Dealing with the Deep Web and all its Quirks

Meghyn Bienvenu
CNRS; Université Paris-Sud
meghyn@lri.fr

Daniel Deutch
Ben Gurion University
deutchd@cs.bgu.ac.il

Davide Martinenghi
Politecnico di Milano
martinen@elet.polimi.it

Pierre Senellart
Institut Mines–Télécom
Télécom ParisTech; CNRS LTCI
pierre.senellart@telecom-paristech.fr

Fabian Suchanek
Max-Planck-Institute for Informatics
fabian@suchanek.name

ABSTRACT

Several approaches harvest, query, or combine Deep Web sources. Yet, in addition to well-studied aspects of the problem such as query answering using views, access limitations, or top-*k* querying, the Deep Web exhibits a number of peculiarities that are often neglected. First, the services usually deliver not all results, but only the *top-n results* according to some ranking function. This function may not be compatible with the ordering specified in a user's query. Subsequent results have to be obtained by paging, or may not even be accessible. Second, the services may deliver results in a *granularity* that is incompatible with the query or joinable services (e.g., months vs. exact dates). Moreover, the services may perform selections or ranking over attributes that are not exposed in the results: this poses an *incompleteness* problem. Additional challenges come from uncertainty, recency constraints, and inter-service dependencies. In this article, we shed light on these peculiarities, and compile a list of desiderata of a query answering system for the Deep Web.

1. INTRODUCTION

A wealth of structured information lies in the *Deep Web* [11], the part of the Web accessible only by submitting Web forms or using Web services. A simple example is hotel information (availability, category, price, user rating) that requires accessing Web sites of travel agencies or individual hotels. Realtor Web sites are another example. They offer apartments for sale, and display apartments that match user-specified criteria. The sites also show the apartment properties, such as ranking, size, location, etc. The structured character of the information means that query answering over the Deep Web can be much more elaborate than on the *Surface* Web.

Because of this potential, many research works have studied problems relevant to the Deep Web. In addition to work on form understanding and wrapper induction [11], a number of aspects of query answering over the Deep Web have been investigated: in particular, answering queries using views, limited access constraints, top-*k* ranking (see Section 5 for an overview of the literature). For the most part, the emphasis has been on building clean theoretical models of Deep Web services over which query answering is studied. We pinpoint in this work a number of annoying *quirks* of actual Deep Web sources that do not fit well into these models. We claim that these quirks significantly impact query answering and that they need to be taken into account by any practical system for querying Deep Web sources. Let us illustrate with two examples.

VLDS'12 August 31, 2012. Istanbul, Turkey.

Choosing hotels for an academic conference is a non-trivial task. In particular, for conferences with a large number of participants, the organizers may wish to choose for the conference attendees two highly-rated hotels, which are located close to each other so that all participants can easily attend talks. Of course, the hotels should have availability for the conference dates, and room prices should not exceed some threshold. All these constraints can be expressed as a query over a virtual schema, with services such as hotel Web sites, mapping services, and travel sites modeled as limited-access views over this schema. However, existing services do not readily lend themselves to finding an optimal rewriting (or even a single rewriting) of the query using the views: a proximity service may only return the *n* hotels closest to a given hotel, with no possibility of going beyond this limit if none of them meet the other ranking criteria; a Web site listing hotel recommendations may allow for the hotels to be ranked by ratings, but may not display the actual ratings; there might be many different ways of accessing the information of a given source (different ranking criteria, different limits on the number of items per page, etc.) that all expose different pieces of information, making the choice of the optimal rewriting with respect to a query non-trivial.

As another example, consider the tedious process of looking for an apartment. To be effective, this typically involves a search through multiple real-estate owners, sorting through different criteria such as price and location, and going through many options until finding the most relevant ones. *Recency* of data is an important issue in this case, since only recent real-estate ads are typically relevant. Furthermore, because the search for an apartment is a repetitive process, we only want to access each day or each week the new offers for the day or week. How is one to know whether an offer is more recent than another, if only the month is displayed in a posting, i.e., the granularity of the attribute is not the same as expected? How can we make use of the results of the query from last week to better answer that from this week? If we are looking for an apartment that is both recent and cheap on one given Web site, should we ask for a ranking by date (but then, obtaining the price of each posting may require another service call) or one by price (but then, we may hit the limit of the number of ads we are allowed to retrieve per query without getting all the most recent ones)? Consider also that the precise ranking functions used in these services may be uncertain, since we do not have access to the exact definition of the services.

We envision a declarative framework for querying the Deep Web, which will encompass all these peculiarities. The idea is that queries would be written in a simple, SQL-like language, which would also serve to formally describe Web services, including access constraints, ranking mechanisms, granularity mismatches, paging limits, the total number of results returned, etc. The goal of the envisioned framework would then be to find a "query plan," i.e., a way to compose queries to the Web services along with "local" data manipulations (selections, reordering, etc.), so as to realize the user query. This query plan should ideally be *optimal* in some sense (with respect to either the actual number of network requests or amount of data exchanged, or an estimation of this).

2. FEATURES OF THE DEEP WEB

In this section, we describe the main components and features that we envision for a query answering system for the Deep Web.

The primary goal of such a system is to unveil the data that are not directly accessible as Web pages, but that rather lie behind Web forms. To this end, one must at least be able to represent Web sources in the same way as they are actually implemented, where selection conditions may be applied on the data, along with projections on fields of interest. In more complex scenarios, forms may even use basic join or union operations for combining different Web sources, or simply different relations of a given source. These serve to define services as **views** but obviously the same features are expected of the query language. Therefore, a view and query language for the Deep Web must include **selection** (including in particular **inequality** predicates), **projection**, and, possibly, **join** and **union**. On the other hand, other features such as negation or recursion are more infrequently used in in the context of Deep Web sources and may be left out.

Web forms are typically presented as a collection of input fields, checkboxes, drop-down lists, and other selection elements, some of which may be mandatory. They act as an interface that specifies all possible access patterns to the underlying data, and protects them from unwanted access. An **access pattern** is basically a specification of an access mode for every data field, i.e., it indicates which fields are used as input and which ones are used as output. Query answering under access patterns is more complex than in the classical case and requires special care.

Accessing data in the Deep Web is costly, due, e.g., to latency in network access. Worse, data access is hindered by a number of other limitations than commonly occur and may make querying even more expensive. Some data sources only provide their records in batches of fixed size (**pages**). Some only grant access to the **top-*n* records**, according to some **ranking function**, and hide the remaining ones. In some other cases (typically, for Web services with an API), only a **limited amount of accesses** per time period (say, hour or day) is allowed. In addition to sometimes affecting the possible query rewritings, all these aspects require planning an execution strategy that minimizes the incurred cost (or keeps it within an available budget) while satisfying the user's requirements.

Some sources may even give **incomplete information**, e.g., by presenting only a subset of the underlying fields (some of which are then dismissed via projection) or by delivering data at a different level of **granularity** than what is required by other sources (e.g., months vs. exact dates).

When users pose a query to a Web source, they typically have in mind a criterion for ordering the results. This may or may not be compatible with the order in which results are stored in the source and collected from it. Often, users are only interested in the best results that satisfy their query, as is the case in the field of **top-*k* queries**. In such cases, a query answering system should devise strategies for efficiently retrieving the best results according to the users's criterion and avoiding access to data that are not needed to compute the top results.

As illustrated by the running example, **recency** is often an important criterion in delivering relevant results to user queries. When the output of Web services comes with temporal information, we can treat recency similarly to other ranking criteria (modulo the granularity issue mentioned above). However, it may often be the case that data items are sorted with respect to a temporal order, but no explicit temporal information is provided in the results. In this case, determining recent items may still be possible, but requires some inference (e.g., a comparison with cached results from previous calls to the Web service).

When users specify their ranking criteria through a scoring function, this induces a total order over the query results. However, if the scores are uncertain, due to, e.g., **uncertainty** in the data or in the scoring function, the resulting order is only partial, and the semantics of a top-*k* query becomes unclear. Uncertainties of different kinds may occur. Membership uncertainty is when a record is present with a given probability originating from, e.g., the reliability of the data source in data integration environments, or similarity measures in approximate-matching. Uncertainty in data correctness refers to the probability that the whole tuple gives correct information. Value uncertainty represents attributes as probability distributions on continuous or discrete domains of possible values.

Services of the Deep Web have **dependencies** (functional, inclusion, exclusion) on each other, especially when one particular source is accessible through different services. They need to be taken into account in query planning and optimization.

Typically, users wish to browse through results that not only meet their preferences but are also diverse. Readability of results also requires that different entries that refer to the same entity be merged. Therefore **diversification** and **deduplication** of results are significant aspects of query answering in the Deep Web. In addition, users might be interested in the **provenance** of results of their queries, e.g., with an indication of the sources they came from, but they might also wish to have an **explanation** for the result set. For example, a query might return an empty answer either by contingency of the underlying data instance, or by necessity, due to a misconception of the user about the data sources. In the latter case, the user should be informed. Explanation can also serve other purposes, in particular explaining why certain answers were given as results; this is especially crucial in the presence of uncertainty.

3. MODEL AND LANGUAGE

We envision a framework where users can write queries on Deep Web data in some declarative language, which is then compiled into a query plan that essentially composes accesses to Web services and manipulations of retrieved data. We next describe some desiderata of such a declarative language, then exemplify how queries in such a language would look.

3.1 Desiderata

We identify the following as desiderata of a query language and a supporting framework for it, in this context.

1. The query language should be *declarative*, i.e. users should specify the result they are interested in, independently of the concrete way in which it will be retrieved (referred to as "query plans").

2. In addition to the user specification of desired results, the query language should allow one to *express the available Web services* along with their access patterns.

3. The query language should be accompanied with some *cost model*, to distinguish between different query plans. This also calls for the development of a query optimizer, finding and executing optimal plans w.r.t. the cost model.

4. Last, we believe that incremental maintenance of results should be at the core of such a framework. In particular, results of previous queries can be used as the input to further queries, interpreted as "virtual" services.

We next exemplify how these desiderata could be accomplished in a query language, using some simple syntax.

3.2 Example Syntax and (partial) Semantics

We exemplify the principles of the proposed approach using a simple SQL-like syntax, with some special adornments. These pertain to the different properties of access patterns (sorting, restriction to *k* results, granularity, access restrictions), as well as the cost associated with each access. This syntax allows us to uniformly specify both the Web services and the user queries

EXAMPLE 1. *Consider a Web Service of an online travel agency, allowing users to view hotels in the city of their choice, ordered by their user ratings. In an SQL-like syntax, the service functionality can be expressed as follows:*

```
CREATE VIEW HotelsService1($c,$o) AS
SELECT name, city, price, AvailableRooms,
       rating, DAY(LastUpdate)
FROM Hotels1
WHERE city=$c
ORDER BY rating DESC
LIMIT $o,10
```

c and *o* are parameters of the service (called *HotelsService*1). *c* intuitively stands for the city chosen by the user and *o* stands for the number of results to appear in a single request to the service.

This service definition (using the `CREATE VIEW` operator) exemplifies several quirks of the Web service. First, the *number of results* is naturally limited. Second, the results are *ordered* by rating. This means that if users are interested in hotels in particular location (and willing to get results with low rating), many accesses to this service may be required. However, there is some implicit *cost model* associating an additive cost with each result returned by the service (in a more realistic scenario, it should be possible to define more explicit cost functions associated with a service, via language extensions). Third, *recency* is an important issue: rooms availability varies quickly and only recent updates can be used for that. Fourth, the dates of updates are in *granularity* of days – if users are interested in availability updates of greater granularity (e.g. hours), this may require the use of a different service.

In general, we may have many services providing relevant data. For instance, a different service (say, *HotelsService*2) may be expressed as a different query, allowing one to sort results by price etc. Finally, we may also have an available map service, allowing users to look for hotels in a particular area (represented as a single point on the map and allowed distance from this point):

```
CREATE VIEW MapService($locX,$locY,$radius, $o) AS
SELECT name, HotelLocX,HotelLocY,
square(HotelLocX-$locX) + square(HotelLocY-$locY) As SqrDist
FROM GeoDB
WHERE SqrDist < square($radius)
ORDER BY SqrDist ASC
LIMIT $o,10
```

Now, consider conference organizers looking for two good hotels that are located at most 1000 meters from each other in Istanbul. The following query can express this:

```
SELECT Hotels1.name, Hotels2.name
FROM (HotelsService1+HotelsService2+MapService) As Hotels1,
(Hoteslservice1+HotelsService2+MapService) As Hotels2
WHERE Hotels1.city= "Istanbul" AND Hotels2.city="Istanbul"
AND Hotels1.rating > 4
AND Hotels2.rating > 4
AND square(Hotels1.HotelLocX-Hotels2.HotelLocX) +
    square(Hotels1.HotelLocY-Hotels2.HotelLocY)
    < 1000
```

Note the use of the novel "+" syntax for combining the 3 Web services in the above query. The obtained expression is then treated in the query as a regular relation, and in particular we perform a join over two instances of it. The intuition is that, in order to realize these relations, concrete query plans may perform any combination of accesses to the three services, e.g., by matching attributes by name as in a natural join (and choosing appropriate values for the parameters, decided as part of the plan). The query optimizer needs to compute an optimal (e.g., w.r.t. the number of requests) plan of requests to the three services in order to realize the user query.

4. PROBLEMS TO STUDY

To give a flavor of the proposed research directions, we next describe a few fundamental problems that are of interest in this context. Naturally, as the research advances, further interesting problems are likely to arise.

The first challenge in this respect is the design of a *formal language* for specifying the possible *evaluation plans*. Such a language should allow one to express:

- accesses to available Web services, along with the corresponding bindings, and access strategy with respect to paging;
- local computations, such as projection, reordering of results, limits, etc.;
- ordering and combination of services.

Given a query in our language, the most basic question of interest is whether or not the query is *realizable* given the available services and the access patterns associated with them, and if so, to return (all) such plans. For instance, consider our running example of looking for two hotels that are close to each other and have room availability. Realizing this query given two Web services, one for hotels and the other a mapping service, may be non-trivial. Here, one possible plan involves searching, using the map service, for pairs of hotels that satisfy the distance criterion. Then, we may look for the hotel availability and rating to see which of the pairs of hotels constitute a match.

The existence of a query plan may be of interest by itself. However, in presence of multiple plausible plans, users may be interested in finding (top-*k*) *best plans* according to the cost of the involved services, in addition to "standard" database criteria such as selectivity. For instance, in our example another plausible solution is to start with hotels that satisfy the availability and rating criteria, and then look for them on the map. This may be a better solution based, e.g., on the expected selectivity.

We further note that this optimization may either be performed in a *static* manner (each plan is assigned a cost when constructed, depending on the services and operators used), or in a *dynamic* one based on partial execution and constant reevaluation of cost.

5. RELATED WORK

Computing answers to queries using materialized views is a classical problem [10, 15] that consists in finding a rewriting of a query by using the views. This problem naturally arises in many contexts, such as data integration, where views are needed to describe the relationship between the mediator and the data sources, and query optimization, where the rewriting can yield a more efficient query plan. Query answering using views lays the basis for addressing queries over relations with access patterns, and, ultimately, queries over the Web.

Queries posed over relations with access patterns can be classified according to their potential to retrieve the query answers. Conjunctive queries (i.e., the select-project-join queries of relational algebra) can be classified as *executable* (or *well-moded*) if they can be executed in the traditional left-to-right reading of the query, *feasible* (or *orderable*) if an executable reordering exists, and *stable* (also called *feasible* by some) if semantically equivalent to an executable query [6,16,18,22,25]. For all these classes, one can always retrieve the complete answer to the query as if the relations with access patterns were ordinary relational tables. However, in general, one can only find a subset of the actual answers (the so-called *reachable certain answers*), obtained by adopting an extraction strategy that makes use of all the constants known from the query and/or previous domain knowledge. This query evaluation strategy is expensive, but can be improved by avoiding accesses that are irrelevant for the computation of the reachable certain answer. This brings forward the problem of minimization of the accesses used for the the evaluation of a given query, both static (before execution) [5,17] and dynamic (during execution) [1,2,4].

A crucial issue in data intensive systems is the ability to address *top-k queries* (a.k.a. *ranking queries*) [8,13], i.e., to retrieve only the best answers to a query without computing the entire result set. Ranking queries usually require an *early-out strategy* [12,23] that, based on a convenient use of thresholds, allows stopping the construction of the results and the inspection of the relations involved in the query. Early termination is necessary when accessing the relations is costly, e.g., when data are over the Web.

In this context, much of the recent research on queries over multiple domains where ranking and access restrictions are of primary importance goes under the banner of *search computing* [14]. In a broad sense, its goals include the definition of suitable cost models and the identification of optimization strategies for query plans involving several sources [3] while preserving the ordering of results that the user has in mind. Extensions of the model have considered cases in which user preferences are uncertain and therefore the top-k answers are also uncertain [24], which requires characterizing a new semantics, based on the notion of most representative ordering. The literature on ranking queries typically considers two kinds of access to data: sorted access (e.g., objects are retrieved by score, in decreasing order) and random access (given an id, retrieve the corresponding object) [7]. Search computing models covering these cases have placed special emphasis on the evaluation of joins between different sources [20,21]. Another important aspect of search computing concerns the geo-localization of results, typically enabled by the availability of sorted access by increasing distance from a given point. In this context, users often wish to enforce proximity [19] or diversity [9] of results.

6. PERSPECTIVES

We have observed in this work some challenges in querying Deep Web data, originating from "quirks" in Web services. This led us to present a list of desiderata of a solution that addresses these challenges. Of course, we do not claim to be exhaustive, and there are many additional challenges in the development of such a system that are beyond the scope of this paper. In particular, we have assumed that the Web services are readily available, but a practical solution would additionally involve a mechanism for service discovery. Moreover, we have assumed a simple relational model for data, but data on the Web comes in various formats: linked data, Web services, Web forms, plain tables, or even relation extraction from text. The exploitation of these different types of Web data poses further intriguing challenges.

7. REFERENCES

[1] M. Benedikt, P. Bourhis, and C. Ley. Querying schemas with access restrictions. *PVLDB*, 5(7), 2012.

[2] M. Benedikt, G. Gottlob, and P. Senellart. Determining relevance of accesses at runtime. In *PODS*, 2011.

[3] D. Braga, S. Ceri, F. Daniel, and D. Martinenghi. Optimization of multi-domain queries on the web. *PVLDB*, 1(1), 2008.

[4] A. Calì, D. Calvanese, and D. Martinenghi. Dynamic query optimization under access limitations and dependencies. *J. Univer. Comp. Sci.*, 15(21), 2009.

[5] A. Calì and D. Martinenghi. Querying Data under Access Limitations. In *ICDE*, 2008.

[6] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *TCS*, 371(3), 2007.

[7] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[8] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2), 2002.

[9] P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Top-k bounded diversification. In *SIGMOD*, 2012.

[10] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[11] B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the deep Web: A survey. *Communications of the ACM*, 50(2):94–101, 2007.

[12] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[14] I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Rank-join algorithms for search computing. In *SeCo Workshop*, 2009.

[15] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS*, 1995.

[16] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3), 2003.

[17] C. Li and E. Chang. Answering queries with useful bindings. *ACM TODS*, 26(3), 2001.

[18] B. Ludäscher and A. Nash. Processing union of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.

[19] D. Martinenghi and M. Tagliasacchi. Proximity rank join. *PVLDB*, 3(1), 2010.

[20] D. Martinenghi and M. Tagliasacchi. Top-k pipe join. In *ICDE Workshops*, 2010.

[21] D. Martinenghi and M. Tagliasacchi. Cost-aware rank join with random and sorted access. *IEEE TKDE*, 2012.

[22] A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[23] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008.

[24] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *SIGMOD*, 2011.

[25] G. Yang, M. Kifer, and V. K. Chaudhri. Efficiently ordering subgoals with access constraints. In *PODS*, 2006.