which come from “http://dbpedia.org/page/list_of_...”. They are used to find a list of individuals for a given keyword constraint. For instance, we can use “http://dbpedia.org/page/action_films” to find action films. This query set is to show the capability of Semplore to handle simple keyword queries.

- **QS2.** It consists of 1,242 queries concerning subjects and objects of 621 relations in DBpedia. For example, queries corresponding to subjects and objects of relation “directedBy” aim at finding films and directors respectively. This query set is to show the capability of Semplore to handle simple structured queries.

- **QS3.** It is composed of 287 queries using the content of “http://dbpedia.org/page/List_of_film_director_and_actor_collaborations” and “http://dbpedia.org/page/List_of_film_director_and_composer_collaborations”, which involves three relations: “directedBy”, “actedIn” and “composedBy”. This query set is to evaluate the precision and recall of our proposed ranking scheme on these hybrid queries using the ground truth set collected from these DBpedia lists.

- **QS4.** We manually created 20 queries according to the questions provided by 10 users. These queries were thought to be answerable using DBpedia. They range from simple to complex tree-shaped conjunctive queries that consist of up to 5 predicates. An example is “Find institutions that Turing Award winners work at”. Note that “Turing Award winners” is not explicitly given in the structured data but is only contained as text in some entity descriptions. This query set is to show the capability of Semplore to handle complex hybrid queries. You can find the detailed information of queries in QS4 from the same URL for those structured query sets.

5.3.1. Impact of ranking

We compared the effectiveness of the ranked results produced by Semplore against Lucene and SOR with Text Extender6 (abbreviated as SOR hereafter). We indexed entities as virtual documents with their descriptions (i.e., labels and literals) in the text field using Lucene, which is similar to what has been proposed by most Semantic Web search engines. In SOR, both the RDF data and entity descriptions were stored. In particular, we created one additional table to store textual descriptions for entities. Since QS4 lacked the ground truth, we conducted a survey with 20 participants from our laboratory to obtain a reference standard on precision for this query set. Each participant was asked to rate the top 10 results for some queries produced by the three systems. Each query was rated by at least 3 persons. Only results receiving a unanimous judgement of “relevant” are used as the ground truth. We used the standard IR metric P@n [2] to compute the precision for the top n results returned by the systems. Note that the queries in QS4 were submitted by the users arbitrarily and no ground truth was available, so it was impossible to calculate the recall for QS4.

The precision and recall performance of Semplore, SOR, and Lucene for QS3 is shown by Fig. 9(a). Thanks to the rich metadata especially relations in DBpedia and our effective ranking scheme, Semplore achieved much higher precision and recall than SOR and Lucene. Fig. 9(b) shows that search on both structured and textual data can greatly improve precision. In particular, the P@10 of SOR is about 20% higher than that of Lucene for the query “Find institutions that Turing Award winners work at”. To answer this query, SOR can exploit structured data about institutions and persons in DBpedia and combine it with text about Turing Award winners. In contrast, Lucene relies entirely on textual descriptions and returned results simply contain the terms “institution”, “Turing Award winner” and “work”. However, the ranking mechanism in SOR considers only the scores for the keyword search on the text column. Semplore goes a step further to consider propagation and aggregation of scores among elements. With respect to the example query, institutes are ranked higher by Semplore when they are related with a larger number of persons with textual descriptions that strongly match “Turing Award winners”.

5.3.2. Results and facets calculation time

To test Semplore’s efficiency, we ran all query sets and listed the response time when computing results and facet information. Since we need to calculate concept facets and two kinds of relation facets (with the answers being their subjects and objects respectively) for the current returned results, it is expected to take three times response time for returning facets compared with that of returning results if we assume that the cost of mass-union operations and the data involved are approximately the same. According to the results shown in Table 4, we can notice that Semplore achieves its query task in less than half a second on average. QS1 is the least expressive query set: it only contains one concept constraint, which leads to fast query answering. However, as it will usually return long lists of diverse results, it causes many facets to be calculated. For QS4, the most expressive query set, more join operations are required than for other query sets. It takes more time to return results, while the time for facet calculation is usually short due to the small number of results. It reveals that the gap between the result and the facet computation time is becoming smaller with the growth of query expressivity. To reduce the time for facet calculation, only the top-k results are considered when computing their associated facets. We plan to carry out more experiments to see the performance difference in terms of both efficiency and effectiveness by using different ranking functions and optimization technologies.

The evaluation results discussed above indicate that Semplore exhibits affordable time and space requirement for building indexes, supports incremental index update to handle the data change, offers acceptable response time for processing the structured queries, returns relevant results effectively according to the input hybrid queries, and calculates facets efficiently. In brief, the overall system can scale to a realistic search environment for the Web of Data.

6. Related work

There exist several dimensions of related work. We structure our discussion along the presentation of our contributions: (1) the integration of DB and IR for scalable hybrid query processing, (2) a scheme of ranking results of hybrid queries (3) faceted search and browsing functionalities, and (4) an efficient index update mechanism. Compared with our previous work [42], we further
investigated the ranking, faceted search, and index update issues. Moreover, we carried out more comprehensive evaluation to show that our proposed IR approach scales to large amounts of Web data.

6.1. DB and IR integration

Existing work on querying and searching Semantic Web data can be roughly divided into two categories: IR-based approaches and DB-based approaches. Prominent examples of the first category include the various Semantic Web search engines such as Falcons [10], Sindice [37], Swoogle [14] and Watson [33]. These engines crawl Semantic Web documents and use inverted indexes to provide lookup functionalities for them. Keywords submitted by the user are matched against the indexed resources. Then the relevant ones are ranked according to the matching scores returned by the IR engine. These engines take a pure IR approach and do not support structured queries on the Web of Data.

Most DB-based systems such as OracleRDF [12], Virtuoso [15] and SOR [28] rely on the underlying database engine for indexing and querying Semantic Web data. These DB-based approaches have no inherent support for keyword search. In order to support this kind of search, these systems employ a separate IR engine with special columns used for this purpose.

RDF-3X [30] is a native repository specially designed and built for RDF, which uses indexed index structures (i.e., B+ trees). YARS2 [18] is another native RDF store where index structures and query processing algorithms are designed from scratch and optimized for RDF processing. The novelty of the approach proposed by YARS2 lies in the use of multiple indexes to cover different access patterns. In Semplore, the inverted indexes support quick lookups for subject, object, and predicate of a triple pattern. In YARS2, the closest coverage of access patterns is achieved for the management of quads in the form of (s, p, o, c), where c stands for the context of a triple (s, p, o). Six different indexes are proposed to cover 16 possible access patterns of quads, i.e., each of s, p, o and c can be either a constant or a variable. While disk usage increases, more efficient query processing can be achieved in this way. This strategy of using multiple indexes is adopted by the Kowari system [40], and also represents an interesting direction for future development of Semplore. Compared to this line of work, Semplore is built upon scalable IR technologies to support simple semantic data lookup, similar to existing semantic search engines. It is however extended to support structured queries. In particular, it integrates keyword search and structured queries more tightly into a processing pipeline that supports the propagation and aggregation of score—this distinguishes it from keyword search support in DB-based approaches.

6.2. Ranking scheme

Ranking has been well studied in the IR community. TF/IDF-style similarity measures [2] are used to estimate the relevance between a keyword query and a textual document. Semantic Web search engines like Sindice, Watson, Swoogle, and Falcons directly adopt this IR-style ranking. Essentially, these systems provide lookup functionalities based on an IR engine such as Lucene. The IR engines of these systems are used to index ontologies and the containing semantic data. Keywords submitted by the user are then matched against the indexed resources, and results are ranked according to the matching scores returned by the IR engine. In systems like Sindice [37], some additional ad hoc rules are applied, e.g., “prefer data sources whose hostnames correspond to the resource hostnames”.

Studies such as HITS [24] and PageRank [7] capture the “popularity” of Web pages based on the linkage information. Recently, much work has been devoted to applying PageRank to relational data [3] and RDF data [35,21]. Different ranking schemes [4,9] have been proposed for structured queries on (RDF) data. They compute scores using spreading activation, which is similar to the ranking principles used in our approach. In [11], a function similar to ours is used for ranking entities extracted from Web documents. NAGA [23] considers different factors to rank structured data, e.g., the extraction confidence and the query length. Our work explicitly takes the structure of both a hybrid query and a data graph into account for score propagation and aggregation. We also propose novel algorithms to tightly integrate the ranking scheme into query processing.

6.3. Faceted search and browsing

Faceted search or browsing is a major facility to support exploratory search [39,38]. It facilitates the formulation of queries when the contextual knowledge is not sufficient, when the information space is complex to navigate, when the search task requires browsing and exploration, or when the indexes of available information in a system is inadequate.

The Flamenco system [41] originally demonstrated the success of faceted search through a comprehensive user study. Dakka et al. [13] proposed an automatic facet construction approach based on the frequency information. The faceted search and browsing module of our system was built in the same spirit as the frequency-based facet construction. facet [20,31] extended faceted navigation for Semantic Web. They also demonstrated the benefits and necessity of involving relations in faceted search. However, neither of them discussed the efficiency or evaluated the quality of the ranked results. To the best of our knowledge, Semplore is the first faceted system to tackle both issues.

6.4. Index update

Update mechanisms for documents managed using inverted indexes [8,25] are well studied. Büttcher et al. [8] presented a hybrid approach where long posting lists are updated in-place, while short lists are updated using a merge strategy. Work in [25] improves the in-place update by saving the short posting lists within the vocabulary and over-allocating the long lists.

However, there is little work on the index update for Semantic Web data. Swoogle [14] and Sindice [37] are designed as repositories of Semantic Web documents and do not consider how to deal with index update for triples. Lim et al. [27] presented a method to update previously indexed documents by combining the blocking technology together with the diff algorithm. We adapt a similar idea from our previous work [26] that extends the existing index structure to a block-based one to handle the update of the Web of Data.

7. Conclusion and future work

In this paper, we presented Semplore, an IR-based search engine which is capable of scaling to the Web of Data. It is more accessible to lay users than prevalent systems, supporting hybrid queries and faceted search. By adopting and extending the inverted index, search functionalities can be supported efficiently. The relevancy of returned results is ensured by a ranking scheme. Changes to the data are supported by a flexible index update mechanism. The experimental results have shown the potential of our solution.

The management of different types of uncertainty involved in Web data search is one important aspect of future work. We will address uncertainty at the level of result ranking as well as the level of data cleaning and integration. Furthermore, we plan to extend the query capability towards the support for multiple target
variables. We will also investigate more advanced query planning optimization techniques and leverage the potential for parallel processing of expensive operations such as mass-union.

Acknowledgement

We thank the anonymous reviewers for their valuable comments. We also thank Guilin Qi and Haitao Zheng for their careful proof-reading and constructive advice.

References