Querying the Data Web

The MashQL Approach

MashQL, a novel query formulation language for querying and mashing up structured data on the Web, doesn't require users to know the queried data's structure or the data itself to adhere to a schema. In this article, the authors address the fact that being a language, not merely an interface (and simultaneously schema-free), MashQL faces a particular set of challenges. In particular, the authors propose and evaluate a novel technique for optimizing queries over large data sets to allow instant user interaction.

Mustafa Jarrar
Birzeit University, Palestine

Marios D. Dikaiakos
University of Cyprus, Cyprus

In parallel with the hypertext Web's continuous development, we're witnessing an explosion of structured data published on the Web, with companies such as Google, Freebase, Upcoming, eBay, Yahoo, and Amazon competing to gather and publish this content in a way that encourages people to reuse it. The trend of publishing structured data on the Web is shifting the focus of Web technologies toward new paradigms of *structured data retrieval*. Traditional search engines can't serve such data adequately because a keyword-based query will still be ambiguous, even though the underlying data is structured. To exploit the massive amount of structured data on the Web to its full potential, users should be able to easily query and fuse it.

Unlike keyword-based information retrieval, the main challenge in structured data retrieval is that, before formulating a query, the user has to know the data's structure and attribute labels (that is, the *schema*). However, in many cases, a schema can be dynamic, adding and dropping several kinds of items that have different attributes. Other data sources might be schema-free, or the schema might consist of assertions mixed up with the data. In RDF, for example, the data doesn't have to be consistent with a certain schema or ontology, so the user has to manually investigate the RDF data itself before querying it to understand both vocab-

1089-7801/10/$26.00 © 2010 IEEE

Related Work in Query Formulation

Here's a brief review of some of the main approaches to query formulation and how they relate to MashQL's novel contributions:

- *Query-by-form* is the simplest approach for queries, but not flexible.
- *Query-by-example* requires data to be schematized and users to be aware of the schema.
- *Conceptual queries* let people query a database starting from its diagrams (many databases have EER or ORM conceptual diagrams describing them).[1]
- *Natural language queries* let users write their queries as natural language sentences, which are then translated by query engines into SQL[2] or XQuery.[3] Unfortunately, this approach is fundamentally bound with language ambiguity.
- *Mashup editors and visual scripting* let users write query scripts inside a module and visualize these modules as boxes connected with lines. However, when a user needs to express a query over structured data, he or she has to use that editor's formal language (for example, YQL for Yahoo). Deri Pipes[4] are inspired by visual scripting, letting users write SPARQL queries in a *textual form* inside a box and link it to other boxes. Deri Pipes focus on the pipelining aspects, such as what operators are needed in a pipeline. MashQL was inspired by the way Yahoo visualizes query modules, but its main purpose is query formulation itself — that is, what's inside a query model. Hence, MashQL is a complement, rather than an alternative, to Yahoo and Deri Pipes.
- *Interactive queries* let users query schema-free XML. The closest approach to MashQL is Lorel,[5] but instead of querying the original data, Lorel queries a summary of the data in a DataGuide that contains only the possible paths between predicates (that is, it groups unrelated items having the same property labels). Moreover, Lorel doesn't support querying multiple sources, and its expressivity is very basic. In contrast, MashQL supports path disjunctions, negations, variables, union, and reverse properties, among others.

References

1. A. Bloesch and T. Halpin, "Conceptual Queries Using ConQuer-II," *Proc. 15th Int'l Conf. Conceptual Modeling* (ER 97), LNCS 1157, Springer, 1996, pp. 113–126.
2. Y. Li, H. Yang, and H. Jagadish, "NaLIX: An Interactive Natural Language Interface for Querying XML," Proc. ACM SIGMOD Int'l Conf. Management of Data, ACM Press, 2005, pp. 900–902.
3. A. Popescu, O. Etzioni, and H. Kautz, "Towards a Theory of Natural Language Interfaces to Databases," *Proc. 8th Int'l Conf. Intelligent User Interfaces*, ACM Press, 2003, pp. 149–157.
4. D. Le Phuoc et al., "Rapid Prototyping of Semantic Mash-Ups through Semantic Web Pipes," *Proc. 18th Int'l Conf. World Wide Web*, ACM Press, 2009, pp. 581–590.
5. R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," *Proc. 23rd Int'l Conf. Very Large Data Bases*, Morgan Kaufmann, 1997, pp. 436–445.

ulary and data structures. This becomes even more challenging when a query involves multiple RDF sources.

As discussed in the "Related Work in Query Formulation" sidebar, helping users easily search and consume data is a general and known challenge in many different areas. For a query language to be practically sound in the context of an open environment such as the Web, the language should hold the following assumptions:

- The user doesn't know the schema.
- The data might be schema-free.
- A query could involve multiple data sources.
- The language is sufficiently expressive — that is, it isn't merely a single-purpose user interface.

We present here a query formulation language called MashQL, the novelty of which is that it considers all these assumptions. MashQL is a general-purpose query formulation language, but we focus on the *data Web* scenario, meaning we regard the Web as a database in which each data source is a table. In this way, we can regard a data mashup as a query involving multiple data sources. To illustrate MashQL's power, we focus on RDF not only because it's the most challenging data model, but because it's also the most primitive — that is, we can easily map other models (such as XML and databases) into it.

MashQL

Figure 1 shows two RDF sources and a MashQL query to retrieve "anything written by Lara, published after 2007, and with a title." The first module specifies the input and the second the query. The user can pipe the output into a third module (not shown here), thereby rendering the results into a certain format (such as HTML or XML) or as RDF input to other queries. As we describe later, such a query is formulated interactively, without any prior knowledge of the

Figure 1. A query over two RDF data sources. Here, we're trying to retrieve "anything written by Lara, published after 2007, and with a title."

schema and without assuming that data adheres to a schema or ontology.

The union operator "\" allows the combination of two properties within or across data sources (for example, `Year\PubYear`). Notice that although we can use MashQL for data integration and fusion, this isn't a goal per se for the language. Data integration requires not only syntax but also semantic integration, which MashQL doesn't support. Instead, it's mainly designed for query formulation, by which people are able to spot different labels for the same properties (as they navigate through data sets) and to manually combine them, as the previous example illustrates.

MashQL's Intuition

Each MashQL query is seen as a tree. The root is called the *query subject*, which is the subject matter being inquired; a subject can be a particular instance, an instance type, or a user-defined variable label. Each branch of the tree is called a *restriction* and is used to restrict a certain property of the subject; branches can be expanded to allow subtrees, called *query paths*. In this case, a property's object is seen as the subject of its subquery, helping users navigate through the underlying data set and build complex queries. Objects marked with "✓" will be returned in the query results. When querying different sources, MashQL considers two properties (or two instances) to be the same if they have the same URI.

To illustrate the notion of query paths, Figure 2 shows a query to retrieve recent articles from Malta — that is, the title of every article that has an author, this author has an address, this address has the country "Malta," and the article is published after 2007.

Query Formulation

Formulating a query is an interactive process, during which a user performs selections from drop-down lists. While the user interacts with the query editor, the editor queries the data set in the background to generate these lists. After specifying the input sources (the data set), the user first selects from the *subjects list*, which contains either the set of subject types (such as `Article`) or the union of all subject and object identifiers in the data set, such as `A1` or `B2`. The user can also choose to introduce a new subject label, which the MashQL editor then considers as a variable and displays in italics — the default variable is `Anything`. The user then selects from the *properties* list, which the editor generates on the fly and that comprises all properties pertinent to the chosen subject. Finally, the user selects a *restriction*: if he or she wants to add an object filter on the previously selected property, the editor

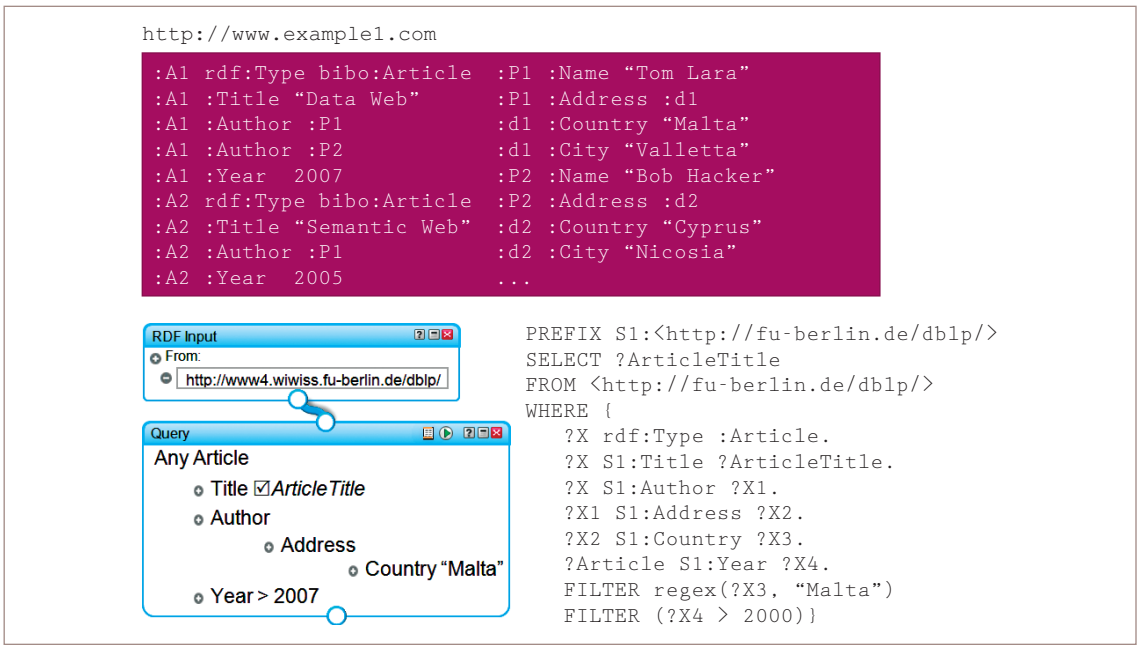


Figure 2. Query paths (/`subtrees`) in MashQL. Here, we're trying to retrieve recent articles from Malta.

will offer a list {`Equals`, `Contains`, and so on}. If the user wants to add an object identifier as a restriction, the editor will generate a list of the possible objects (depending on the previous selections). Users can also choose to expand the property to declare a query path. In this way, they can navigate and query a data graph without prior knowledge of it, even if the data is schema-free.

The formal algorithm to generate the drop-down lists appears in the extended version of this article.[1]

Syntax and Semantics

MashQL queries aren't executed directly; instead, the editor translates them into and then executes them as SPARQL queries, thus MashQL's semantics follow SPARQL's. When evaluating a query $Q(S)$, only the triples satisfying all restrictions are retrieved, such that

- If a restriction isn't prefixed with a modal operator (`R:=<empty, P, Of>`), the editor considers its evaluation to be true if the subject, the predicate, and the object-filter are matched (see Def. 3 in Definition 1); this case is mapped into a normal triple pattern in SPARQL.
- If a restriction modality is "maybe" (`R:=<maybe, P, Of>`), its evaluation is always true and mapped into an optional triple pattern in SPARQL.
- If a restriction is modality "without" (`R:=<without, P, Of>`), its evaluation is true if the subject `S` and the predicate `P` don't appear together in a triple — that is, the object `O` shouldn't be bound.

Definition 1 presents a summary of MashQL's formal definitions; the full SPARQL interpretation appears elsewhere.[1]

When evaluating the query in Figure 3, we retrieve all RDF triples with the same subject that have a predicate `Title`, a predicate `Artist` with an object identifier being `Shakera`, and an optional predicate `Album`, and don't have the predicate `Copyright`.

MashQL supports nine forms of filters and query paths (Def. 4); four forms of unions (Def. 5); formulating queries at the type level (Def. 6); and exploring an RDF graph backward (Def. 7), not just forward.

The MashQL Editor

We implemented MashQL in two scenarios: a server-side query and mashup editor, and a browser-side Firefox plug-in. As Figure 4 shows, the server-side MashQL editor's functionality includes a state machine for dispatching background queries, translating a formulated MashQL query into SPARQL, and executing a query and rendering its results. Users can materialize and publish MashQL queries if needed. The output

Def. 1 (query): A query Q with a subject S, $Q(S)$, is a set of conjunctive restrictions on S. $Q(S) := R_1 \wedge \ldots \wedge R_n$.

Def. 2 (subject): A subject $S \in I \cup V$, where I is an identifier, and V is a variable.

Def. 3 (restriction): A restriction $R := <R_x, P, O_f>$, where R_x is a modal operator, $R_x \in \{\texttt{empty}, \texttt{maybe}, \texttt{without}\}$; P is a predicate; $P \in I \cup V$; and O_f is an object filter.

Def. 4 (object filter): An object filter $O_f := <O, f>$, where O is an object, and f is a filtering function; f can have one of the following nine forms:

1. $O_f := <O>$, where O is an object, $O \in V \cup I$.
2. $O_f := <O, \text{Equals}(X, T, L_t)>$, where X can be a variable or a constant, T a data type, and L_t a language tag.
3. $O_f := <O, \text{Contains}(X, T, L_t)>$.
4. $O_f := <O, \text{MoreThan}(X, T)>$.
5. $O_f := <O, \text{LessThan}(X, T)>$.
6. $O_f := <O, \text{Between}(X, Y, T)>$.
7. $O_f := <O, \text{OneOf}(V)>$, where V is a set of values $\{v_1 \ldots v_n\}$.
8. $O_f := <O, \text{Not}(f)>$, where f is one of the functions defined previously.
9. $O_f := <O, Q_i(O)>$, where O is an object $O \in V \cup I$, and $Q_i(O)$ is a subquery, with O being the query subject.

Def. 5 (union): A union operator can be defined as the following:

1. $O_n = <O_1 \backslash O_2 \backslash \ldots \backslash O_n>$, unions between objects, $O_i \in I$.
2. $P_n = <P_1 \backslash P_2 \backslash \ldots \backslash P_n>$, unions between predicates, $P_i \in I$.
3. $S_n = <S_1 \backslash S_2 \backslash \ldots \backslash S_n>$, unions between subjects, where $S_i \in I$.
4. $Q_n = <Q_1 \backslash Q_2 \backslash \ldots \backslash Q_n>$, unions between queries,

Def. 6 (types): A subject ($S \in I$) or an object ($O \in I$) can be prefixed with "Any" to mean the instances of this subject/object type.

Def. 7 (reverse): $<\sim P>$ the reverse of the predicate P. Let R_1 be a restriction on S such that $<S\ P\ O>$, and R_2 be $<O \sim P\ S>$, then $R_1 = R_2$.

Definition 1. The formal definition of MashQL. Here, we also see its main constructs.


```
SELECT ?SongTitle, AlbumTitle
WHERE {
 ?X :Title ?SongTitle.
 ?X :Artist :Shakera.
 Optional{?X :Album ?AlbumTitle}
 Optional{?X :Copyright ?X1}
 FILTER !Bound(?X1)}
```

Figure 3. A MashQL query and its mapping into SPARQL. The query retrieves everything that has a title and artist, maybe has an album, and doesn't have a copyright.

of each published query receives a URL and is seen as a concrete RDF source — that is, materialized and stored physically. Cyclic queries aren't allowed, meaning a query's input can't also be its output, directly or through a chain.[1]

When a user specifies a data source as input, the editor bulk loads it into Oracle 11g. Subsequently, the MashQL editor uses AJAX to dispatch background queries and the SPARQL translation for Oracle 11g execution. We chose Oracle 11g for its native RDF support and database management system (DBMS) functionalities, such as materialization, indexing, and partitioning.

For the MashQL GUI, we adopted the Yahoo Pipes style for visualizing Web feed mashups and used Yahoo's open source JavaScript libraries. Hence, a data mashup becomes a query over multiple data sources, and its setup follows the simple paradigm of Web feed filters and mashups. It's worth noting that the examples we describe in this article can't be built via Yahoo Pipes: Yahoo allows a limited support of XML mashups, but this is neither graphical nor intuitive because you have to write complex scripts in YQL, the Yahoo Pipes query language.

Our second implementation (Firefox plug-in) has the same functionality as the online editor, but it doesn't use databases in the back end. Queries are executed inside the browser, using the Jena SPARQL query libraries to allow querying and fusing of websites that embed RDFa.[1,2] In this way, the browser serves as a Web composer rather than simply a navigator.

Implementation Issues

To further improve MashQL's elegance and intuitiveness, we present some technical important challenges and how we resolve them.

URI Normalization

Because RDF can contain unwieldy URIs, queries might be inelegant. Thus, the MashQL editor normalizes URIs and displays that normalization instead — for example, `Type` instead of `www.w3.org/1999/02/22-rdf-syntax-ns#type`. If the user mouses over `type`, its full URI is displayed as `tip`. In case of different URIs leading to the same normalization, the editor distinguishes them by a gray prefix (`1:type`, `2:type`). This normalization is based on a repository we built for the common namespace prefixes (such as `rdf`, `owl`, and `foaf`), which the user can also edit and extend. The MashQL editor uses heuristics for other cases — for example, taking the last part of a URI, after "#" (if "#" doesn't exist, the part after "/"). The result should be at least three characters and start with a letter; otherwise, it takes the last two parts of the URI, and so on. Our experiments on many data sets showed that this works in most cases, but there's no guarantee of always producing elegant normalization.

Verbalization

To further improve MashQL's elegance, we implemented a verbalization mode. After editing a restriction, the editor verbalizes all control boxes and lists in this restriction, meaning it converts their content into friendly text, and displays the verbalization instead. If the user returns to edit a restriction by clicking on it, the editor switches this restriction to edit mode (control boxes and lists are made visible again) and all other restrictions switch to verbalization mode — that is, only one restriction will have the edit mode at a time. This makes the query representation closer to natural language and facilitates query validation by users.

Scalable Lists

When querying large data sets, the usual drop-down list demonstrates its lack of scalability. We developed a scalable and friendly list that supports search, auto-complete, and sorting tasks based on *rank* and *asc/desc* (for ascending/descending). If the user selects *rank*, the editor orders items and nodes based on how many nodes point to them. Our list also supports scalable scrolling: the first 50 results are displayed first, but users can scroll to get the next, arbitrarily middle, or last 50. Furthermore, our list lets users select either instances (that is, any URI in the data set) or types of instances (that is, instances that have the predicate `rdf:type`, such as Author, Person, University).

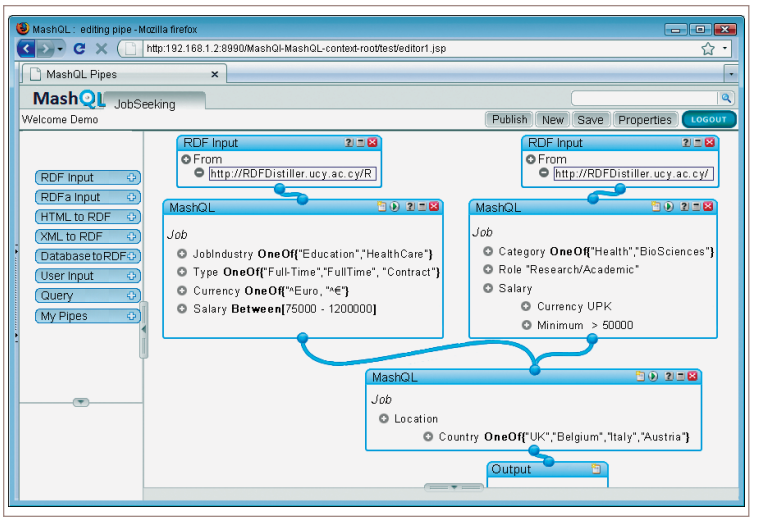


Figure 4. Screenshot of the server-side MashQL editor. The editor follows the same visualization that Yahoo Pipes uses, showing that we can use MashQL to query and mash up the data Web as simply as filtering and piping Web feeds.

Performance Considerations

When formulating a MashQL query, the editor queries the data set in the background to generate the list of next choices. In such an interactive setting, the response time should be small,[3] preferably less than 100 ms. However, as our initial evaluations showed,[1] such a short response time can't be achieved through existing SQL-backed RDF querying technologies[4,5] over large graphs that can't fit in memory. This is because querying a graph stored in a relational table (*S*, *P*, *O*) involves many self-joins, which are expensive even if the table is well indexed. For large data stores, we suggest constructing the *graph signature*, a graph index designed to enhance the MashQL editor's interactivity. The idea (similar to XML summaries[6]) is to generate a small summary of an RDF graph, so that this summary can answer the editor's background queries more quickly than querying the original graph. The graph signature groups into the same category all RDF nodes with the same set of outgoing paths. A category is the set of all subjects that have exactly the same property labels, and the objects of each of their properties belong to the same categories. Definition 2 lists the formal definitions.

Figure 5 shows an RDF graph and its graph signature, also represented in a table for-

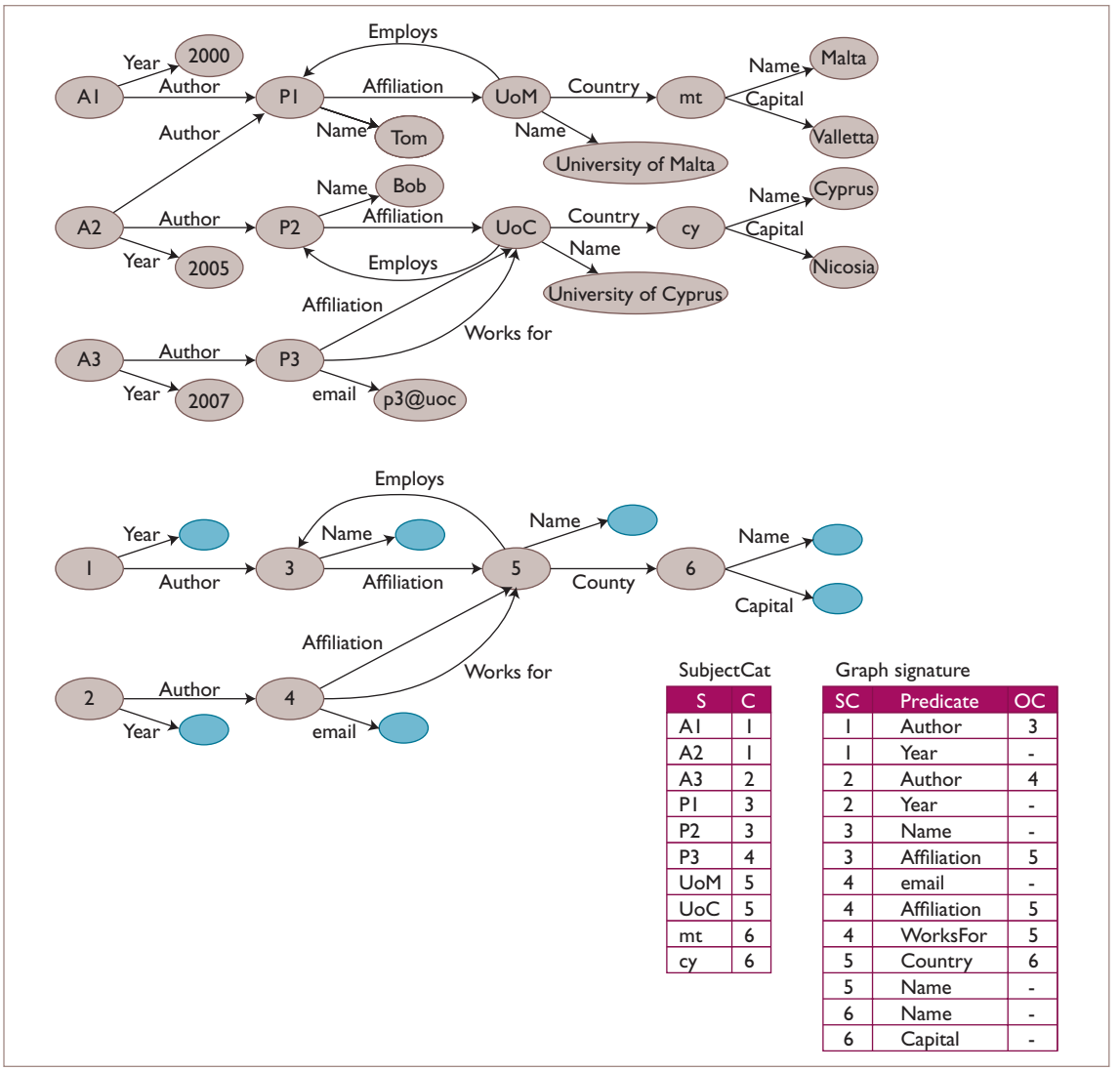
Def. 8: Two RDF subjects S_1 and S_2 have the same category C_i, if and only if

1. They share the same property labels: there exist $(S_1\ P_1\ O_{1,1})\ \ldots\ (S_1\ P_m\ O_{1,m})$, $m > 0$, and $(S_2\ P'_1\ O_{2,1})\ \ldots\ (S_2\ P'_n\ O_{2,n})$, $n > 0$, such that the set $\{P_1, \ldots, P_m\}$ of properties of S_1 is equal to the set of properties of $S_2\{P'_1, \ldots, P'_n\}$.
2. The objects of each property of S_1 and S_2 belong to the same category: for each property P_i of S_1 and S_2, its corresponding objects $O_{1,i}$ and $O_{2,i}$ belong to a same category C_j.

Def. 9: A Category Signature is a set of triples of the form $<S_C, P, O_C>$, where S_C is a category of some subjects, P is a property label, and O_C is a category of some objects.

Def. 10: Graph Signature is the set of all category signatures. A Graph Signature is also a directed labeled graph as RDF.

Definition 2. The formal definition of graph signature. We present it here in RDF terms.[1]



SubjectCat

S	C
A1	1
A2	1
A3	2
P1	3
P2	3
P3	4
UoM	5
UoC	5
mt	6
cy	6

Graph signature

SC	Predicate	OC
1	Author	3
1	Year	-
2	Author	4
2	Year	-
3	Name	-
3	Affiliation	5
4	email	-
4	Affiliation	5
4	WorksFor	5
5	Country	6
5	Name	-
6	Name	-
6	Capital	-

Figure 5. Graph signature index. This example data graph and its graph signature index show both graph and table formulas.

mat. The `SubjectCat` table indexes the extent of the categories. Notice that `P1` and `P2` are both assigned category 3 because they share the same set of property labels {`Affiliation, Name`}. `P3` is assigned another category 4 because its properties aren't the same as `P1` and `P2`. Finally, `A1`, `A2`, and `A3` have the same properties, but `A3` is assigned a different category

Table 1. Statistics about DBPedia and the graph signatures.

	Number of	B32	B16	B8	B4
Original data	Unique triples	32 million	16 million	8 million	4 million
	Unique subjects	9.4 million	6 million	4 million	2.6 million
	Unique predicates	35	35	34	34
	Unique objects	16 million	8.7 million	4.7 million	2.5 million
	Data size	6.7 Gbytes	3.1 Gbytes	1.4 Gbytes	550 Mbytes
Graph signatures	Indexing time (seconds)	2,378	1,177	528	285
	Unique categories	6,000	32,000	16,000	6,000
	Triples in graph signature	165,000	486,000	185,000	56,000

(see Def. 8.2) because the object of its author has another category. That is, the category of `A1.Author` and `A2.Author` is 3, whereas the category of `A3.Author` is 4.

Querying the graph signature is similar to querying an RDF graph. Our query model is much simpler than SPARQL — it is, $O_n|P_n$:{O_1 P_1 O_2 P_2 . . . P_n O_n}, where O_i is a node, P_i is a predicate, and each can be a constant or a variable. Notice we only retrieve/project the last node or predicate label ($O_n|P_n$), which is sufficient for our query formulation purposes. We can illustrate how to evaluate such a query with the following example: suppose we need to know, "What are the properties of the countries of the affiliations of the authors of `A3`?" To answer this query, we must first look up the category of `A3`, which is 2, and then generate the SPARQL query: `SELECT P WHERE {:2 :Author ?O1)(?O1 :Affiliation ?O2)(?O2 :Country ?O3)(?O3 ?P ?O5)}`. Answering such queries from the graph signature is fast because its size is typically small.

Given an RDF graph with size n, where n is the number of the triples, we can summarize this graph by m triples, where $1 \leq m \leq n$. If a graph is heterogeneous — that is, there are no similar subjects (Def. 8) — its summary size is the same as its original size. In practice, RDF graphs tend to be structurally homogenous. To reduce the size of the graph signature, we can exclude some irrelevant annotation triples (such as `Label`, `Comments`) and normalize the triples that indicate equivalence (such as `SameAs`, `Redirect`) before computing the graph signature. Our full report has more details and also explains that the graph signature follows an opposite approach to XML summaries, which are based on incoming (rather than outgoing) paths.[1]

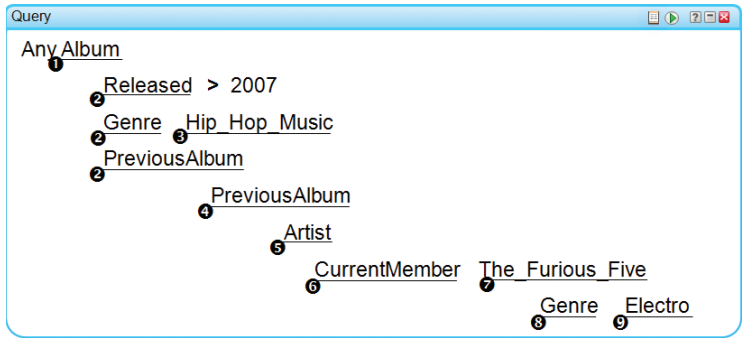
Evaluation

Our evaluation is based on DBPedia, the RDF version of Wikipedia, which includes 32 million RDF triples (6.7 Gbytes). From this, we extracted four subgraphs: B32 contains 32 million triples, B16 contains 16 million triples from B32, B8 contains 8 million, and B4 contains 4 million (see Table 1). We didn't use any sorting before partitioning (such as `Create B16 As select * from B32 where rownum <16000001`). We loaded each partition into a table in Oracle 11g, which is installed on a 2-GHz server with 2 Gbytes of RAM.

As Table 1 shows, the time cost of building a graph signature is linear with data size: 4 million triples cost 285 seconds, and 8 million cost 528 seconds. The behavior of the index with regard to the number of triples is scalable — for example, 165,000 triples summarize 32 million triples, and 56,000 summarize 4 million. Notice that B16 and B8 generated bigger summaries than B32. This is because we found more similarities when all the data is put together.

We aren't interested in evaluating the final query's execution — rather, we want the queries MashQL performs in the background to generate the "next" drop-down list (that is, MashQL's response time during user interaction). To formulate the query in Figure 6 over DBPedia, the user first selects the subject from a list. The query producing this list is annotated with a circled number 1. The user then selects a property of this subject from a list, produced by that query, and so on. A table in the figure shows the cost.

As this experiment shows, the cost for all queries remains within a few milliseconds, regardless of the data size or the length of the join-path expressions. This is because the graph



1. `O:(?S Type ?O)`
2. `P:(?S Type Album)(?S ?P ?O1)`
3. `O:(?S Type Article)(?S Genre ?O)`
4. `P:(?S Type Album)(?S PreviousAlbum ?O1)(?O1 ?P ? O2)`
5. `P:(?S Type Album)(?S PreviousAlbum ?O1)(?O1 PreviousAlbum? O2)(?O2 ?P ?O3)`
6. `P:(?S Type Album)(?S PreviousAlbum ?O1)(?O1 PreviousAlbum? O2)(?O2 Artist ?O3)(?O3 ?P ?O4)`
7. `O:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? O2)(?O2 Artist ?O3)(?O3 CurrentMember ?O4)`
8. `P:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? O2)(?O2 Artist ?O3)(?O3 CurrentMember The_Furious_Five)(The_Furious_Five ?P ?O5)`
9. `O:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? O2)(?O2 Artist ?O3)(?O3 CurrentMember The_Furious_Five)(The_Furious_Five Genre ?O5)`

Query	(B32) 32 M	(B16) 16 M	(B8) 8 M	(B4) 4M
Q_1	0.003	0.003	0.003	0.003
Q_2	0.002	0.002	0.002	0.002
Q_3	0.005	0.004	0.003	0.003
Q_4	0.005	0.004	0.004	0.004
Q_5	0.005	0.004	0.004	0.004
Q_6	0.005	0.005	0.005	0.005
Q_7	0.007	0.007	0.007	0.007
Q_8	0.005	0.005	0.005	0.005
Q_9	0.007	0.007	0.007	0.006

Figure 6. Experimental queries and the cost in milliseconds. Here, we present a MashQL query over DBpedia, the nine background queries, and their time cost in milliseconds over different portions of the data set.

signature's size is small compared to the original graph. More data sets, experiments, and comparisons with Oracle's Semantic Technology appear in the full report.[1]

We plan to extend this work in several directions. We will introduce a search box on top of MashQL to allow keyword search and then use MashQL to filter the retrieved results. To let people use MashQL in a typical data integration scenario, we'll support several reasoning services, including `subtype`, `sub-property`, and `part-of`. We're also collaborating with colleagues to use MashQL as a business rules language, which will let us include several reaction and production operators and aggregation functions. Last but not least, we're extending the graph signature for general-purpose query optimization.

Acknowledgments

The SEARCHiN project (FP6-042467, Marie Curie Actions) partially supported this research.

References

1. M. Jarrar and M. Dikaiakos, *Querying the Data Web*, tech. report TR-08-04, Dept. Computer Science, Univ. of Cyprus, Nov. 2008; www.cs.ucy.ac.cy/~mjarrar/TAR200904.pdf.
2. M. Jarrar and M. Dikaiakos, "A Data Mashup Language for the Data Web," *Proc. Linked Data on the Web Workshop* (LDOW2009), CEUR Workshop Proceedings, 2009; http://ceur-ws.org/Vol-538/ldow2009_paper14.pdf.
3. R. Miller, "Response Time in Man-Computer Conversational Transactions," *Proc. December 9-11, 1968,*

Fall Joint Computer Conf., Part I, ACM Press, 1968, pp. 267–277.
4. E.I. Chong et al., "An Efficient SQL-Based RDF Querying Scheme," *Proc. 31st Int'l Conf. Very Large Data Bases*, VLDB Endowment, 2005, pp. 1216–1227.
5. T. Neumann and G. Weikum, "The RDF-3X Engine for Scalable Management of RDF Data," *VLDB J.*, vol. 19, no. 1, 2010, pp. 91–113.
6. S. Nestorov et al., "Representative Objects: Concise Representations of Semistructured, Hierarchical Data," Proc. 13th Int'l Conf. Data Eng., IEEE CS Press, 1997, pp. 79–90.

Mustafa Jarrar is an assistant professor at the University of Birzeit in Palestine. His research interests include the Semantic Web, ontology engineering, databases, Web 2.0, and data mashups. Jarrar has a PhD in computer science from Vrije Universiteit Brussel in Belgium. Contact him at mjarrar@birzeit.edu.

Marios D. Dikaiakos is an associate professor of computer science at the University of Cyprus. His research interests include network-centric computing systems and Web technologies. Dikaiakos has a PhD in computer science from Princeton University. He's a senior member of the ACM and a member of the IEEE Computer Society and the Technical Chamber of Greece. Contact him at mdd@cs.ucy.ac.cy.