Rule-Based Exploration of Structured Data in the Browser

Sudhir Agarwal[1], Abhijeet Mohapatra[1],
Michael Genesereth[1], and Harold Boley[2]

[1] Logic Group, Computer Science Department, Stanford University, USA
[2] Faculty of Computer Science, University of New Brunswick, Fredericton, Canada
{sudhir,abhijeet,genesereth}[AT]cs.stanford.edu, harold.boley[AT]unb.ca

Abstract. We present Dexter, a browser-based, domain-independent structured-data explorer for users. Dexter enables users to explore data from multiple local and Web-accessible heterogeneous data sources such as files, Web pages, APIs and databases in the form of tables. Dexter's users can also compute tables from existing ones as well as validate the tables (base or computed) through declarative rules. Dexter enables users to perform ad hoc queries over their tables with higher expressivity than that is supported by the underlying data sources. Dexter evaluates a user's query on the client side while evaluating sub-queries on remote sources whenever possible. Dexter also allows users to visualize and share tables, and export (e.g., in JSON, plain XML, and RuleML) tables along with their computation rules. Dexter has been tested for a variety of data sets from domains such as government and apparel manufacturing. Dexter is available online at `http://dexter.stanford.edu`.

1 Introduction

Data is the fuel of innovation and decision support. Structured data is available to users through different sources. Examples include local files, Web pages, APIs and databases. Oftentimes, users need to quickly integrate and explore the data in an *ad hoc* manner from multiple such sources to perform planning tasks, make data-driven decisions, verify or falsify hypotheses, or gain entirely new insights.

Unfortunately, it can be very cumbersome, tedious or time consuming for users to explore data in an ad hoc manner using the current state of the art tools. This is because the current state of the art tools (a) provide limited or no querying support over the underlying data (e.g. domain-specific Web applications, public sources), (b) cannot compile information from multiple sources (e.g. search engines), or (c) require users' private data to be shipped to a remote server.

For example, consider the Govtrack website (`https://www.govtrack.us`) which has information (e.g. age, role, gender) about members of the U.S. Congress. Suppose, a user wishes to know "Which U.S. senators are 40 years old?" or "Which senate committees are chaired by a woman?". Even though Govtrack has the requisite data, it is very tedious to find answers to such elementary

questions. This is because Govtrack's UI and APIs present limited querying capabilities to a user. It is even harder to query across data from multiple sources e.g. "Which members of U.S. Congress were the Head of DARPA".

To address these problems, we have developed Dexter, a *domain-indepedent*, *browser-based* structured data explorer for users. Dexter enables users to create and connect multiple Web-accessible structured data (e.g. from Web pages, databases, APIs) as tables, and to explore these tables through Dexlog [14] rules with higher expressivity than is supported by the underlying data sources. Dexlog is a variant of Datalog [9] that is extended using negation and aggregation [15], and supports an extensive list of built-in arithmetic, string, as well as tuple- and set-manipulation operators.

The fundamental novelty of Dexter is its *client side evaluation* of user queries. To reduce query processing times, Dexter leverages two techniques. First, Dexter follows a hybrid-shipping strategy [13] and fetches query answers, instead of base data, from sources that offer support for queries. We note that, in such cases, a user's private data (e.g. locally stored data) is *never* shipped to a remote source. Second, to overcome browsers' memory limitations and to efficiently evaluate a user's query, Dexter horizontally partitions the tables and executes queries over the partitions in parallel, subsequently compiling the answers in a manner similar to the MapReduce [8] programming paradigm.

To enable users to effectively and conveniently explore structured data, Dexter presents users with two interfaces: (a) an intuitive table editor to work with data and (b) the *Dexlog Rule Editor* that is equipped with syntax highlighting and auto-completion. In addition, Dexter allows users to *visualize* their tables as charts and *export* their table along with their associated Dexlog rules in various formats including RuleML [4, 7]. Moreover, Dexter also allows users to *share* their tables with other users through a publicly accessible server.

The rest of the paper is organized as follows. In Section 2, we present an overview of the Dexlog language, which serves as the foundation of Dexter's data exploration capabilities. Then, in Section 3, we describe Dexter's features that enable users to *plug-in* structured data as tables and *play* with these tables (e.g. explore, visualize, export and share). In Section 4, we describe how Dexter efficiently evaluates a user's queries on the client side. Dexter has been tested for a variety of ad hoc queries over data sets from multiple domains. In Section 5, we present some scenarios involving ad hoc exploration of data about the U.S. government, apparel manufacturing, and restaurants. We compare and contrast Dexter to related tools and technologies in Section 6.

2 Dexlog

Dexter presents to its users a unified notion of tables that transcends the traditional separation between base tables and views. In Dexter, a table can contain tuples that are manually entered, computed using Dexlog rules, or both. We note that manual entry of a tuple in a table can be emulated through a trivial Dexlog rule where the tuple appears in the head and the body is a *valid* Dexlog formula

i.e. it evaluates to true in all possible interpretations. In this regard, Dexlog rules serve as the foundation for data exploration in Dexter. Dexlog is a variant of standard Datalog that is extended using negation (as failure), aggregation, and built-in operators over numbers, strings, tuples and sets. In this section, we present an overview of Dexlog, highlighting its distinguishing features with respect to standard Datalog and refer the reader to [15] for details regarding the syntax and semantics of Dexlog.

The vocabulary of Dexlog consists of *basic constants*, *relation constants*, *variables*, *tuples*, *sets*, and reserved keywords (such as `illegal`, which are discussed shortly). Basic constants are strings that are enclosed within double quotes e.g. `"1.23"` and `"Dexter"`. Relation constants and variables are strings of alphanumeric characters and underscore. However, relation constants must begin with a lower-case letter, e.g. `person`, `q_12`, while variables must begin with an upper-case letter, e.g. `X1`, `Y_`.

A key difference between standard Datalog and Dexlog is the introduction of tuples and sets as *first-class* citizens. A tuple is a non-empty, ordered collection of basic constants, variables, sets or other tuples which are separated using commas and enclosed within square brackets, e.g. `["John", "2"]`. Sets can be specified in Dexlog as *set constants*, each being a possibly empty unordered collection of basic constants, tuples or additional sets enclosed within braces, e.g. `{}` and `{{"1", "2"},{"3"}}`. In addition, Dexlog supports a special operator, called `setof`, for constructing sets.

Suppose that $\bar{X}$ and $\bar{Y}$ are two collections of Dexlog terms (i.e. constants and variables) and suppose that $\phi(\bar{X}, \bar{Y})$ is a conjunction of Dexlog literals. The `setof` atom `setof(`$\bar{X}$`,` $\phi(\bar{X}, \bar{Y})$`, S)` computes a set $\mathtt{S}_{\bar{Y}} = \{\bar{X} \mid \phi(\bar{X}, \bar{Y})\}$ for every binding of values in $\bar{Y}$.

Consider two relations: a unary relation `continent` and a binary relation called `located`, and their respective instances[3] as shown below.

continent
Asia
Africa
North America

located	
India	Asia
USA	North America
Canada	North America

Assuming that the `located(X,Y)` indicates that country `X` is geographically located in continent `Y`, the following Dexlog rule computes the set of countries for each continent.

```
v(X, S) :- continent(X) & setof(Y, located(Y, X), S)
```

The above Dexlog rule computes a set $\mathtt{S}_{\mathtt{X}} = \{\mathtt{Y} \mid \mathtt{located(Y,X)}\}$ for every binding of `X` such that `continent(X)` is true. Evaluation of the above rule on the instances of the tables `continent` and `located` results in the following tuples.

[3] For the sake of better readability, we omit quotes when a table cell is a basic constant.

v	
Asia	{"India"}
Africa	{}
North America	{"USA","Canada"}

Safe Dexlog Rules: To ensure the *safety* of Dexlog rules, all variables that occur in a negated atom in a rule's body must also occur in a positive atom. In addition, all the variables that occur in the body of a `setof` atom (but not as an aggregated argument) must be bound in some positive atom outside the `setof` atom. For example, the following rule is *unsafe* because there are infinitely many possible bindings of `X` such that `q(X, {})` evaluates to true.

```
q(X, S) :- setof(Y, located(Y, X), S)
```

Constraints in Dexlog: In addition to the rules for compute new facts from existing facts, Dexlog also allows users to validate a table's base or computed data by defining *constraints* over the table. Constraints have the general form `illegal :-` ϕ, where `illegal` indicates constraint violation [10] and ϕ is a Dexlog formula. Suppose that we wish to specify a constraint over the table `located` such that the second argument of every tuple in the `located` table must be contained in the `continent` table as well. We note that this constraint is an example of a *foreign-key constraint* [9] which can be specified using the following Dexlog rule involving negation.

```
illegal :- located(X, Y) & ~continent(Y)
```

Dexlog also supports an extensive list of built-in operators. Such operators include arithmetic operators (e.g. `gt`, `plus`, `mod`), string-manipulation operators (e.g. `length`, `substr`, `concat`), set-based operators (e.g. `sumOf`, `unionOf`) and tuple-based operators (e.g. `attr`). The complete listing of the built-in operators, their arity, binding patterns and description, is provided in [14].

3 Dexter: Interface and Features

In this section, we walk the reader through Dexter's interface and describe some of its main features and their role in enabling ad hoc exploration of structured data.

Dexter introduces a unified notion of tables that transcends the traditional separation between base tables and computed tables. In Dexter, a table can have manually entered data as well as rules that compute further data and constraints. To accommodate this unified notion of a table, Dexter presents its users with two interfaces to interact with tables. The first interface is a table editor that allows a user to manually enter data into a table or to update the data and the schema of a table. In addition, the user is also presented with an interface, called the Dexlog Rule Editor, for computing and validating a table's data through Dexlog rules.

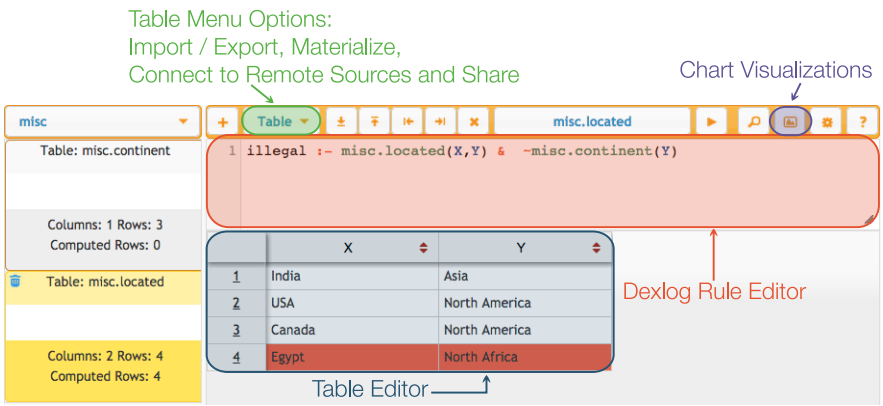

Fig. 1: Screenshot of Dexter's User Interface

A screenshot of Dexter's user interface depicting its different components is shown in Figure 1. On the left hand side, a list of tables is shown with the currently selected table highlighted with yellow background.

Creating and Managing Tables through the Table Editor: When a user creates a table or views a previously created table by selecting it in the list of tables through Dexter's UI, he/she is presented with a table editor as shown in lower right area of Figure 1.

The simplest way to populate a table in Dexter is to manually fill in values into the cells of the table editor. Dexter also allows a user to import data from CSV, JSON, or XML files that are located in the user's file system or accessible through a URL by interacting with the *Table* menu in Dexter's UI. Furthermore, a user can also extract structured data from a Web page into a table in Dexter by copying the relevant HTML DOM fragment and pasting it on to the table. Dexter uses the extraction algorithm that is proposed in [2] to convert a copied HTML DOM fragment into a table.

In addition to above methods, Dexter also allows users to create *remote tables* by connecting to MySQL databases (by specifying the MySQL server's address, the database name and the authentication details) as well as to Web APIs that support responses in JSON format. In contrast to Web applications that typically hard-code the set of underlying sources, Dexter is based on a generic source model that characterizes (a) the querying capability of a source, (b) accessibility of the source from the client side (e.g. whether the source is directly accessible or requires a proxy), and (c) the conversion of the data sent from the source to a table in Dexter. We note that, unlike tables that are created by manual data entry, importing files or extracting HTML DOM fragments, the data in a remote table cannot be directly updated. Instead, users may duplicate and edit a copy of a remote table in Dexter.

Users can manage the data and the schema of their tables through Dexter's table editor by updating the values in the table's cells, by inserting or deleting

rows or columns of a table respectively, or changing the column names. Users can *sort* a table by choosing an appropriate column through the table editor. Sometimes a user wishes to quickly see only a selected part of a large table. Defining a query for selecting the relevant rows could be too cumbersome in such a case. Therefore, Dexter's UI allows users to define filters on columns and shows only the matching table rows.

Validating and Computing Tables using the Dexlog Rule Editor: Dexter presents to its users an interface called the Dexlog Rule Editor where they can specify Dexlog rules. The Dexlog Rule Editor supports auto-completion and syntax highlighting to make it convenient for users to enter Dexlog rules. For any table in Dexter, the Dexlog Rule Editor records *all* of the Dexlog rules that are associated the table. These rules can either be used to validate or to compute the tuples of the table.

Consider our example from Section 2 of a Dexlog constraint over the `located` table.

```
illegal :- located(X, Y) & ~continent(Y)
```

As shown in Figure 1, a user can associate such a constraint with the `located` table in Dexter by entering the constraint into the Dexlog Rule Editor. Traditional database systems [9] enforce constraints on a table by preventing violations. In contrast, Dexter allows constraint violations to occur and employs visual feedback to pinpoint violations to a user. Specifically, the rows of a table that contribute to a constraint violation are *colored red* when the table's data is validated against the constraints. An example of such a violation pinpointing is shown in Figure 1. Since the tuple `"North Africa"` is not present in the `continent` table, the tuple `("Egypt", "North Africa")` violates the constraint on the `located` table and is, therefore, colored red.

Similar to specifying Dexlog constraints and validating tables, a user can also use the Dexlog rule editor to specify Dexlog rules and evaluate these rules to compute a table's data. When a user evaluates a computed table, the answers are streamed into Dexter's table editor as *read-only* rows. We discuss the process of evaluating Dexlog Rules in Section 4.

Materializing Tables: Dexter allows users to *materialize* computed tables in their browser's local storage. Materializing a computed table can significantly reduce the time taken to evaluate Dexlog rules over the table by using the materialization of the computed table instead of re-computing the table. We note that, currently, Dexter does not automatically update the materialization of computed tables when the data in the underlying sources is changed. Rather, Dexter relies on users to re-evaluate and re-materialize the computed tables any time they wish to do so.

Visualizing Tables: Charts are a great way to comprehend, analyze, and effectively interact with a table's data. In addition to allowing users to explore table through Dexlog rules, Dexter enables users to visualize their table's data (including computed tuples) as charts. In order to create a chart based on the data in the currently selected table in Dexter's UI, a user opens the chart creation window by clicking on the "Chart Visualization" button (see Figure 1). A user

can then select the columns to be plotted and the type of the chart and inspect the chart. Dexter supports popular chart types such as line chart, area chart, bar chart, column chart, scatter plot and pie chart. Dexter allows users to export their charts as images in popular image formats.

Exporting and Sharing Tables: Dexter stores user's tables locally inside their respective browsers. Tables that are accessible through Dexter in one browser are not accessible through a different browser (even within the same machine). In order to support inter-application exchange of user's data, Dexter allows its users to *export* their tables in two different ways. First, users can export their table's *data* as CSV, XML or JSON files. Users can also export a table's data along with the associated *validation* and *computation rules* in Dexter's native file format, which uses the extension `.dxt` and in Naf Datalog RuleML / XML[4], which uses the extension `ruleml`.

Dexter makes it possible for users to share their data with other users and use data shared by other users. With Dexter's UI users can easily publish their tables to Dexter's sharing server to make their tables accessible through different browsers or to other users. The shared tables are accessible to all Dexter users by selecting the folder "shared" in the upper left corner in Dexter's UI.

4 Efficient Evaluation of Dexlog Rules

Dexter allows users to query their tables by specifying Dexlog over the (relational) schemas of the involved sources. These queries are evaluated by Dexter on the client side by employing a Hybrid-Shipping strategy [13]. Our query evaluation strategy ensures acceptable query answering performance without requiring users to compromise on their privacy or overloading the client machines. We note that the naive approach of fetching all the required data to the client machine and subsequently, evaluating the query answers, also referred to as Data-Shipping [13], is not practical mainly because the client machines are usually not high-performance machines.

Dexter evaluates a user's query in the following steps. First, an input query is *decomposed* into partial queries, such that each partial query can be answered independently at a source. Next, the resulting sub-queries are filtered to remove rules that would never be fired while evaluating the user's query. After, *removing irrelevant rules*, the resulting partial queries are *fragmented* by horizontally partitioning the data of the underlying sources. Finally, the fragmented queries are executed in parallel and the resulting answers are compiled to construct the answers to the input query.

4.1 Query Decomposition

A query over the internal schema is decomposed into partial queries and rules that assemble the answers to the partial queries. The hybrid-shipping strat-

[4] `http://wiki.ruleml.org/index.php/Dexter_and_RuleML`

egy [13] for query decomposition in Dexter depends on the sources that are required to evaluate a query and on the querying capability of a source.

Suppose that `senator(Name, Party, State)` and `party(Id, PartyName)` are two tables in Dexter that are created by connecting to Govtrack's relational database. Furthermore, suppose that the table `tweet(U, Tweet)`, which represents the Twitter posts of a user `U`, is created by connected to the Twitter API. Since databases support filtering of table on its attribute values, the query `q(T):-senator(U,P,S) & tweet(U,T) & party(P,"Rep")` will be decomposed into partial queries `q1` and `q2` as follows.

```
    q(T) :- q1(U) & q2(U, T)
   q1(U) :- senator(U, P, S) & party(P,"Rep")
q2(U, T) :- tweet(U, T)
```

In order to evaluate the query `q`, the partial queries `q1(U)` and `q2(U,T)` are sent to Govtrack's database and the Twitter API respectively.

However, if Govtrack does not allow the `senator` table to be filtered by the attribute `Party`, then the whole table is shipped to the client side where the filters are, subsequently, applied. In addition, if it is not known whether a certain operation (say, join or negation) can be performed at a source, then the relevant operation is performed on the client side after shipping the data from the source. We note that, in Dexter, the queries over local data (such as CSV, XML or JSON files) are *always* evaluated on the client side and never shipped to a source. Thus, the privacy of a user's local data is always preserved.

4.2 Removal of Irrelevant Rules

The collection of partial queries that results from the decomposition step is filtered to remove any irrelevant rules from it. The irrelevant rules are filtered using a simple procedure that, for a given input predicate, iterates through the rules and recursively selects all the rules that the input predicate depends on. Using this procedure for the query predicate as input gives us all the relevant rules filtering out any irrelevant rules.

We note that Dexter supports stratified evaluation of queries with negation or `setof`. In order to evaluate such queries, the strata of the predicates involved in the query are computed using the technique presented in [1]. The evaluation of the query starts by evaluating the predicates at strata 0.

4.3 Query Fragmentation

The straightforward evaluation of the partial queries resulting from the previous steps can become a double bottleneck due to (a) the complexity of the query, and (b) the size of the answers, especially when the size of the data shipped from a source is too big to handle for a user's browser. To resolve these bottlenecks, partial queries are fragmented horizontally into chunks based on the the size of

the browser's local store and the maximum number of answers to a query that can be returned from a source. For example, suppose that the Twitter API allows a maximum of 500 tweets to be returned per call. If the number of senators, number of parties and the total number of tweets are, say, 500, 100 and 10000, respectively, then the query `q(T)` is fragmented into 20 fragments (assuming the chunk size to be 500). In order to be able to execute the fragments in parallel, the corresponding offsets and limits are appended to the relation names in the rules of a fragment. For our example, the rules for the first fragment will be as follows (and analogous for the other 19 fragments):

```
   q(T) :- q1(U) & q2(U,T)
  q1(U) :- senator_0_500(U,P,S) & party_0_100(P,"Rep")
q2(U,T) :- tweet_0_500(U,T)
```

4.4 Parallel Evaluation of Queries

In general, the number of partitions obtained from the previous step can be so large that it may not be feasible for a user's machine to evaluate all of them at the same time in parallel. Dexter allows users to set the maximum number of parallel threads they want Dexter to use while evaluating queries. Dexter schedules the execution of partitions such that at any time the number of query fragments executed in parallel does not exceed this limit.

Source Invocation and Building the Cache A partition first builds the fact base required for answering the query. For this purpose, its rules are processed and appropriate source-specific queries are constructed. The construction of source-specific queries takes into consideration the source-connection information as well as the information about offset and limit appended to the relation names in the previous steps. Then, the source-specific queries are sent to the sources and results are converted to relations by using the appropriate wrappers, which could be located either on the client (e.g. local files, servers that support JSONP requests) or on the server (e.g. Twitter API, databases such as MySQL). To increase efficiency and relieve the sources, Dexter caches the data of a source invocation as long as doing so does not exceed the maximum cache size. Dexter's cache can be used by all partitions. In our example, since `party_0_100(P,"Rep")` is present in every partition, not caching source responses would lead to 20 Govtrack API invocations for the same data.

Answer Construction In the answer construction stage, the answers of the partial query fragments are assembled to compute the answer to the user's input query. Although one query fragment does not return one answer multiple times, same tuple could be returned by different query fragments. Therefore, the final answer is computed by taking the set-union of the answers of all query fragments.

5 Demonstration Scenarios

In this section, we present some example scenarios for which Dexter has been successfully evaluated. Note that Dexter is a *domain-independent* data browser and the number of scenarios for which it can be used is potentially infinite. The purpose of the scenarios described in this section is merely to demonstrate Dexter's features.

Nr.	Format	URL and Description
1	CSV	`http://www.fec.gov/data/AdminFine.do` Data about administrative fines
2	API	`https://www.govtrack.us/developers/api` Data about current U.S. senators and committees / sub-committees.
3	API	`http://www.yelp.com/developers/documentation/search_api` Data about pizzerias in Downtown, Washington D.C.
4	CSV	`http://www.opendatadc.org/dataset/restaurant-inspection-data` Health inspection data in Washington D.C restaurants
5	Webpage	`https://www.eecs.mit.edu/people/faculty-advisors` Data about MIT faculty
6	JSON	`http://manufacturingmap.nikeinc.com/maps/export_json` Data about manufacturers of Nike collegiate products.

Table 1: Data Sources used in the Demonstration Scenarios

S1: *Which U.S. senator or representative has incurred the maximum administrative fine? What is the distribution of administrative fines across the U.S. states?*

For this scenario, a user imports the CSV file (Line 1 of Table 1) into a table called `fines`, and sorts the table by column containing fines amounts (`Finamo`) in descending order. The user, then, creates e.g. a chart such as the one shown in Figure 2 with Dexter's interactive interface for visualizing the distribution of administrative fines across U.S. states.

S2: *Which of the current U.S. senators have been fined by the Federal Election Commission (FEC)? Do any of them chair a committee or a sub-committee?*

For this scenario, a user creates two tables `persons` and `committees` in Dexter and connects the tables to the Govtrack API (Line 2 in Table 1) to obtain the data about members of the U.S. congress and the congressional committees, respectively. Then, the user joins the `persons` table with the `fines` table (from Scenario **S1**) to find the senators who have been fined by the FEC. By joining the result of the previous query with the `committees` table, the user can find the current U.S. senators, who chair a committee or a sub-committee and have been fined by the FEC.

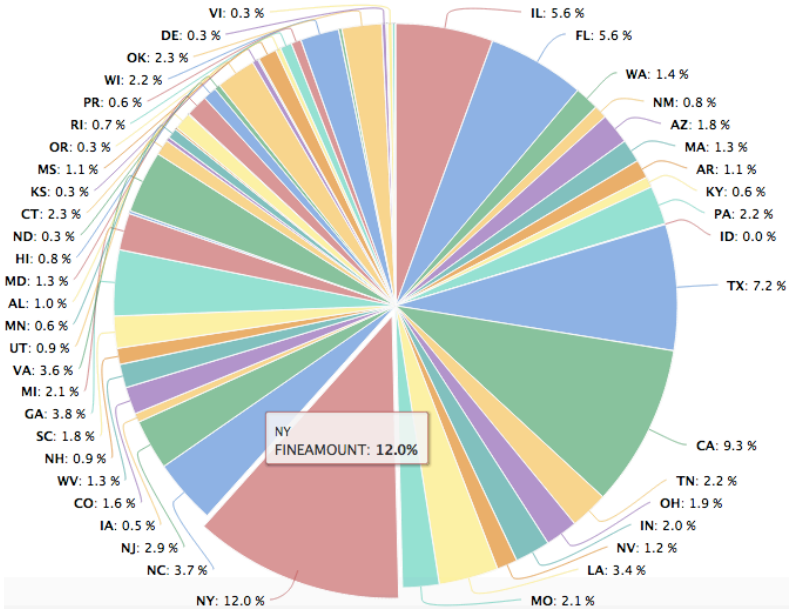

Fig. 2: Distribution of administrative fines across U.S. states.

S3: *Which pizzerias in Downtown, Washington D.C. have health code violations?*

For this scenario, a user creates a table called `dcPizzerias` and connects the table to the Yelp API (see Line 3 in Table 1) to obtain the data about pizzerias in Downtown, Washington D.C. Then, the user imports the data about health code violations (see Line 4 in Table 1) into a table called `healthViolation`. By evaluating a join over the two tables, the user can obtain the list of pizzerias in Washington D.C. that violate the health code regulations.

S4: *Which MIT CS faculty have not listed their contact details (phone number, office)?*

For this scenario, a user opens the MIT CS faculty Web page (Line 5 in Table 1) in his/her browser, selects the fragment of the Web page that contains the information on MIT CS faculty members, and pastes it in a table, say `mitCSFaculty` in Dexter. Dexter converts the pasted Web page fragment automatically into a table (for more details see [2]). Then, the user specifies the following constraints on the `mitCSFaculty` table.

```
illegal :- mitCSFaculty(Name,Pos,Email,"",Off,Inst)
illegal :- mitCSFaculty(Name,Pos,Email,Ph,"",Inst)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://deliberation.ruleml.org/1.01/xsd/nafdatalog.xsd"?>
<RuleML xmlns="http://ruleml.org/spec">
   <Assert>
      <Forall>
         <Var>Name</Var><Var>Pos</Var><Var>Email</Var><Var>Off</Var><Var>Inst</Var>
         <Implies>
            <Atom><Rel>mitCSFaculty</Rel><Var>Name</Var><Var>Pos</Var><Var>Email</Var>
               <Data></Data><Var>Off</Var><Var>Inst</Var></Atom>
            <Atom><Rel>illegal</Rel></Atom></Implies></Forall>
       <Forall>
          <Var>Name</Var><Var>Pos</Var><Var>Email</Var><Var>Ph</Var><Var>Inst</Var>
          <Implies>
             <Atom><Rel>mitCSFaculty</Rel><Var>Name</Var><Var>Pos</Var><Var>Email</Var>
                <Var>Ph</Var><Data></Data><Var>Inst</Var></Atom>
             <Atom><Rel>illegal</Rel></Atom>
          </Implies></Forall></Assert></RuleML>
```

Fig. 3: Validation Rules (Constraints)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://deliberation.ruleml.org/1.01/xsd/nafdatalog.xsd"?>
<RuleML xmlns="http://ruleml.org/spec">
   <Assert>
      <Forall>
         <Var>A</Var><Var>B</Var><Var>C</Var><Var>D</Var><Var>E</Var><Var>F</Var><Var>I</Var>
         <Implies>
            <And>
               <Atom><Rel>mitCSFaculty</Rel><Var>A</Var><Var>B</Var><Var>C</Var>
                     <Var>D</Var><Var>E</Var><Var>F</Var></Atom>
               <Atom><Rel>indexOf</Rel><Var>F</Var><Data>CSAIL</Data><Var>I</Var></Atom>
               <Atom><Rel>gt</Rel><Var>I</Var><Data>-1</Data></Atom>
            </And>
            <Atom><Rel>mitCSAIL</Rel><Var>A</Var><Var>B</Var><Var>C</Var>
               <Var>D</Var><Var>E</Var><Var>F</Var>
            </Atom>
         </Implies></Forall></Assert></RuleML>
```

Fig. 4: Computation Rule

When the above constraints are evaluated, the rows corresponding to MIT CS faculty members who have not listed a phone number or their office details are *colored red* in Dexter's UI to indicate a constraint violation.

Figure 3 shows the Naf Datalog RuleML/XML generated with Dexter for the above two constraints. Dexter can also export computation rules in Naf Datalog RuleML/XML syntax. Suppose, we have the following computation rule to compute only those MIT CS faculty members that are members of CSAIL.

```
mitCSAIL(A,B,C,D,E,F)  :-
    mitCSFaculty(A,B,C,D,E,F) & indexOf(F,"CSAIL",I) & gt(I,"-1")
```

Figure 4 shows the Naf Datalog RuleML/XML generated with Dexter for the above computation rule.

S5: *For every Nike product type, what is the total number of immigrant workers in the factories that supply the product?*

For this scenario, a user imports the file containing data about manufacturers of Nike collegiate products (see Line 6 in Table 1) into a table called `nikeFactories`. By evaluating the following rules, the user obtains the total number of migrant workers per product type. Note that in the second rule we use '*' only for the sake of readability of the paper.

```
s5(PType,NumMigWrkrs) :- aux(PType,IdTemp,MwTemp) &
    setof([Id,Mw],aux(PType,Id,Mw),MwSet) & sumOf(MwSet,2,NumMigWrkrs)

aux(PType,Id,MigWrkrs) :- nikeFactories(Id,*,Wrkrs,*,MwP,*,PType,*) &
    replace(MwP,"%","",MwPer) & times(MwPer,.01,Wrkrs,MigWrkrs)
```

Feature\Scenario	**S1**	**S2**	**S3**	**S4**	**S5**
Import	✓	✓	✓		✓
Export				✓	
Web Extraction				✓	
API/Database		✓	✓		
Sorting	✓				
Constraints				✓	
Queries (Select / Project)	✓	✓		✓	✓
Queries (Join / Aggregates)		✓	✓		✓
Data Visualization	✓				
Table Editing	✓	✓	✓		✓

Table 2: Features of Dexter covered by the Demonstration Scenarios S1–S5

We note that the scenarios described above cover *most* of the features of Dexter. The features that are not covered in the demonstration scenarios S1–S5 are the exporting and sharing of tables. Table 2 summarizes the coverage of Dexter's features in the demonstration scenarios S1–S5.

6 Related Work and Concluding Remarks

We presented Dexter, a tool that empowers users to explore structured data from various Web-accessible sources such as databases, local files (e.g. CSV, XML and JSON), Web pages and Web-APIs expressively and in ad hoc fashion. To the best of our knowledge, Dexter is the first system to provide such functionality to users. Popular search engines do not support *compilation of information from multiple documents*, a prerequisite to satisfy the overall information need of a user. Semantic Web [5] and Linked Data [6] rely on existence of semantic annotations for websites or directly accessible semantic data respectively. Dexter

takes a bottom-up approach by supporting multiple widely used types of data sources and data formats instead of only RDF.

Dexter stores user's data locally inside his/her browser as opposed to typical server-side systems that store user's data on the server. This feature of Dexter ensures that users can combine their private and confidential data with public data without compromising on their privacy. Although popular spreadsheet software such as Microsoft Excel support local storage and ad hoc analysis of user's data, they lack data capability of querying across multiple remote sources such as *joins across multiple sources*. Google's Fusion Tables [11] supports more expressive queries than traditional spreadsheets. However, Google's Fusion Tables is a purely server-side system and requires all the data to be on the server.

DataWrangler is a browser-based interactive data cleaning and transformation tool [12]. While DataWrangler can suggest edits based on user's interactions and Dexter does not, Dexter supports complex validation rules incl. conditions involving multiple tables whereas DataWrangler can check only for missing values. In addition to research prototypes, there is an ever increasing number of commercial systems such as Trifacta, Microsoft Azure, Tamr, MindTagger, Informatica, and Tableaux. While each of them has its own strengths and weaknesses, they all are targeted primarily toward making organization-internal data easily consumable for the employees of an organization. In contrast, Dexter is primarily targeted toward making publicly available data easily consumable for users.

Apart from the lack of support for privacy, server-side systems targeted toward users such as Wikidata [16] and Socrata (`http://www.socrata.com/`) typically do not support expressive queries; scalability being one of the many reasons for this choice. Dexter addresses the scalability problem with a novel architecture combined with the hybrid shipping strategy [13], in which queries are evaluated on the client side while exploiting the querying capabilities of remote sources. Dexter-Client communicates directly with data sources when possible, and through Dexter-Server (proxy) otherwise. Dexter query evaluation technique respects a user's privacy as it never ships a user's local data to a remote server. By enabling users to pose highly expressive queries over a source, across sources (including local data), Dexter bridges the gap between the querying capability of sources and the information need of a user. For detailed analysis of the above mentioned and more Dexter-related tools and technologies we refer to [3].

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995), `http://www-cse.ucsd.edu/users/vianu/book.html`
2. Agarwal, S., Genesereth, M.R.: Extraction and integration of web data by end-users. In: He, Q., Iyengar, A., Nejdl, W., Pei, J., Rastogi, R. (eds.) CIKM. pp. 2405–2410. ACM (2013)
3. Agarwal, S., Mohapatra, A., Genesereth, M.: Survey of dexter related tools and techonologies. `http://dexter.stanford.edu/semcities/TR-DexterRelatedWork.pdf` (2014)

4. Athan, T., Boley, H.: The MYNG 1.01 Suite for Deliberation RuleML 1.01: Taming the Language Lattice. In: Patkos, T., Wyner, A., Giurca, A. (eds.) Proceedings of the RuleML 2014 Challenge, at the 8th International Web Rule Symposium. vol. 1211. CEUR (Aug 2014)
5. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American 5(284), 34–43 (May 2001)
6. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
7. Boley, H., Paschke, A., Shafiq, O.: RuleML 1.0: The Overarching Specification of Web Rules. In: Proc. 4th International Web Rule Symposium: Research Based and Industry Focused (RuleML-2010), Washington, DC, USA, October 2010. Lecture Notes in Computer Science, Springer (2010)
8. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008)
9. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems - the complete book (2. ed.). Pearson Education (2009)
10. Genesereth, M.R.: Data Integration: The Relational Logic Approach. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2010)
11. Gonzalez, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R., Shen, W.: Google fusion tables: data management, integration and collaboration in the cloud. In: Hellerstein, J.M., Chaudhuri, S., Rosenblum, M. (eds.) Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010. pp. 175–180. ACM (2010)
12. Kandel, S., Paepcke, A., Hellerstein, J., Heer, J.: Wrangler: interactive visual specification of data transformation scripts. In: Tan, D.S., Amershi, S., Begole, B., Kellogg, W.A., Tungare, M. (eds.) Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011. pp. 3363–3372. ACM (2011)
13. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. 32(4), 422–469 (Dec 2000)
14. Mohapatra, A., Agarwal, S., Genesereth, M.: Dexlog: An overview. `http://dexter.stanford.edu/main/dexlog.html` (2014)
15. Mohapatra, A., Genesereth, M.R.: Reformulating aggregate queries using views. In: Frisch, A.M., Gregory, P. (eds.) SARA. AAAI (2013)
16. Vrandecic, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM 57(10), 78–85 (2014)