Web Semantics: Science, Services and Agents on the World Wide Web 7 (2009) 189–203

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

Hermes: Data Web search on a pay-as-you-go integration infrastructure

Thanh Tran [a,*], Haofen Wang [b], Peter Haase [a,1]

[a] *Institute AIFB, Universität Karlsruhe, D-76128 Karlsruhe, Germany*
[b] *Shanghai Jiao Tong University, Shanghai, China*

ARTICLE INFO

Article history:
Received 22 January 2009
Received in revised form 20 May 2009
Accepted 2 July 2009
Available online 9 July 2009

Keywords:
Keyword search
Structured search
Web of data
Data integration

ABSTRACT

The Web as a global information space is developing from a Web of documents to a Web of data. This development opens new ways for addressing complex information needs. Search is no longer limited to matching keywords against documents, but instead complex information needs can be expressed in a structured way, with precise answers as results. In this paper, we present Hermes, an infrastructure for data Web search that addresses a number of challenges involved in realizing search on the data Web. To provide an end-user oriented interface, we support expressive user information needs by translating keywords into structured queries. We integrate heterogeneous Web data sources with automatically computed mappings. Schema-level mappings are exploited in constructing structured queries against the integrated schema. These structured queries are decomposed into queries against the local Web data sources, which are then processed in a distributed way. Finally, heterogeneous result sets are combined using an algorithm called map join, making use of data-level mappings. In evaluation experiments with real life data sets from the data Web, we show the practicability and scalability of the Hermes infrastructure.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

The Web as a global information space is no longer only a Web of documents, but a Web of data—the *data Web*. In recent years, the amount of structured data available on the Web has been increasing rapidly. Currently, there are billions of triples publicly available in Web data sources of different domains. These data sources become more tightly interrelated as the number of links in the form of mappings is also growing. The process of interlinking open data sources is actively pursued within the Linking Open Data (LOD) project [2].

This development of a data Web opens a new way for addressing complex information needs. An example might be: "*Find articles from Turing Award winners at Stanford University*". No single LOD data source can completely satisfy our example information need. Yet, with the integration of the data sources DBLP, Freebase and DBpedia – all of them publicly available in LOD as RDF data – an answer in principle can be obtained: DBLP contains bibliographic metadata such as authors along with their affiliations, and more information about universities and award winners can be found in Freebase and DBpedia, respectively. Still, the effective exploitation of the data Web brings about a number of challenges:

Usability: Searching the data Web effectively requires the use of a structured query language. Yet one cannot assume the user to know what data sources are relevant for answering a query and their schemas. The burden of translating an information need into a structured query should not be imposed on the end users, as it would hinder the widespread exploitation of the data Web. Simple search paradigms adequate for the lay user are needed.
Heterogeneity: In order to fully exploit the data Web, available data sources need to be managed in an integrated way. However, data sources cover different, possibly overlapping domains. Data contained in different sources might be redundant, complementary or conflicting. We encounter discrepancies on the schema-level as well as the data-level, i.e. differences in the way the conceptualization, the identifiers and the data values of real world entities are represented. While the LOD project alleviates some of the heterogeneity problems by promoting the creation of links between data sources, such a (manual) upfront integration effort is only a partial solution. In order to deal with the dynamic nature and scale of the data Web, it needs to be complemented with mechanisms that can interrelate and reconcile heterogeneous sources (whose relationships might be not known a priori) in an continuous and automatic manner.
Scalability: The amount of data on the Web is ever increasing. The LOD project alone already contains roughly two billion RDF triples in more than 20 data sources. Clearly, efficient query answering that can scale to this amount of data is essential for data Web search.

* Corresponding author. Tel.: +49 721 608 4754.
E-mail addresses: dtr@aifb.uni-karlsruhe.de (T. Tran), whfcarter@apex.sjtu.edu.cn (H. Wang), peter.haase@fluidops.com (P. Haase).
[1] Present address: Fluid Operations, D-69190 Walldorf, Germany.

1570-8268/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.websem.2009.07.001

To address the problems of integration in open data spaces such as the data Web, the pay-as-you-go paradigm to data integration has been proposed. According to Madhavan et al. [19], the main concepts for an affordable integration of the various data sources on the Web are *approximate schema mappings*, *keyword queries with routing* and *heterogeneous result ranking*. Integration is regarded as a process that begins with disparate data sources and continues with incremental improvement of semantic mappings amongst them. At any point during this ongoing integration, the system should be able to process queries using the available information and mappings. Thus it is different from traditional data integration systems that require large upfront effort to manually create complete mappings for the available data sources.

In our paper, we follow the paradigm of pay-as-you-go integration and propose an infrastructure called Hermes that addresses the challenges discussed above:

- *Expressive keyword search*: In Hermes, users can formulate queries in terms of keywords. These keywords are translated to the best (top-k) structured queries representing possible interpretations of the information need. Unlike approaches in existing systems (e.g. Sindice,[2] Watson[3]) that simply match keywords against an index of data elements, the results obtained using Hermes do not only match the keywords but also satisfy the structured query computed for the keywords. While existing approaches to keyword translation focus on single data source [16,13,28], we propose a novel technique for the computation of queries that might span over multiple data sources, i.e. distributed queries.
- *Integration of Web data sources*: Hermes integrates publicly available data sources such that users can ask queries against the data Web in a transparent way. In order to support this, mappings at both the schema- and data-level are precomputed and stored in an index. Existing techniques are used for the actual computation of the mappings. This computation is embedded in a procedure that implements an iterative integration of Web data sources. In particular, it crawls data sources, extracts schemas, and automatically computes mappings as needed, i.e. only those mappings are precomputed that can be used for query processing. This substantially reduces the size of the data that have to be analyzed during the computation of mappings.
- *Efficient query processing*: We present techniques for an efficient translation of keywords to structured queries. Instead of searching the entire data space for possible interpretations [16,13], we construct a query space primarily composed of schema elements. Since it is much smaller than the data space, the search for interpretations can be performed more efficiently. For an efficient processing of the distributed queries computed from the keywords, we propose a special procedure for combining results from different data sources. In particular, we propose the *map join*, a variant of the similarity join [17,24]. This form of join is necessary to combine information about the same entities that have different representations in different data sources. An important step part of the join processing is the computation of similarities. The map join procedure can leverage the data-level mappings and thereby avoid the expensive online computation of similarities during online query processing.

The rest of this paper is organized as follows: In Section 2, we introduce the underlying data-, and query model and architecture of Hermes. We then discuss specific aspects of data and query processing in more detail: preprocessing and indexing in Section 3, translation of keywords into structured queries in Section 4, and the distributed processing of queries in Section 5. In Section 6 we report on our evaluation experiments performed with Hermes. Finally, after a discussion of related work in Section 7 we conclude in Section 8.

2. Hermes infrastructure

In this section we introduce the conceptual architecture of our Hermes infrastructure. Before discussing the components of the infrastructure in detail, we will define the data and queries involved in our data Web search setting.

2.1. Data model

We consider the data Web as a set of interrelated Web data sources, each of them identified using a data source identifier. We use a graph-based data model to characterize individual Web data sources. In that model, we distinguish between a *data graph*, capturing the actual data, and a *schema graph*, which captures the structure and semantics of the data.

Definition 1. A *data graph* g_D is a tuple (V, L, E) where

- V is a finite set of *vertices* as the union $V_E \uplus V_V$ with E-vertices V_E (representing entities) and V-vertices V_V (data values),
- L is a finite set of *edge labels*, subdivided by $L = L_R \uplus L_A$, where L_R are relation labels and L_A are attribute labels.
- E is a finite set of *edges* of the form $e(v_1, v_2)$ with $v_1, v_2 \in V$ and $e \in L$. Moreover, the following types are distinguished:
  - $e \in L_A$ (A-edge) if and only if $v_1 \in V_E$ and $v_2 \in V_V$,
  - $e \in L_R$ (R-edge) if and only if $v_1, v_2 \in V_E$,
  - and *type*, a pre-defined edge label that denotes that denotes the membership of an entity in a particular class.

In a data graph g_D, we do not distinguish between different types of entities in v_E, such as classes and individuals. Classes and other schematic elements can be explicitly defined through a schema graph.

Definition 2. A *schema graph* g_S is a tuple (V, L, E) where

- V is a finite set of *vertices*. Here, V is conceived as the disjoint union $V_C \uplus V_R \uplus V_A \uplus V_D$ with C-vertices V_C (classes), R-vertices V_R (relations), A-vertices V_A (attributes), and D-vertices V_D (data types).
- L comprises of the pre-defined edge labels subclass of, domain, range.
- E is a finite set of *edges* of the form $e(v_1, v_2)$ with $v_1, v_2 \in V$ and $e \in L$, where
  - e = domain if and only if $v_1 \in V_A \cup V_R$ and $v_2 \in V_C$,
  - e = range if and only if $v_1 \in V_A, v_2 \in V_D \cup V_C$ or $v_1 \in V_R, v_2 \in V_C$, and
  - e = subclass of if and only if $v_1, v_2 \in V_C$.

The presented model is general enough to represent different types of Web resources. In particular, it captures RDF(S) as a special case.[4] But also XML data can be represented as graphs. Web documents form a graph where documents are vertices and links correspond to edges. In many approaches [16,13], even databases have been treated as graphs where tuples correspond to vertices and foreign relationships to edges.

[2] http://sindice.com.
[3] http://watson.kmi.open.ac.uk.

[4] The intuitive mapping from RDF(S) to our data model is: resources correspond to entities, classes to classes, properties to either relations or attributes, literals to data values.

To match the nature of Web data sources, we assume that in many cases, a schema might be incomplete or does not exist for a given data graph.

To interrelate the elements of individual data sources, our data model is extended with mappings:

Definition 3. A *mapping M* is set of mapping assertions representing approximate correspondences between graph elements. Specifically, mapping assertions in M are of the form $m(v_1, v_2, s)$ where $v_1, v_2 \in V$ are graph vertices and $s \in [0, 1]$ is a score denoting a confidence value associated with the mapping.

Data sources together with mappings relating them form a *data Web* defined as an *integrated data graph*:

Definition 4. An *integrated data graph* g_{ID} is a tuple (G_D, M_D), where G_D is a finite set of data graphs and M_D is a set of approximate correspondences between data graph E-vertices called *individual mappings*.

Analogously, we define the *integrated schema graph*:

Definition 5. An *integrated schema graph* g_{IS} is a tuple (G_S, M_S), where G_S is a finite set of schema graphs and M_S is a set of approximate correspondences between schema elements, i.e. *class mappings* $(v_1, v_2 \in V_C)$, *relation mappings* $(v_1, v_2 \in V_R)$ and *attribute mappings* $(v_1, v_2 \in V_A)$.

While edges of a particular graph are called *intra-data-source edges*, edges representing mappings between elements of different data graphs will be referred to as *inter-data-sources edges*.

In contrast to local- and global-centric approaches to data integration, there is no assumption of a *mediated schema* in our approach. Constructing and maintaining a mediated schema that provides a shared vocabulary for all resources on the highly dynamic Web environment is difficult [5]. Also, the complexity and overheads in mapping local schemas with the mediated schema is not affordable. In our approach, mappings might exist between any pair of data sources on the Web. The creation and maintenance of mappings in this *mapping between local schemas* approach is simpler and thus more manageable with respect to the data Web.

Finally, we note that since graph elements range over individuals, classes, relations and attributes, the notion of mapping employed in our approach is more general. It is not restricted to schema elements only [9] but includes also data-level correspondences.

Example 1. Fig. 1 illustrates data graph fragments for Freebase, DBLP and DBpedia. Together with the individual mappings $m1$ and $m2$, these three graphs form an integrated data graph covering various domains. This data might be used to address the information need motivated by our example. A corresponding integrated schema graph will be shown in Fig. 3 (augmented with keyword matching elements that will be discussed in Section 3.1).

2.2. Query model

In our setting, we distinguish between the notion of a *user query* and a *system query*. While the system query is constructed using a structured query language, the user query can be expressed using keywords. Keyword queries are preferable in our setting, as the relevant data sources, their schemas and labels might not be known to the user a priori.

Specifically, the *user query* $\mathcal{Q}_U$ is a set of keywords $(k_1, \ldots, k_i)$. The system queries $\mathcal{Q}_S$ are *c*onjunctive queries. Conjunctive queries have high practical relevance because they are capable of expressing the large class of relational queries. The vast majority of query languages for many data models used in practice fall into this fragment, including large parts of SPARQL and SQL.

Definition 6. A *conjunctive query* is an expression of the form $(x_1, \ldots, x_k).\exists x_{k+1}, \ldots, x_m.A_1 \wedge \ldots \wedge A_r$, where $x_1, \ldots, x_k$ are called *distinguished variables*, $x_{k+1}, \ldots, x_m$ are *undistinguished variables* and $A_1, \ldots, A_r$ are *query atoms*. These atoms are of the form $P(v_1, v_2)$, where P is called *predicate*, v_1, v_2 are variables or, otherwise, are called *constants*.

Since variables can interact in an arbitrary way, a conjunctive query q can be seen as a graph pattern constructed from a set of triple patterns $P(v_1, v_2)$ in which zero or more variables might appear. A solution to q on a graph g is a mapping μ from the variables in the query to vertices e such that the substitution of variables in the graph pattern would yield a subgraph of g. The *substitutions of distinguished variables* constitute the answers (cf. [28] for formal definition of these answers).

We can define the semantics of a conjunctive query over an integrated data graph $g_{\text{ID}} = (G_D, M_D)$ by simply considering the union of the individual data graphs in G_D, in which all graph elements whose correspondences are above a certain threshold confidence (as defined by the mappings M_D) are treated as identical.

2.3. Conceptual architecture

Fig. 2 depicts the conceptual architecture underlying Hermes. In the architecture, we can distinguish between components supporting *offline* and *online* processes.

2.3.1. Offline processing of data graphs

During offline *graph data processing*, different information are extracted from the data graphs and stored in specific data structures of the *Internal Indices*. Firstly, the labels of data graph elements are *extracted*. A standard *lexical analysis* including stemming, removal of stopwords and term expansion using *Lexical Resources* (e.g. WordNet) is performed on the labels, resulting in a set of terms. These terms are stored in the *keyword index*. If no schema information is available, we apply *summarization* techniques to construct a schema graph for a given data graph. Schema graphs are stored in a *structure index*. For ranking support, *scores* are computed and associated with elements of the keyword and the structure indices. Additionally, tools are employed to discover mappings at both the data- and schema-level. The computed mappings are stored in a separate internal index called the *mapping index*. The internal indices are used to identify the elements in the data graph matching a keyword, and to retrieve schema graphs and mappings.

2.3.2. Online keyword query processing

The processing of keyword queries over Web data sources can be decomposed into three main steps, namely keyword translation, distributed query processing and local query processing. The input, the intermediate queries as well as the output for our example are shown in Fig. 2(a).

Keyword translation focuses on translating keywords to query graphs—intermediate representations of the user information need from which conjunctive queries will be derived. Keywords are firstly submitted to the keyword index to determine whether they can be answered using the available data. This step referred to as *k* eyword mapping results in a set of keyword elements. These keyword elements are combined with schema graphs retrieved from the structure index to construct a query space. During top-*k* query graphs search this query space is explored to find query graphs, i.e. substructures that connect all keyword elements. According to a query ranking scheme, the computed query graphs are sorted and finally, presented to the user for *selection*.

The selected queries might cover multiple data sources. During *distributed query processing*, the query graph selected by the user is *decomposed* into parts that can be answered using a

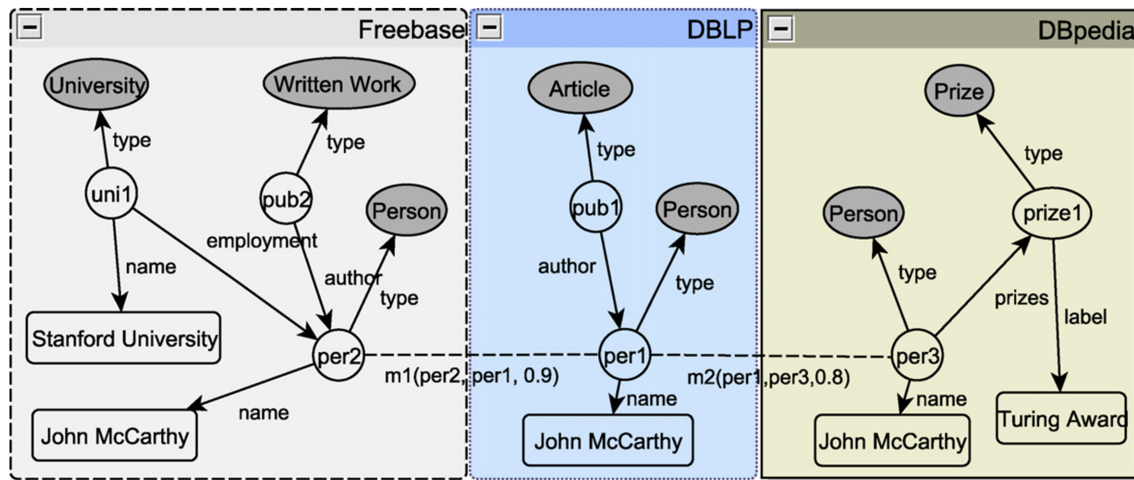

Fig. 1. Integrated data graph.

particular data source. The optimal order for processing these parts is determined during *query planning*. The query parts resulting from query decomposition are sets of triple patterns. According to the semantics of our query model, processing these patterns amounts to finding substitutions of variables in this pattern. For this, two modes of operations are supported: (1) routing the query parts to external engines or (2) processing the query parts internally using the internal *graph data indices*. For local query processing, each of the query parts are *mapped* to the syntax of conjunctive queries that is supported by the respective Web data source, e.g. SQL or SPARQL. Finally, the results retrieved for each query part are *combined*, i.e. a set of *join* operations are performed on the intermediate results. In order to deal with the differences in data representation in an efficient way, we utilize a special procedure called map join. This is a special implementation of similarity join [17,24], which however, can leverage the individual mappings stored in the index to avoid the online computation of similarities during join processing.

2.3.3. *Distribution of data and control*

Fig. 2 illustrates a possible physical distribution of the data: internal indices are maintained in Hermes while Web data sources are distributed across multiple nodes. The owner of these data sources provide capabilities for local query processing accessible via open interfaces, e.g. in the form of SPARQL endpoints. Some of the data sources are replicated and maintained in Hermes as graph data indices.

A different physical distribution is possible. Data source owners might want to have complete control. In this case, all the data as well as the internal indices on that data are maintained and distributed across multiple nodes. On the other hand, all data sources might be replicated and maintained under centralized control. With more control, the owner of the search engine can ensure reliability, quality and performance. Apparently, this realization of Web search has proven to be practicable for document retrieval.

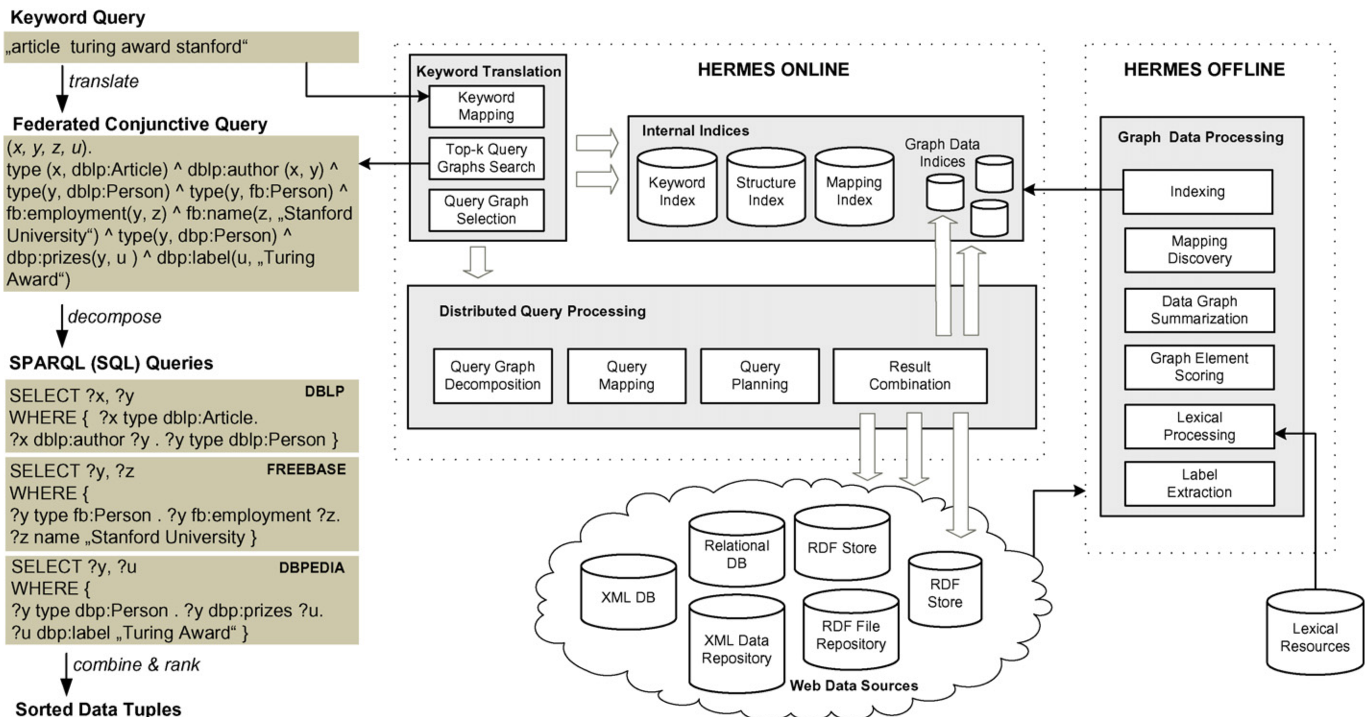

Fig. 2. (a) Example queries. (b) Hermes infrastructure.

Table 1
Indexing graph elements and mappings.

Terms	Graph element type [data structure]
e_r, synset of e_r	Relation label $[e_r, g_{D_{e_r}}]$
e_a, synset of e_a	Attribute label $[e_a, g_{D_{e_a}}, (v_{c_1}, \ldots, v_{c_n})]$
v_e label, synset of v_e	Entity vertex $[v_e, g_{D_{v_e}}]$
v_v data value	Value vertex $[v_v, g_{D_{v_v}}, e_a, (v_{c_1}, \ldots, v_{c_n})]$
$g_{S_{n_1}}$ id, $g_{S_{n_2}}$ id	Schema mapping $[n_1, g_{S_{n_1}}, n_2, g_{S_{n_2}}, s]$
v_c label	Data mapping $[n_1, g_{D_{n_1}}, n_2, g_{D_{n_2}}, s]$

In practice, the actual distribution of data and control will depend on how much the data source owners are willing and able to expose the data and how much the owner of the engine requires and can afford centralized control.

3. Data preprocessing

This section describes the offline process where the data graphs are preprocessed and stored in specific data structures of the internal indices.

3.1. Construction of the keyword index

Keywords entered by the user might correspond to elements of the data graph (entities, data values and edge labels). For mapping keywords to elements of the data graphs, we employ a *keyword index* that is commonly employed for keyword search [28,16,13]:

Definition 7. The *keyword index* is a keyword element map that is used for the evaluation of a multi-valued function $f : K \to 2^{V_E \uplus V_V \uplus L}$, which for each keyword returns the set of corresponding *keyword matching elements*.

As summarized in Table 1 (upper part), this index returns for a term, e.g. $e_r =$ "Stanford University" (or some synonyms of its synset) a complex data structure that contains besides the corresponding graph element also information about the *origin* (data source identifier $g_{D_{e_r}}$). In the case the term corresponds to an attribute label e_a or a value vertex v_v, a set of adjacent graph elements is also returned. The attribute label is stored along with its adjacent class vertices $v_{c_1}, \ldots, v_{c_n}$. For a value vertex v_v, the adjacent attribute edge e_a as well as class vertices $v_{c_1}, \ldots, v_{c_n}$ is maintained in the element data structure. This information about adjacent elements constitutes the immediate neighborhood of these vertices. Intuitively speaking, the neighborhood information will be used for the on-the-fly construction of the query search space (Section 3.2).

Example 2. With respect to the data in Fig. 1, the next adjacent attribute label and class vertex to the value vertex from Freebase labelled Stanford University, is name and University respectively. Thus the data structure [Stanford University, Freebase, name, (University)] will be returned for the term "Stanford University".

The keyword index is implemented using an inverted index, i.e. every data graph element along with its associated data structure is stored as a document, and its label will be used as the document term.

Example 3. The value vertex with the label Stanford University will be stored as a separate document along with the data [Stanford University, Freebase, name, (University)]. This document contains only one term, namely "Stanford University".

In order to support an effective and robust keyword mapping, a lexical analysis (e.g. stemming, stopword removal) as supported by standard IR engines (cf. Lucene[5]) is performed on the labels extracted from the data graphs to obtain a list of terms to be indexed for a particular graph element. The Levenshtein distance is used for supporting an imprecise matching of keywords against terms based on syntactic similarity. Further, terms are expanded with semantically related entries extracted from WordNet[6] (synonyms) to support matching based on semantic similarity. This means that besides the element label, the document created for a graph element contains also synonyms for that label.

Due to this imprecise matching, elements retrieved from this keyword index are associated with *a score* denoting the degree of matching.

3.2. Construction of the structure index

We use the structure index to perform an efficient exploration of substructures that connect keyword matching elements. The structure index is basically an "augmented" schema graph. It has been shown in [28] that the exploration of query graphs on the structure index is more efficient than using the data graph (cf. [13,16]. Instead of defining a structure index for a single source [28], we extend it to a multi data sources scenario as follows:

Definition 8. The *structure index* is a map that represents for a data source identifier the corresponding schema graph $f : \mathbb{N} \to G_S$

The structure index can thus be used to retrieve the schema graphs for the Web data sources from which the keyword matching elements originate.

In practice, a schema is often not available or incomplete (e.g. for data graphs in RDF). In these cases, techniques for computing structural indices [11,28] are employed. In particular, a schema graph is derived from a given data graph through the following steps:

(i) Delete all *V*-vertices and *A*-edges
(ii) Every E-vertex v_{e_i} that is associated with a C-vertex v_c, i.e. there is an edge *type*(v_{e_i}, v_c), is deleted and v_c inherits all *R*-edges of v_{e_i}. Thereby, all relations specified for v_{e_i} are captured by its class v_c.
(iii) Every other E-vertex v_{e_j} that has no explicit class membership is associated with a pre-defined vertex *Thing*. Also, *Thing* inherits all *R*-edges s.t. all relations exhibited by v_{e_j} are captured by this pre-defined class vertex.

In essence, we attempt to derive relations between classes from the connections given in the data graph. Using the top-level class *Thing*, this works even when there are no explicit class memberships given in the data graph. It is straightforward to prove that via this procedure, all *R*-edge paths in the data graph are captured by the resulting schema graph, i.e. for every *R*-edge path in the data graph, there is at least one corresponding path in the schema graph. Thus, we can use this computed schema graph for exploring paths (comprising of *R*-edges only), instead of relying on the data graph. Note that the procedure presented here is similar to database approaches for computing structure indices (cf. the data guide concept [11]).

3.3. Construction of the mapping index

As discussed, there is no global schema in our approach. Instead, pairwise correspondences between elements of the schema graphs are computed. Additionally, correspondences at the level of data

[5] http://lucene.apache.org.
[6] http://www.cogsci.princeton.edu/wn/.

graphs are considered such that mappings might involve classes, relations, attributes or individuals:

Definition 9. A *mapping index* is used to store and to retrieve two types of mappings: (1) Given an identifier for the schema graph $g_S = (V, L, E)$, it returns all mappings associated with that graph, i.e. all $m(v_1, v_2, s)$ where $v_1 \in V$ or $v_2 \in V$. (2) Given a class vertex v_c, it returns all mappings for individuals of that class, i.e. all $m(v_1, v_2, s)$ where $type(v_1, v_c)$ or $type(v_2, v_c)$.

A separate inverted index is used for the storage and retrieval of the mappings. The indexing of these mappings as documents is depicted in Table 1 (lower part). Given the data source identifier for a schema graph (g_{Sn_1} or g_{Sn_2}), the mapping index can be used to retrieve all schema-level mappings specified for that graph. Likewise, given the label of a class vertex v_c, all data-level mappings (individual mappings) computed for v_c will be returned.

The use of these mappings is not restricted to federated query processing [25]. They are also exploited during query translation (for exploration of interpretations spanning multiple graphs) and during result combination (for joins based on individual mappings).

A mapping discovery service is employed to obtain these mappings. In order to obtain high quality mappings, we follow a standard process established in state-of-the-art mapping approaches. This process can be decomposed into (1) engineering of similarity features, (2) selection of candidates, (3) computation of similarities (4) aggregation of similarities, and (5) derivation of correspondences based on the aggregated similarity values. For the similarity measures, we rely on existing, well-known measures that have proven effective in state-of-the-art matching systems [10]. For the sake of scalability, we primarily use simple, but effective measures based on syntactic and structural features.

Mappings are first computed for pairs of schema graphs. For every resulting class mapping, correspondences between individuals of the involved classes are examined. That is, only individuals of two given classes are processed at a time. Since the number of individuals in the involved data graphs might be very large, this “focussed” discovery of mappings is essential for efficiency and scalability. Also, this integration process is in line with our “pay-as-you-go” paradigm as mappings are only computed as needed: we will show that only individual mappings that are associated with class mappings are actually used for result combination.

3.4. Scoring of graph elements

Scores indicating the relevance of elements within the graph are essential for ranking (of both translated queries and answers). A popular technique for deriving scores for graph elements is PageRank [3]. However, the application of PageRank is not straightforward in the data Web scenario. Unlike links on the Web, the edges between data elements have different semantics. As a result, the effectiveness of PageRank heavily depends on the weights assigned to different edge types—a task that is assumed to be performed by a domain expert. This manual upfront effort might be affordable for a single data source, but certainly not for the data Web setting.

In our previous work, we have proposed a simpler technique for the computation of popularity that is based on “the frequency” of a vertex. In particular, the score of a schema graph element correlates with the number of entity vertices [28]. This has shown to be a reliable measure for computing the popularity of an element w.r.t. a data source. For a multi-data-source scenario, we propose to combine this notion with the distinctiveness of an element, i.e. how well an element discriminates a given data source from others. In particular, we propose an adoption of the TF-IDF concept for scoring Web data. The main aim is to strike a balance of effectiveness and efficiency that is appropriate for data Web search: the scoring mechanism should allow for proper estimates of the popularity of a graph element, albeit being affordable such that the amount of (manual upfront) effort is manageable. In our approach, scores are used only for ranking queries. Thus, we will focus on the scoring of elements of the schema graph:

Popularity w.r.t. a data source: The term frequency (TF) has proven to be an effective measure for popularity in the context of document retrieval. Based on this notion, we define the element frequency (EF) as a measure for the popularity of a graph vertex v_i w.r.t. the particular data source $g_j(V, L, E)$ containing v_i. This measure is simply the number of occurrences occ_{v_i,g_j} of v_i in the data source g_j normalized with the number of occurrences of all vertices in g_j to avoid the effect of data source size, i.e. $EF_{v_i,g_j} = (\mathrm{occ}_{v_i,g_j})/(\sum_{v_k \in V} \mathrm{occ}_{v_k,g_j})$.

This metric is applied for scoring the vertices V_C and V_R in the schema graphs. For a class vertex $v_c \in V_C$, the number of occurrences $\mathrm{occ}_{v_c,g_{S_j}}$ is the number of individuals v_i that are of type v_c, i.e. $type(v_i, v_c)$, in the corresponding data graph. Similarly, for a relation vertex $v_r \in V_R$, the number of occurrences $\mathrm{occ}_{v_r,g_{S_j}}$ is the number of instantiations of the relation.

Distinctiveness w.r.t. the data Web: The inverse data source frequency (IDF) can be seen as a measure for the distinctiveness of a vertex v_i w.r.t. to the data Web. For a vertex v_i, it is defined as $\mathrm{IDF}_{v_i} = \log(|g_S|)/(|g_{S_{v_i}}|)$ where $|g_S|$ is the total number of schema graphs in the structure index and $|g_{S_{v_i}}|$ the number of schema graphs containing v_i.

The total score of a schema graph element is defined as $\text{EF-IDF}_{v_i,g_{S_j}} = EF_{v_i,g_{S_j}} \cdot \mathrm{IDF}_{v_i}$ Compared to the frequency metric used in [28] (similar to the EF measure defined above), the additional use of IDF helps to discount the impact of elements that appears commonly throughout the data Web. An element v_i that has a high EF-IDF is important for a data source and at the same time, is effective in discriminating that data source from others. Intuitively speaking, the distinctiveness of an element helps to find and prioritize the right data source during the translation and ranking of queries (just like in IR, where IDF of a term helps to find the right document).

Example 4. Fig. 4(a) shows EF-IDF scores for elements of the integrated schema graph constructed for our running example. The vertex connected with Stanford University for instance, has an EF-IDF score of 0.027, which is substantially higher than the score of the vertex connected with Turing Award (its ED-IDF is 0.0089). This is due to two factors: it denotes University, which contains many more instances than the other vertex, which stands for Price (thus, its EF is higher). Also, whereas University occurs only in Freebase, Price is a common concept that is mentioned in both Freebase and DBLP (thus, its IDF is higher).

4. Keyword query translation

In this section, we describe the computation of possible interpretations of the user keywords. These interpretations are presented to the user in the form of *query graphs*. For computing such query graphs from keywords, Ref. [28] proposes a procedure consisting of three main steps: (1) construction of the query search space and (2) top-k query graph exploration, and (3) query graph ranking. We extend this work on keyword search to the data Web scenario. Instead of a single data source, the search space in our approach *spans multiple graphs*. Also, the ranking mechanism has been extended to incorporate aspects that are specific for the data Web. The rank of a computed query graph reflects not only the popularity of the graph elements it is composed of, but also the r elevance of the data graphs (data sources) it spans over.

4.1. Construction of the query search space

The query search space shall contain all elements that are necessary for the computation of possible interpretations. Commonly, keyword search bases on the assumption that keywords denote some elements that can be found in the data [27]. Thus, the search space employed for keyword search is typically the data graph [13,16]. Similar to [28], we employ a query-specific search space (called *query space*), consisting of two parts: (1) the graph elements that match the user keywords (to explore the query constants) and (2) the structural elements of the data graph (to derive the query predicates). Since most of the elements in the data graphs are omitted, the use of such a query space can offer substantial increase in performance when compared to search over the entire data graph [13,16].

In order to obtain the first part, the keywords entered by the user are submitted against the keyword index. Note that the data structure of the resulting *keyword elements* bears origin information (data source identifiers) as well as information about adjacent elements.

The identifiers of these relevant data sources are submitted against the structure index to obtain the second part, i.e. a set of *s* chema graphs that are relevant for the query.

The schema graph and the keyword elements are then combined to obtain the query space. For this, the information about adjacent elements is used to connect the keyword matching elements with the corresponding elements of the schema graphs:

- If the keyword matching element is a value vertex $v_v^k \in V_V$ with the adjacent elements being the attribute edge label $e_a \in L_A$ and the class vertices $v_{c_1}, \ldots, v_{c_n} \in V_C$, then the edges $e_a(v_v^k, v_{c_1}, \ldots, v_{c_n})$ will be added to connect v_v^k with the class vertices $v_{c_1}, \ldots, v_{c_n}$ of the relevant schema graph (i.e. the schema of the data source v_v^k originated from).
- If the keyword matching element is an attribute edge label $e_a^k \in L_A$ with the adjacent elements being the class vertices $v_{c_1}, \ldots, v_{c_n} \in V_C$, then the edges $e_a^k(\text{value}, v_{c_1}, \ldots, v_{c_n})$ will be added to the relevant schema graph. Note that the *value* vertex is an artificial element employed to include matching attribute edges in the query space.
- Otherwise, the keyword element must be a class vertex $v_c^k \in V_C$ or a relation edge label $e_r^l \in L_R$. In this case, no further elements shall be added as the relevant schema graph shall already contain the keyword element.

Specific to the data Web scenario are *mappings*. Since possible interpretations of the user keywords might span multiple data sources, these mappings need to be considered in the construction of the query space. Thus, the identifiers of the relevant schema graphs are also submitted against the mapping index to obtain a set of relevant mappings. Together with the schema graphs augmented with keyword elements, the mappings constitute the following query space:

Definition 10. The query space is an *integrated schema graph* $g_I^q = [G_S(V, L, E), E_I]$ that is augmented with keyword matching elements N_K computed for a given query q, i.e. g_I^q comprises a set of schema graphs G_S augmented with

- the edges $e(v, v_k)$, $e_k(v, \text{value})$ and $e_k(v, v_k)$, where $v_k, e_k \in N_k$, $e, e_k \in L$, $v, v_k \in V$, and *value* is an pre-defined vertex,
- and the edges $m(v_i, v_j, s) \in E_I$ where v_i is a vertex of a schema graph g_{S_i} and v_j is a vertex of a schema graph g_{S_j}.

Example 5. Fig. 3 illustrates the query space constructed for our example keyword query. The keyword elements are *Article*, *Stanford University* and *Turing Award*. These elements originated from the three different data graphs Freebase, DBLP and DBpedia. The corresponding schema graphs are retrieved. Keyword elements not covered by these schemas are added. In particular, the adjacent e_a (*name*) is used to connect *S* tanford University with the adjacent v_c (*University*). Likewise, an edge with the label *label* is created to connect *Turing Award* with *Prize*. For these schemas, the mappings $m3$, $m4$, $m5$, and $m6$ have been found. Corresponding edges are created to establish links between the schema graphs.

4.2. Exploration of top-k query graphs

Given the query space, the remaining task is to search for the minimal query graphs in this space. With respect to our data models, a query graph is formally defined as follows:

Definition 11. Let $g_I^q = (G_S, N_K, E_I)$ be the query space, $K = \{k_1, \ldots, k_n\}$ be a set of keywords, and let $f : K \to N'_K$ be a function that maps keywords to sets of corresponding graph elements (where $N'_K \subseteq N_K$). A *query graph* is a matching subgraph of g_I^q defined as $g_q = (G_S^q, N_K^q, E_I^q)$ with $G_S^q \subseteq G_S, N_K^q \subseteq N_K, E_I^q \subseteq E_I$ such that

- for every $k \in K, f(k) \cap N_K^q \neq \emptyset$, i.e. g_q contains at least one representative keyword matching element for every keyword from K, and
- g_q is connected, i.e. there exists a path from every graph element to every other graph element.

A matching graph g_{q_i} is minimal if there exists no other g_{q_j} of g such that $\text{Score}(g_{q_j}) < \text{Score}(g_{q_i})$.

We extend the top-k procedure proposed in our previous work [28] to find such query graphs. This procedure starts from the keyword elements N_K and iteratively explores the query space g_I^q for all distinct paths beginning from these elements. For top-k termination, we maintain a queue of elements and paths respectively, which we have explored, along with their scores.

- First, we initialize the queue with the keyword matching elements N_K, from which we shall start the exploration.
- Iteratively, the element with the highest score (the top element in the queue) is chosen for “expansion”, i.e. it is expanded to the next neighbor that has been not visited before and can be reached with lowest cost. Note that every such expansion constitutes an exploration along a particular path. The score of the path explored this way is updated based on the cost incurred by the expansion. The updated path along with it score goes back into the queue.
- At some point, an element might be discovered to be a connecting element, i.e. there is a path from that element to at least one keyword element, for every keyword in K.
- These paths are merged to form a query graph.
- The graphs explored this way are added to the candidate list.
- The process continues until the upper bound score for the query graphs yet to be explored (derived from the paths managed in the queue) is lower than the score of the k-ranked query graph in the candidate list.

Compared to our previous work [28], the procedure we employ operates on several data sources, i.e. it deals with keyword search on the data Web. During exploration, an existing path might be expanded to include elements of a different data source. That is, we traverse also along mappings to find queries spanning multiple data sources. Since mappings have different semantics than standard edges connecting elements within a data graph, we distinguish inter-data-source edges ($e \in M_S$) from intra-data-source edges ($e \in E$). These two types of edges have different prioritization

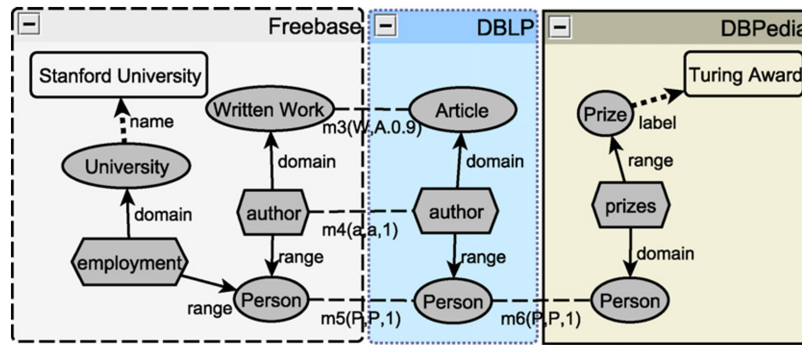

Fig. 3. Query space: integrated schema graph augmented with keyword-matching graph elements.

during exploration. This is incorporated into the scoring and ranking mechanism, which will be discussed in the next section.

Example 6. Fig. 4(a) shows an example query space containing elements associated with scores (discussed in the next section). The keyword matching elements *Stanford University*, *Article* and *Turing Award* are highlighted (labels of "non-keyword elements" are omitted due to lack of space). These keyword matching elements are put into the queue. For the first iteration, either *Turing Award* or *Stanford University* is chosen for expansion (as both have the same score, which is the highest). From *Turing Award* for instance, we would expand the current path to the node EF-IDF = 0.0008, resulting in a path with updated score $= 0.8 + s(0.0008)$. Since exploring this element adds cost to the path, the updated score will be lower than 0.8 ($s(0.0008)$ is a negative value). Thus, the next best element chosen from the queue would be *Stanford University*. The different paths starting from the keyword elements that are explored this way are shown in Fig. 4(a). Note that these paths meet at several elements, e.g. they connect EF-IDF = 0.012 with all the three keyword matching elements. An example query graph that can be derived from such connections is shown in Fig. 4(b) (the mapping of elements of query graphs to variables of conjunctive queries is discussed in Section 5). Clearly, this example also shows that mappings provide "bridges" between data sources. Expansions across data sources through these bridges are needed in order to connect keyword matching elements found in different data sources. If there were for instance no mapping connecting EF-IDF = 0.029 in Freebase with EF-IDF = 0.012 in DBLP, queries that can be computed through this algorithm may contain *Article* and *Turing Award* or only *Stanford University*, but would not capture the meanings of all the three keywords.

4.3. Scoring query graphs

The previous top-k computation outputs the query graphs with highest scores. The quality of the translation results thus depends largely on the scoring function used for calculating the scores of paths that are explored during the process.

In keyword search [13,15,12,28], scoring typically incorporates three different aspects: (1) the popularity of graph elements, (2) the matching score of keyword elements (captures imperfection in the mapping of keywords to graph elements) and (3) the length, where queries of shorter length are preferred due to the assumption that closely connected entities more likely match the information need [27]. In particular, since every query graph g_q is constructed from a set of paths P, the score of g_q can be defined as a monotonic aggregation of its path scores, which in turn, are computed from the element scores, i.e. $C_{g_q} = \sum_{p_i \in P}(\sum_{n \in p_i} C_n)$, where C is in fact not a score, but denotes the cost [28]. The lower the cost of g_q, the higher should be it rank. In the simplest scheme, the cost of an element C_n is 1, i.e. only the length is incorporated. In [28], it has been shown that a more effective scheme can be obtained by combining the length with the matching score and the popularity score.

In order to deal with the additional levels of uncertainty involved in keyword search on the data Web, we extend existing scoring schemes to define the cost for query graphs as $C_{g_q} = \sum_{p_i \in P} \sum_{n \in p_i} 1/(S_n * \text{coverage}(g_{D_i}))$, where

$$S_n = \begin{cases} S_{\text{sim}}(n) & \text{if } n \in E_I \\ \text{EF-IDF}(n) & \text{if } n \text{ element of } g_{S_i} \\ S_m(n) & \text{if } n \in N_K. \end{cases}$$

Note that $C_n = 1/(S_n * \text{coverage}(g_{S_i}))$ and $S_n \in [0, 1]$, i.e. the various scores denoted by S_n are turned into costs such that the higher the score of an element n, the lower is the cost n contributes to the paths it belongs to.

Factors that are considered in this scoring function include the matching score S_m which can be obtained for keyword matching elements N_K returned from the keyword index. The importance score *EF-IDF* computed offline for elements of all schema graph g_{S_i} is also employed. The above formula shows that associated score when an element is a keyword matching element ($n \in N_K$)

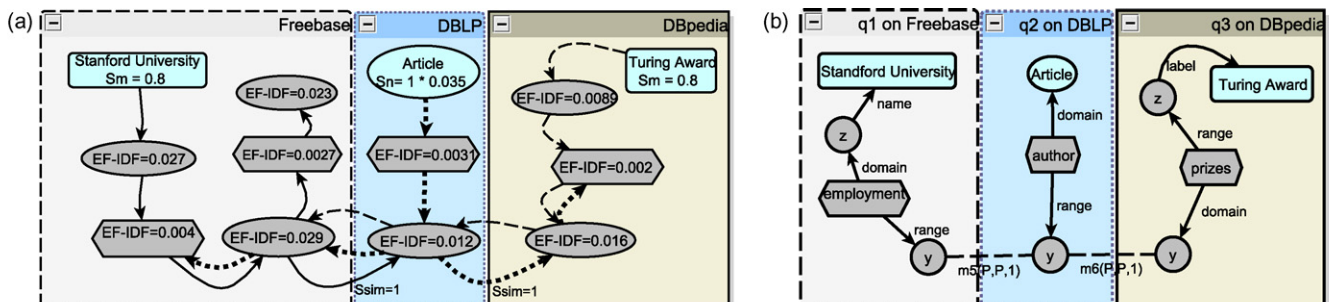

Fig. 4. (a) (Scores of) three paths through the query space and (b) query graph mapped to conjunctive query.

or when it is an element of the schema graph (n element of g_{S_i}). In the special case where a schema graph element match the keyword, we combine the matching score with its importance score, i.e. $S_n = \text{EF-IDF}(n) * S_m(n)$. Since the cost of a path monotonically increases with the number of constituent elements, the length is also implicitly captured by this cost function.

Besides these factors typically used in keyword search (and in our previous work [28]), also the mapping score S_{sim} is incorporated. Note that this score is associated with every inter-data-source edge and measures the quality of the denoted mapping. Another factor that is specific for keyword search on the data Web is the coverage. This notion is defined as the number of keywords that can be processed with a data graph g_{D_i}. The coverage of a graph g_{D_i}, i.e. coverage(g_{D_i}), is computed during the processing of keywords against the keyword index by counting the number of keywords that match elements of g_{D_i}.

Based on this ranking function, top-k exploration is guided towards finding those query graphs containing paths of shorter length, i.e. containing fewer elements. Further, paths are preferably explored when the constituent elements are important and match the user keywords. Exploration across schema graphs are preferred along inter-data-source edges denoting high quality mappings.

According to the notion of coverage, schema graphs that can "answer" a large number of keywords are prioritized. This is because the coverage of a data graph influences the scores of all its elements (since coverage is applied to every S_n). This results in a gap between scores of elements from different schema graphs, i.e. there are different score levels for schema graphs with different coverage. Hence, exploration likely starts from schema graphs which cover a large number of keywords. Further, the use of coverage has the effect that exploration across schema graph boundaries is discouraged as it would incur a substantial increase in cost, especially when the difference in score level between the two graphs under consideration is large. Note that the intended outcome is in line with the intuition that while a combination of data from different data sources might be necessary, only as few sources as needed shall be considered for answering a keyword query.

Example 7. Fig. 4(a) shows example scores of different types. In particular, the three keyword elements are associated with the matching scores S_m. Each of these elements comes from a different data source. Thus, the *coverage* of DBLP, DBPedia and Freebase is simply 1. There is no gap, and thus no bias towards a particular data source. This is reasonable because none of them can completely answer the given keyword query, i.e. all three have to be used to answer the query. Since the mappings in this example are of high quality, the mapping scores S_{sim} associated with the inter-data-source edges are simply 1. Also, example EF-IDF scores are shown for the elements of the query space under consideration. *Article* is the only element that is both a keyword matching element and a schema element and thus is associated with an aggregated score.

5. Distributed query processing

Query translation results in a list of top-k query graphs. Distributed query processing is the subsequent step that starts with the query graph g_q selected by the user. The query graph is d ecomposed into parts such that each part can be evaluated against a particular data graph. Before *routing*, each part needs to be *mapped* to the query format supported by the local query engines. For optimizing performance, a *planner* is employed to determine an appropriate order of query execution. Finally, the results obtained from the local query processors are *combined* to arrive at the final results.

5.1. Query graph decomposition

As defined previously, a query graph g_q contains two types of edges: intra-data-source edges connecting elements of a single summary graph and inter-data-source edges $e_i \in E_I^q$ connecting elements of two summary graphs. Based on this structure, query decomposition can be accomplished by simply omitting all e_i from the query graph. The resulting query graph is a set of strongly connected components g_{q_i} containing only intra-data-source edges. Each g_{q_i} represents a partial query that can be evaluated against a single data graph g_{D_i}. Fig. 4(b) illustrates the decomposition of the example query into three parts: $q1$ on Freebase, $q2$ on DBLP, and $q3$ on DBpedia.

5.2. Query planning

Query planning concerns with the order of execution of the partial queries. For this task, an "abstract" query graph g_q' is employed. Its vertices represent the partial queries and the inter-data-source edges E_I^q constitute links between them. Given g_q', query answering breaks down to two operations: (1) processing the vertices of g_q' to obtain intermediate result sets (referred to as local query processing), and combining the intermediate results along the inter-data-source edges. The optimal order of execution of these operations is estimated according to the optimization techniques proposed for RDF in [22]. In particular, statistics (e.g. about selectivity) are collected to arrive at estimates for (1) prioritizing vertices that more likely lead to smaller intermediate result sets and (2) selecting a cost-efficient join implementation (nested-loop vs. bind join), given two intermediate result sets. With respect to the example illustrated in Fig. 4(b), vertices of g_q' are simply $q1$, $q2$ and $q3$.

5.3. Query graph mapping

During this step, the partial query graphs g_{q_i} are translated to queries that can be answered by the local query engines. This translation is performed during local query processing. Basically, edges of the query graphs are mapped to predicates whereas vertices are mapped to variables and constants of the conjunctive query. Fig. 4(a) together with Fig. 4(b) exemplify these correspondences. We now give a more precise mapping of query graphs to conjunctive queries. Since we are concerned with partial query graphs, edges must be of the form $e(v_1, v_2)$, where $e \in L_A \uplus L_R$ and $v_1, v_2 \in V_C \uplus V_V \uplus \{\text{value}\}$, i.e. there are only intra-data-source edges.

– *Processing of vertices*: Labels of vertices might denote query constants. We use constant(v) to return the label of the vertex v. Also, vertices might stand for variables. Every vertex is therefore also associated with a distinct variable such that var(v) returns the variable representing v. For instance, constant(University) returns University and var(University) returns z.
– *Mapping of A-edges*: Edges $e(v_1, v_2)$ where $e \in L_A$ and $v_2 \neq \textit{value}$ are mapped to two query predicates of the form *type*(var(v_1), constant(v_1)) and e(var(v_1), constant(v_2)). Note that e is an attribute edge label. By definition of the query space, v_1 thus denotes a class and v_2 is a data value. Accordingly, constant(v_1) returns a class name and constant(v_2) returns the value. For instance, name(*University*, *StanfordUniversity*) is mapped to *type*(z, *University*) and name(z, *StanfordUniversity*).
In case $v_2 = \textit{value}$, $e(v_1, v_2)$ is mapped to the predicates *type*(var(v_1), constant(v_1)) and e(var(v_1), var(value)). Note that this is to deal with situations where the keyword matching element is an edge label. The difference to the previous case is that v_2 does not denote a concrete value, and thus is mapped to a variable instead of a constant.

– *Mapping of R-edges*: Edges $e(v_1, v_2)$ where $e \in L_R$ are mapped to three query predicates of the form *type*(var(v_1), constant(v_1)), *type*(var(v_2), constant(v_2)) and e(var(v_1), var(v_2)). Note that since e is an R-edge, v_1, v_2 denote classes. Hence, constant(v_1), constant(v_2) return two class names and var(v_1), var(v_2) return the variables representing some entities of these two classes. For instance, *employment*(*University*, *Person*) is mapped to *type*(z, *University*), *type*(y, *Person*) and *employment*(z, y).

The resulting query is simply a conjunction of all the predicates generated for a query graph. Since conjunctive queries represent a fragment of SPARQL (and SQL), it is straightforward to translate g_{q_i} directly to the query language supported by the local RDF stores (or relational database), cf. the conjunctive query in Fig. 4(b) and SPARQL queries in Fig. 2(a).

If there is no further information available other than keywords, a reasonable choice is to treat all query variables as distinguished to obtain all variable substitutions of a given query. In our system, the user can select the query and after that, can choose the type of entities she is interested in, i.e. choose the distinguished variable manually.

5.4. Query result combination

The results obtained from the local query engines are combined to obtain the final answer for the distributed query. Each result set for a partial query graph g_{q_i} can be seen as a relation R_{q_i}, where a column r_i in R_{q_i} captures bindings for a particular vertex of g_{q_i}. Table 2 shows three relations obtained for our example queries, i.e. R_{q_1}, R_{q_2} and R_{q_3} for $q1$, $q2$ and $q3$ respectively. The relations R_{q_i} are joined along the inter-data-source edges, i.e. $R_{q_i} \bowtie_{e_i(r_i, r_j)} R_{q_j}$, where $e_i \in E_I$ connects r_i (denoting a column in R_{q_i}) with r_j (denoting a column in R_{q_j}). Two types of joins are distinguished in this regard:

If $e_i(r_i, r_j)$ is a class mapping (i.e. r_i and r_j correspond to classes), a similarity join needs to be performed on the entities of r_i and r_j. In order to perform this join more efficiently, entity mappings are pre-computed such that given a class mapping, a two columns "mapping relation" R_m is retrieved from the mapping index. Such a relation contains pairs of entities that have been identified to match based on their similarity. Examples are shown in Table 2, i.e. R_{m5} and R_{m6} for the mappings between person $m5$ and $m6$. Using these results, the similarity join amounts to a two-ways join $R_{q_i} \bowtie_{r_i=r_i} R_M \bowtie_{r_j=r_j} R_{q_j}$ (we refer to as map join):

- the first relation is joined with the mapping relation (on the first entity column),
- and then, the resulting relation is joined with the second relation (on the second entity column).

With respect to our example, the operations $R_{q_1} \bowtie_{r_y=r_{PersonFB}} R_{m5} \bowtie_{r_{PersonDBLP}=r_{y'}} R_{q_2} \bowtie_{r_{y'}=r_{PersonDBLP}} R_{m6} \bowtie_{r_{PersonDBP}=r_{y''}} R_{q_3}$ have to be performed for the computation of the final results. In other words, the similarity join as discussed in literature [17,24] is realized in our approach through two steps: (1) offline computation of mappings and (2) map join that exploits pre-computed mappings. This way, expensive online comparison of entities can be avoided.

This map join concept is also used for the processing of relation and attribute mappings, i.e. $e_i(r_i, r_j)$ connects to attribute or relation vertex r_i and r_j. Technically, a relation mapping $e_i(r_i, r_j)$ can be regarded as two class mappings: $e_{i1}(\text{domain}(r_i), \text{domain}(r_j))$ and $e_{i2}(\text{range}(r_i), \text{range}(r_j))$ that express the correspondences between classes that are the domain of r_i and r_j and the range of r_i and r_j respectively. Accordingly, the processing of a relation mapping breaks down to two map join operations, $R_{q_i} \bowtie_{e_{i1}(\text{domain}(r_i),\text{domain}(r_j))} R_{q_j}$ and $R_{q_i} \bowtie_{e_{i2}(\text{range}(r_i),\text{range}(r_j))} R_{q_j}$. The processing of attribute mappings is similar. However, only one map join operation is needed because an attribute mapping $e_i(r_i, r_j)$ expresses only the correspondence between the domain of r_1 and r_2.

Note that the processing of two intermediate result sets R_i and R_j results in all combinations of tuples in R_i and R_j that are similar on one entity. This captures the intuition that (complementary) information from two data sources should add up. With respect to our example, tuples are joined along entities of the type person. This results in a combination of different information about person, i.e. publication, employment and prizes.

6. Evaluation experiments

We will now discuss experiments we have performed with a system implementing the Hermes infrastructure. The goal of the experiments is to show the performance and effectiveness of our system with real life data sets available on the data Web.

6.1. Evaluation setting

6.1.1. The Hermes system

Hermes is realized as a Web application, publicly accessible at http://hermes.apexlab.org. The application provides a Flash-based user interface. (Fig. 5 shows the screenshot of the interpretation of a keyword query as a query graph.) The core of Hermes, composed of the query disambiguation and of the distributed query processing backends runs on the server part.

In the implementation, we used Lucene for the management of the keyword and the mapping index, the BerkelyDB for the structure index (and cached in memory), and for local query processing we employ Semplore. Semplore is an RDF store supporting conjunctive queries on graph data. The graph data indices we created are in fact inverted indices, which is used by Semplore both for storage and retrieval [29]. The use of Semplore follows recent trends in managing large amount of Web data. Work from Semantic Web [29] as well as database research [8] has shown that the inverted index is a viable choice for indexing (RDF) triples.

We have tightly integrated Semplore into our distributed query engine to minimize the communication overhead for query routing and result combination.

In the current setup (which is also the basis for the experiments described in the following), the Web data sources are logically separate, but all components physically run on a single node: a Linux Server with 64Bit DualCore 2×3 GHz Intel Xeon processor and 8 GB of memory. All internal indices are stored on a Hitachi SATA 1TB hard drive.

6.1.2. Data

For the experiments, we used SwetoDBLP, DBpedia, Freebase, USCensus, GeoNames, semanticWeb.org,[7] and the AIFB portal data[8] – all of them are publicly available in RDF. While DBpedia and Freebase are general knowledge bases, other sources are about specific domains. USCensus contains census statistics for the United States, GeoNames provides geographic features, semanticWeb.org captures information about the Semantic Web community, and the AIFB portal contains data about the research group that organizes the ISWC 2008. Detailed statistics for each dataset can be found in Table 3. In total, the indexed data adds up to 1.1Bio triples.[9] For this

[7] http://semanticWeb.org/.

[8] http://www.aifb.uni-karlsruhe.de/about.html.

[9] Please note that the indexed data contains additional triples over the originally published data sets.

Table 2
Intermediate result sets.

q1 on Freebase			m5		q2 on DBLP		m6		q3 on DBpedia		
Name	z	y	fb:Person	dblp:Person	y'	x	dblp:Person	dbp:Person	y''	z	Label
Stanford	uni1	per1	per1	per2	per2	pub1	per2	per3	per3	prize1	Turing Award
Stanford	uni2	per8	per8	per9	per9	pub2	per9	per7	per3	prize2	Turing Award

Table 3
Statistical information of the data sets.

Data set	#Tripels	#Instances	#Categories	#Relations	#Attributes	#Literals
SwetoDBLP	14,936,600	1,644,086	10	12	17	12,654,821
DBpedia	110,241,463	19,238,235	175,920	12,240	28,216	14,187,352
Freebase	63,069,952	7,517,743	814	917	1099	34,451,000
USCensus	445,752,172	82,702,188	8	496	1373	12,033
GeoNames	69,778,255	14,051,039	1	10	6	57,518
SW.org	67,495	22,682	506	515	190	13,845,576
AIFB	19,271	2991	22	16	21	1,001,976

data, mappings have been computed through the iterative process we explained before. For the evaluation, ca. 2Mio data-level mappings and 1500 schema-level mappings are indexed in the system. We observe that DBpedia and Freebase are connected most tightly as more than 90 percent schema-level mappings and half of the data-level mappings are found between the two data sources.

6.1.3. Queries

We collected keyword queries that have been asked by real users (20 colleagues in our lab) against the system. For the evaluation, we restricted to queries that in principle should be answerable given the data described above. In total, we selected 20 keyword queries where 10 of them (Q11–Q20) can only be answered by combining multiple data sources. We show the keywords and the relevant data sources for these queries in Table 4. For instance, the intended interpretations of the keywords for Q10 and Q11 are: (Q10) find films about "Titanic" and (Q11) find research topics of Rudi Studer, the local chair of ISWC 2008.

6.2. Data preprocessing

Table 5 shows statistics of the size of the data sets, the size of the pre-computed indices and the time to build the indices. For reason of space, we only show the combined time for building all four indices (keyword, mapping, structure, and data graph index). The total time for building all indices for all the 7 data sources amounts to 59 hours (note that throughout the experiments, we use a single machine). The size of the structure index is much smaller than

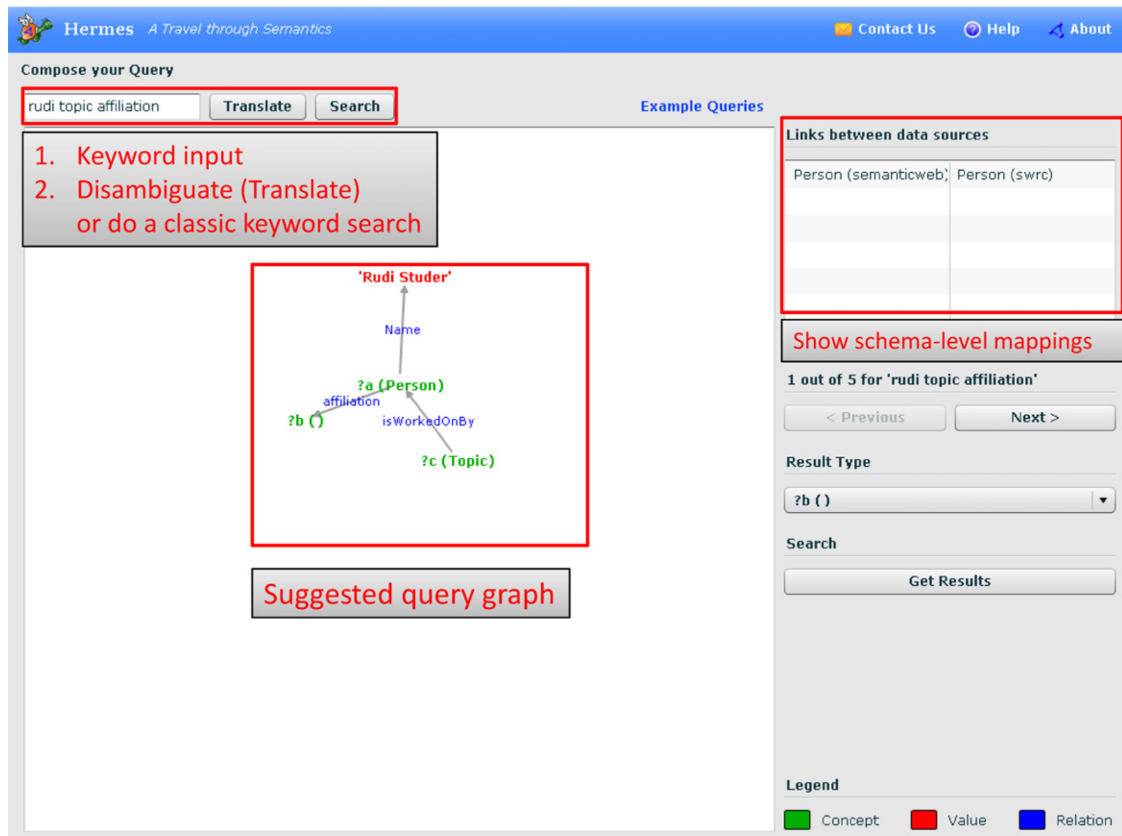

Fig. 5. Hermes query disambiguation interface showing a suggested query graph.

Table 4
Example queries.

#	Keywords	Data sources
Q1	Project, ISWC, person	semanticWeb.org
Q2	Studer, publication	semanticWeb.org
Q3	Undergraduate, topic	semanticWeb.org
Q4	Rudi, proceedings	semanticWeb.org
Q5	Track, 323	Freebase
Q6	Pinocchio, film	Freebase
Q7	Company, owner, "shopping center"	Freebase
Q8	Restaurant, Berlin	Freebase
Q9	"the day after tomorrow", director	Freebase
Q10	Film, Titanic	Freebase
Q11	ISWC2008,Studer, topic	SwetoDBLP, SW.org, semanticWeb.org
Q12	Person, Shanghai, town	Freebase, USCensus
Q13	Ronny Siebes, InProceedings	SwetoDBLP, semanticWeb.org
Q14	Tom, iswc2008, proceedings	SwetoDBLP, SW.org, semanticWeb.org
Q15	Lake, citytown, wood	Freebase, USCensus
Q16	Person, town, village	Freebase, USCensus
Q17	Restaurant, german	Freebase, USCensus
Q18	Album, town, mountain	Freebase, USCensus
Q19	Frank, publications	SwetoDBLP, semanticWeb.org
Q20	Markus, report	SwetoDBLP, semanticWeb.org

indices built for the data graphs. Thus, we cached them in memory to enable faster query translation. The overall size of all indices is 42.4GB.

6.3. Keyword translation

6.3.1. Efficiency

Fig. 6 illustrates the average time for translating the 20 keyword queries to the top-5 conjunctive queries. The overall time breaks down to two parts: time for keyword mapping and time for top-k query construction (which includes query space construction and top-*k* search).

Expectedly, more time is needed for top-k query construction when performed on larger schema graphs. For instance, query construction is much slower for Q5 than Q13, as Q5 is asked against Freebase, which has a much larger schema graph than the integrated graph computed from DBLP and AIFB used to process Q13. From another perspective, this example also indicates that computing queries that span multiple data sources is not much different. In fact, the integration of the two schema graphs is insignificant when compared with the exploration time. It is important to mention that schema graphs cached in memory are used for the experiments. This is affordable (even for a larger number of data sources) as they are relatively small in size.

As further discussed in Section 7, Semantic Web search engines like Sindice, Watson, Swoogle Falcons essentially provide lookup functionalities based on an IR engine, such as Lucene. Using this underlying engine, keywords submitted by the user are matched against the data stored in the inverted index. Note that this corresponds exactly to the keyword mapping step we perform during keyword translation. We compare the time for this step with the total query translation time in order to get some preliminary comparative results in terms of performance. Keyword mapping makes up 25 percent of the overall time on average while it exceeds more than 50 percent in some cases (e.g. Q11 and Q13). In particular, more time is required if the keywords are popular such that it maps to a large number of elements in the keyword index. This is the case for Q7 and Q11. Both the *company owner* and *shopping center* keywords in Q7 result in a large number of keyword elements while the keyword *topic* in Q11 returns a long list of candidates. All 20 queries can be translated within 2 seconds. In all, the results indicate that when compared to keyword lookup, the computation of full interpretation requires additional time, which is affordable for most queries.

6.3.2. Effectiveness

In order to assess the effectiveness of query translation, we adopted a standard IR metric called Reciprocal Rank (RR) defined as $RR = \frac{1}{r}$, where r is the rank of the correct query, i.e. the query matching the intent of the user. If none of the generated queries is correct, RR is simply 0. We invited the users to identify the intended interpretation from the list of top- 5 queries. The average MRR for the 20 queries are shown in Fig. 7. The results indicate that the

Table 5
Size and building time of internal indices.

Data source	Number of triples in millions	Index size in MB				Index time (s)
		I_k	I_m	I_s	I_{dg}	
SwetoDBLP	19	1060	2.22	0.02	655	6922
DBpedia	247	2630	42.7	75	7522	50,946
Freebase	89	1590	48.4	29	2268	19,536
USCensus	694	980	4.1	0.01	18,948	84,795
GeoNames	132	4110	0	0.006	3431	51.314
SemWeb.org	0.17	7.35	1.2	4.2	5	83
AIFB	0.04	3.04	0.022	0.08	1.4	13

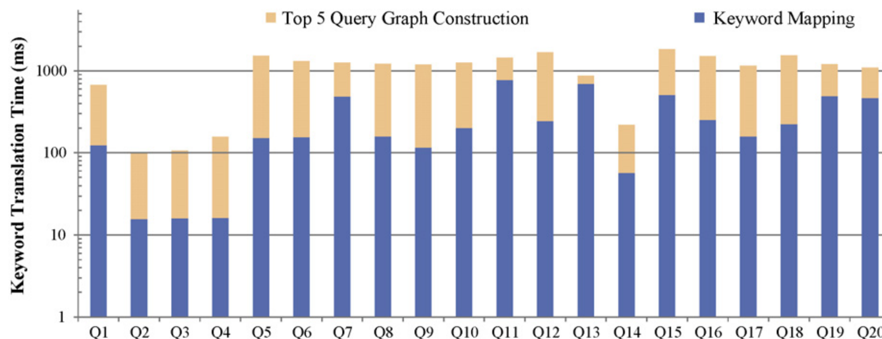

Fig. 6. Average keyword translation time.

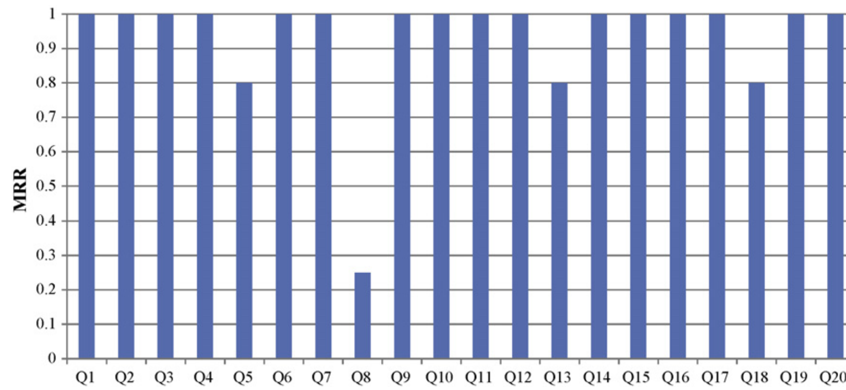

Fig. 7. Mean reciprocal rank of top-5 queries.

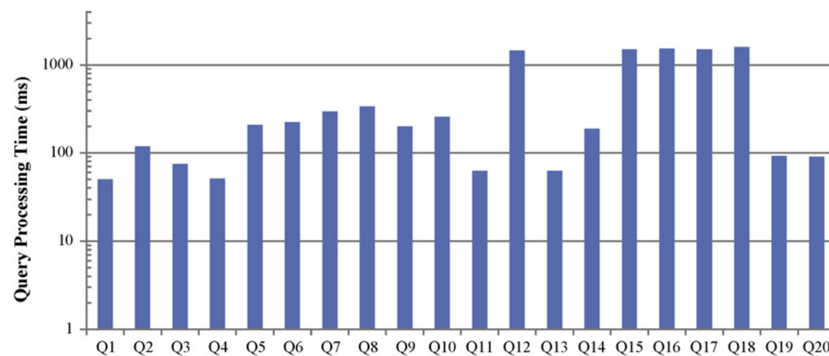

Fig. 8. Average query evaluation time.

intended interpretation could be computed for all queries and also, most of them are ranked correctly, i.e. at first position.

6.4. Distributed query processing

In addition to the time for query translation, we also record the time to evaluate the query selected by the user. The average query evaluation time (10 times for each query) for those 20 queries is shown in Fig. 8. The average query processing times for single-data-source queries (Q1–Q10) is around 1s. Multi-data-source query processing (Q11–Q20) requires more time on average, with the maximum time being within a boundary of 1.5 seconds. Expectedly, queries containing predicates that result in large sets of results spanning over several data sources are harder to process. Q18 is such an example which contains *Album* and *Town*. Each of these predicates results in several thousands of instances. This and other queries such as Q12 and Q15–Q17 run slower, when compared to the single-data-source queries Q1–Q10.

To better understand the performance of our method of distributed query processing, total times for processing the multi-data-source queries is further decomposed into two components: (1) local join processing: we measured the number of joins that have to be performed on the data tuples during local query processing and the total time needed for these operations (2) result combination across data sources: we measured the number of map joins that have to be performed to combine results from different data sources and the total time. Note that local query processing is the process of answering data graph specific query parts. The results to these queries have to be combined, typically via similarity joins [17,24]. Since we leverage precomputed mappings instead of computing similarities online, there is no fair comparison with these approaches. We therefore compare our approach for result combination using map join with standard join processing.

From Table 6, we observe that with most queries, the total time for map join is only half the total time for local join, even though the number of joins involved is almost the same for both. This clearly supports our claim that using precomputed mappings in the index, result combination can be performed very efficiently. In this experiment, it is actually even faster than local query processing. We have investigated this positive result and found out that retrieval of mappings is relatively fast. However, the difference is mainly due to the implementation of join processing. We use hash join for result combination, which is faster the the mass union of posting lists employed by Semplore for join processing [29]. Overall, the results suggest that like standard join processing, distributed query processing using map joins results in affordable response time. Clearly, the potential for parallelizing some operations during this process offers room for future investigation.

To the best of our knowledge, there is no system that offers the capability of Hermes in the context of data Web search. Thus, it is difficult to carry out a comparative study. The evaluation results discussed above however indicate that Hermes exhibits affordable time and space requirements for building indices, can effectively

Table 6
Detailed join information for Q11–Q20.

Queries	Local join (ms)	Map join (ms)	#Local joins	#Map joins
Q11	42.47	19.53	2	2
Q12	956.26	502.74	9	4
Q13	47.54	14.46	7	3
Q14	117.92	69.08	83	82
Q15	1140.6	359.4	301,721	301,710
Q16	1042.91	489.09	2,064	2,063
Q17	1071.28	429.72	16,371	16,370
Q18	1179.17	419.83	23,394	23,385
Q19	70.6	21.4	8	3
Q20	62.47	27.53	874	873

translate keywords to structured queries and also, offers acceptable response time for processing structured query spanning across data sources. The overall system can scale to a realistic data Web search scenario.

7. Related work

There exist several dimensions of related work. We structure our discussion along the presentation of our contributions: (1) infrastructures for data Web search, (2) keyword query translation, and (3) federated query processing.

7.1. Infrastructures for data Web search

In our architecture, we follow the paradigm of pay-as-you-go of data spaces, which previously has been applied successfully to personal information management and enterprise intranets [23]. The application of the pay-as-you-go paradigm to Web scale data integration has been proposed – on a conceptual level – in [19]. To our knowledge, Hermes is the first realization of an infrastructure that enables integration and search over an open set of real life Web data sources.

Other approaches provide a more centralized paradigms to integration on the data Web. For example, Freebase[10] implements a centralized, albeit open and community-based Web database. Data from other Web data sources is copied into Freebase in a controlled way. The problem of heterogeneity is alleviated by a centralized, manual integration and reconciliation effort (gardening).

Recently, a number of Semantic Web search engines (lookup indices) have been developed, including Falcons [4], Sindice [20], Swoogle [7] and Watson [6]. These engines focus on indexing and providing keyword-based lookup services, rather than an integration and search over multiple Web data sources.

7.2. Keyword translation

Existing approaches to data Web search either support expressive queries based on structured query languages or keyword search. For example, Freebase supports an expressive query language called MQL. Semantic Web search engines such as Swoogle and Watson offer keyword-based lookup services. While they are simple to use, the expressivity of this keyword search is rather restricted. Through the translation of keywords to structured queries, we offer more advanced querying capability.

The problem of keyword queries on structured data has been studied from two different directions: (1) computing answers directly through exploration of substructures on the data graph [13,16] and (2) computing queries through exploration of a query space [28]. It has been shown in our previous work [28] that keyword translation operates on a much smaller query space, and is thus efficient. Besides, the structured queries presented to the user help in understanding the data (answer) and allow for more precise query refinement. We follow the second direction to keyword search and extend our previous work to a multi-data-source scenario. We have discussed the main differences to the previous work throughout the paper and will summarize as follows: Instead of a single data source, we have extended the keyword and the structure index to deal with an integrated set of data sources. A dedicate mapping index is proposed to manage links between data sources. The top-k query search algorithm is adopted for the exploration of a query space that may span over multiple data sources. For a more guided exploration, the previous ranking scheme has been refined to cope with the many levels of uncertainty that are specific to keyword search in a multi-data-source scenario.

7.3. Ranking

Ranking has been studied in many contexts. An common measure for ranking is "popularity", which is widely adopted by the IR community. It is captured through the PageRank [3] concept. Recently, much work has been devoted to adopting this PageRank concept to relational data [1] and RDF data [26,14]. As discussed, the edges between (RDF) data elements have different semantics. As a result, the effectiveness of PageRank heavily depends on the weights assigned to different edge types—a task that requires (upfront) manual effort that is not suitable for the large-scale data Web setting. Ranking is also an essential concept in Semantic Web search engines like Sindice, Watson, Swoogle and Falcons. Essentially, these systems provide lookup functionalities based on an IR engine, such as Lucene. The IR engine is used to index ontologies, and the containing semantic data. Keywords submitted by the user are then matched against the indexed resources, where results are ranked according to the matching scores returned by the IR engine. In systems like Sindice, some additional ad-hoc rules are applied on top, e.g. "prefer data sources whose hostname corresponds to the resource's hostname" [21]. More systematic approaches for ranking have been studied for keyword search on data bases [13,15,12]. In Section 4, we have already summarized the main measures employed by these approaches, i.e. matching score, popularity and length. In this regard, we have introduced a special notion called EF/IDF to combine popularity with distinctiveness, and argued that additional factors such as matching score and coverage are required for more effective keyword search in the data Web scenario.

7.4. Federated query processing

For dealing with Semantic Web data, Refs. [18,22] have developed distributed infrastructures for RDF data sources. In [22], the authors proposed optimization techniques for join ordering, which we also employ in our query planning. Yet these works do not take the problem of heterogeneity into account, neither on the schema-level nor on the data-level. In our work, we propose a procedure for iterative integration that compute mappings between pair of data sources as needed.

We make use of data-level mappings to perform similarity joins. Typically, the processing of similarity joins [17,24] involve an expensive computation of similarities. In our approach, we simply retrieve the mappings from the index to perform standard joins over the resulting mapping relation (map join).

8. Conclusions

We have presented Hermes, an infrastructure for search on the data Web. In the realization of Hermes, we have presented a number of original contributions: We have proposed a novel technique for translating user keywords to structured queries against heterogeneous Web data sources. Further, we have designed a number of indices that are needed in order to realize efficient search over the data Web. Finally, we have elaborated on techniques for distributed query processing on the data Web, including a map join procedure that allows efficient combination of results from heterogeneous sources by exploiting pre-computed mappings.

The evaluation experiments clearly show the feasibility and usefulness of the approach. Both the translation of keywords and the processing of queries can be performed in near real time on a standard machine. At the same time, the quality of the interpretation of the user information needs works promisingly effective.

[10] http://www.freebase.com/.

While the two billion RDF triples available through LOD are already an amazing playground, this amount of data is still by orders of magnitude smaller than today's Web of documents. Still, given the added value that can be provided once a critical mass exists, we expect the amount of data to explode in the coming years. While today we are still able to handle the significant part of the Web of data on a single machine, Hermes is ready to scale, e.g. by deploying it on a cloud computing infrastructure.

References

[1] A. Balmin, V. Hristidis, Y. Papakonstantinou, Object rank: authority-based keyword search in databases, in: VLDB, 2004.
[2] C. Bizer, T. Heath, K. Idehen, T. Berners-Lee, Linked data on the web, in: Proceedings of the 17th International Conference on World Wide Web, WWW, 2008.
[3] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Comput. Net. 30 (1–7) (1998) 107–117.
[4] G. Cheng, W. Ge, Y. Qu, Falcons: searching and browsing entities on the semantic web, in: Proceedings of the 17th International Conference on World Wide Web, WWW, 2008.
[5] N. Choi, I.-Y. Song, H. Han, A survey on ontology mapping, SIGMOD Rec. 35 (3) (2006) 34–41.
[6] M. d'Aquin, C. Baldassarre, L. Gridinoc, S. Angeletou, M. Sabou, E. Motta, Characterizing knowledge on the semantic web with watson, in: Proceedings of the 5th International Workshop on Evaluation of Ontologies and Ontology-based Tools, EON, 2007.
[7] L. Ding, T.W. Finin, Boosting semantic web data access using swoogle, in: M.M. Veloso, S. Kambhampati (Eds.), AAAI, AAAI Press/The MIT Press, 2005.
[8] X. Dong, A.Y. Halevy, Indexing dataspaces, in: SIGMOD Conference, 2007.
[9] D.W. Embley, L. Xu, Y. Ding, Automatic direct and indirect schema mapping: experiences and lessons learned, SIGMOD Rec. 33 (4) (2004) 14–19.
[10] J. Euzenat, P. Shvaiko, Ontology Matching, Springer-Verlag, Heidelberg, DE, 2007.
[11] R. Goldman, J. Widom, Dataguides: enabling query formulation and optimization in semistructured databases, in: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, 1997.
[12] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram, Xrank: ranked keyword search over xml documents, in: SIGMOD Conference, 2003.
[13] H. He, H. Wang, J. Yang, P.S. Yu, BLINKS: ranked keyword searches on graphs, in: C.Y. Chan, B.C. Ooi, A. Zhou (Eds.), Proceedings of the 2007 SIGMOD International Conference on Management of Data, ACM, 2007.
[14] A. Hogan, A. Harth, S. Decker, Reconrank: a scalable ranking method for semantic web data with context, in: Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 2006.
[15] V. Hristidis, L. Gravano, Y. Papakonstantinou, Efficient IR-style keyword search over relational databases, in: VLDB, 2003.
[16] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, H. Karambelkar, Bidirectional expansion for keyword search on graph databases, in: K. Böhm, C.S. Jensen, L.M. Haas, M.L. Kersten, P.-Å. Larson, B.C. OOi (Eds.), Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), ACM, 2005.
[17] D.V. Kalashnikov, S. Prabhakar, Fast similarity join for multi-dimensional data, Inform. Syst. 32 (1) (2007) 160–177.
[18] A. Langegger, W. Wöß, M. Blöchl, A semantic web middleware for virtual data integration on the web, in: Proceedings of the 5th European Semantic Web Conference, ESWC, 2008.
[19] J. Madhavan, S. Cohen, X.L. Dong, A.Y. Halevy, S.R. Jeffery, D. Ko, C. Yu, Web-scale data integration: you can afford to pay as you go, in: Proceedings of the 3rd Conference on Innovative Data Systems Research, CIDR, www.crdrdb.org, 2007.
[20] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello, Sindice.com: a document-oriented lookup index for open linked data, Int. J. Metadata Semantics Ontol. 3 (1). http://www.sindice.com/pdf/sindice-ijmso2008.pdf.
[21] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello, Sindice.com: a document-oriented lookup index for open linked data, Int. J. Metadata Semantics Ontol. 3 (1) (2008) 37–52.
[22] B. Quilitz, U. Leser, Querying distributed RDF data sources with SPARQL, in: Proceedings of the 5th European Semantic Web Conference, ESWC, 2008.
[23] A.D. Sarma, X. Dong, A.Y. Halevy, Bootstrapping pay-as-you-go data integration systems, in: J.T.-L., Wang (Ed.), SIGMOD Conference, ACM, 2008.
[24] E. Schallehn, K.-U. Sattler, G. Saake, Efficient similarity-based operations for data integration, Data Knowl. Eng. 48 (3) (2004) 361–387.
[25] A.P. Sheth, J.A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, ACM Comput. Surv. 22 (3) (1990) 183–236.
[26] J. Stoyanovich, S.J. Bedathur, K. Berberich, G. Weikum, Entityauthority: semantically enriched graph-based authority propagation, in: WebDB, 2007.
[27] T. Tran, P. Cimiano, S. Rudolph, R. Studer, Ontology-based interpretation of keywords for semantic search, in: Procedings of the 6th International Semantic Web Conference (ISWC'07), 2007.
[28] T. Tran, H. Wang, S. Rudolph, P. Cimiano, Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data, in: ICDE, IEEE, 2009.
[29] L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, Y. Yu, Semplore: an IR approach to scalable hybrid query of semantic web data, in: K. Aberer, K.-S. Choi, N.F. Noy, D. Allemang, K.-I. Lee, L.J.B. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, C.-M. (Eds.), ISWC/ASWC, Lecture Notes in Computer Science, vol. 4825, Springer, 2007.