On-the-fly Integration of Static and Dynamic Linked Data*

Andreas Harth[1], Craig A. Knoblock[2], Steffen Stadtmüller[1], Rudi Studer[1], and Pedro Szekely[2]

[1] Karlsruhe Institute of Technology (KIT)
{harth,steffen.stadtmueller,studer}@kit.edu
[2] University of Southern California (USC)
{knoblock,szekely}@isi.edu

Abstract. The relevance of many types of data perishes or degrades over time; to support timely decision-making, data integration systems must provide access to live data and should make it easy to incorporate new sources. We outline methods, based on web architecture that enable (near) real-time access to data sources in a variety of formats and access modalities. Our methods also enable rapid integration of new live sources by modeling them with respect to a domain ontology, and by using these models to generate a Linked Data interface to access them. Finally, we present initial experimental results of a scenario involving several static and dynamic sources from the web.

1 Introduction

In recent years the number of web data sources and web APIs has increased tremendously. The explosion is due to the overwhelming growth of available Linked Data (e.g., [5]), the deployment of large numbers of sensors in many types of environments (e.g., [1]) as well as the trend to offer all kinds of functionalities over the web (e.g., [13]). In principle, these trends open up new opportunities for building applications since data and functionalities are abundant and "just" have to be integrated for, e.g., decision making.

However, integration of the different sources and APIs requires programmatic access over heterogeneous data sources that use different formats and protocols following different paradigms for accessing dynamic data. Further, the ability to quickly introduce new data sources into the application requires manual alignment and modeling, which is a labor-intensive and thus time-consuming process.

Integration of data sources in a variety of data source formats and access mechanisms is a difficult problem, especially as we want to add sources on-the-fly, i.e., while the application is running.

* This research is based upon work supported in part by ONR Global under grant number N62909-13-1-N024, in part by the European Commission's Seventh Framework Programme FP7/2007-2013 under PlanetData grant number 257641, and in part by the National Science Foundation under grant number 1117913.

One challenge is to decide on the handling of static and dynamic sources. In many systems, dynamic sources push their updates in a streaming fashion to the target; the target systems handle static and dynamic data sources differently. However, most sources on the web follow a request/response communication pattern based on polling of sources, with support for caching on the protocol level. We investigate the use of polling for frequently changing sources; the only difference between sources are their update rates. The update frequency of sources we currently consider is relatively low, ranging from seconds to months. One research question we address is whether a uniform interface to all data sources based on polling is a feasible approach in such a scenario. A main benefit of using a uniform interface is a simplified and more robust overall architecture.

Another challenge is the amount of manual effort involved in building and maintaining an integration system. We want to reduce the time needed for including a new live source into a constellation of integrated sources. Thus, we aim for a flexible approach based on declarative specifications rather than procedural code. Declarative specifications are used in the modeling of sources, and in the rules that encode how data sources are accessed and how data items relate to each other.

There are different strands of research targeting real-time data access. We are exploring the trade-off outlined by [19]. Many current Linked Data systems use a warehousing approach: all data is loaded into a central repository and then queried, with a considerable time lag between data loading and the first query. There is work on Linked Data Streams (e.g., [11, 15]) which offers one possible route to achieving real-time access. In contrast, our proposed system operates on dynamic data compatible with the polling approach that many of the web sources follow.

The overall benefits from the point of view of an application of such a method and apparatus are as follows:

– We may achieve real-time access to data integrated from several sources, some of them static and some of them dynamic.
– We can quickly integrate new data sources, as we use standard software interfaces to poll the current state of resources at specified time intervals or receiving updates and reacting to them, easing the transition from static to dynamic sources.
– We can quickly integrate new data sources with the help of an application allowing for the semi-automatic modeling and lifting of non-RDF sources to the Linked Data level.

We begin with a scenario in Section 2, followed by an architectural overview in Section 3. The main components, Karma [17] for modeling sources, and Data-Fu [16] for specifying and evaluating integration programs, are covered in Sections 4 and 5. We extended these components to satisfy the requirements set forth in the current work. We discuss initial experimental results in Section 6. We review related work in Section 7 and conclude with Section 8.

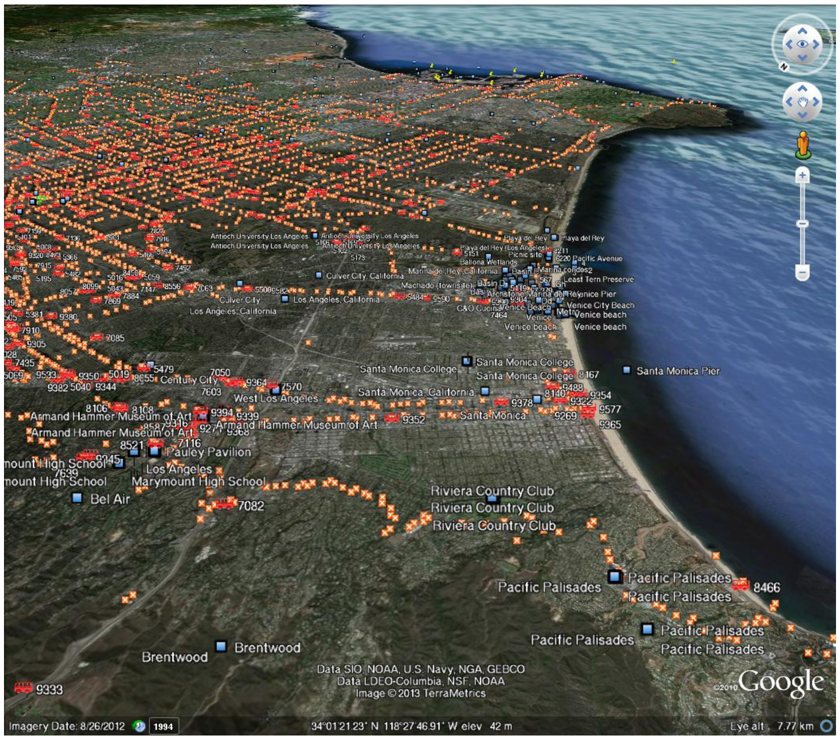


Fig. 1. View of all data: bus stops (orange), buses (red), port data (yellow), campus cruiser (green), points of interest (blue).

2 Scenario

The scenario we consider concerns users who want to go to an event (identified via e.g., Eventful) in the Los Angeles metro area. Based on the users' current location, they want to know when to be at the next bus stop to arrive on time.

The scenario includes a real-time visualization of the area that is augmented with current data from multiple sources to help a user make well-informed and timely judgments. Finally, we want to be able to integrate new sources rapidly; consider the addition of a new dynamic data source, for example the USC campus cruiser source which provides access to campus taxis via a JSON API.

Figure 1 shows the final rendering of the integrated data in Google Earth[3]. Different sources have different update intervals; the bus sources, for example, are updated every several seconds, while others are updated once a month, following the manually determined update interval of the original API. A list of sources used in the prototype together with performance measurements can be found in Section 6.

[3] Final KML available at `http://people.aifb.kit.edu/aha/2013/d3/index.kml`.

3 Overall Approach

We employ Linked Data as a uniform abstraction for sources. Sources following the Linked Data abstraction provide a standardized small set of supported operations and a uniform interface for both their data payload (RDF) and their fault handling (HTTP status codes). Such a common abstraction allowing the manipulation of states as primitives enables us to specify the interactions between components declaratively both on the operational and data levels. Consequently, we require mechanisms to quickly bring data sources to the Linked Data abstraction.

In our system we consider the following data artifacts:

- $\mathcal{D}$ original data sources
- $\mathcal{O}$ domain ontology
- $\mathcal{M}$ Karma models based on domain ontology
- $\mathcal{L}$ data sources as Linked Data
- $\mathcal{P}$ data integration program, consisting of queries and rules

The main runtime components are Karma and the Data-Fu interpreter. Karma uses the data sources $\mathcal{D}$ in conjunction with the domain ontology $\mathcal{O}$ to create the models $\mathcal{M}$; based on $\mathcal{M}$, an execution component creates Linked Data interfaces $\mathcal{L}$ to the original data sources. The Data-Fu interpreter evaluates programs $\mathcal{P}$ that operate on $\mathcal{L}$. We currently manually specify the update intervals for the evaluation of $\mathcal{P}$, which ranges from seconds to months. Finally, Google Earth visualizes the query results. Figure 2 illustrates the components of our system.

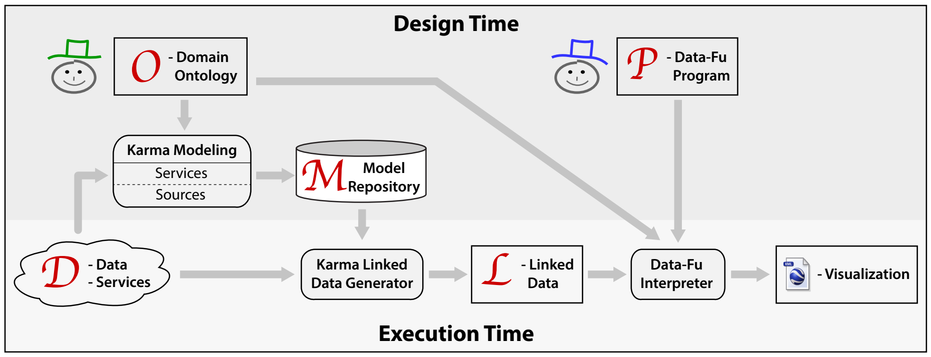


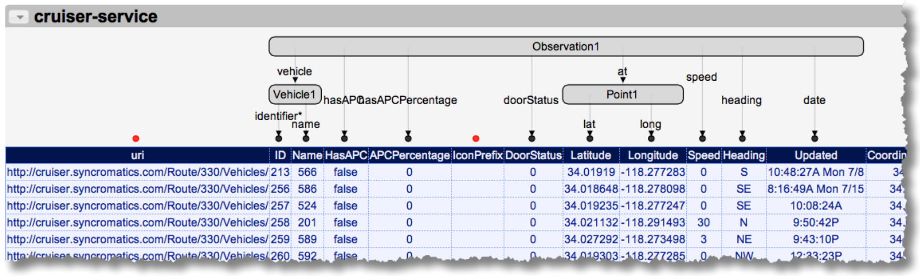
Fig. 2. Architectural overview.

To add a source (at design time), there are two places where manual effort is involved: first, a user has to model a new source in Karma; second, a user has to create a small Data-Fu program to specify what part of the data should be fetched via rules, together with a query that returns the final results. Once the models and the rules are deployed and linked in the main KML file, the new sources are shown in the visualization component after the next refresh.

4 Rapidly Modeling APIs: Karma

One of the challenges in our scenario is the rapid integration of new data sources, such as the service that provides status and location information for vehicles that transport students to and from the USC campus (campus cruiser service). The integration requires solving two key problems. First, the data from a new source needs to be expressed in the same data format and vocabulary as the data from the already integrated sources. Second, since we are dealing with real-time information, it is necessary to generate the corresponding RDF dynamically from the JSON data that the service returns. In this section we describe how we support the automatic modeling of sources to make it possible to rapidly integrate both static and dynamic sources into the system.

To bring in a new source or service, the first task is to produce a model that defines the mapping from the source data or the data returned by the service to a common ontology. In previous work [9, 18], we developed a system called Karma that automates the process of mapping sources and services to an ontology. Karma learns from the sources and services it has already seen to suggest models that users can easily refine using a graphical user interface. Figure 3 shows the model for the campus cruiser service.



uri	ID	Name	HasAPC	APCPercentage	IconPrefix	DoorStatus	Latitude	Longitude	Speed	Heading	Updated	Coordin
http://cruiser.syncromatics.com/Route/330/Vehicles/	213	566	false	0		0	34.01919	-118.277283	0	S	10:48:27A Mon 7/8	34
http://cruiser.syncromatics.com/Route/330/Vehicles/	256	586	false	0		0	34.018648	-118.278098	0	SE	8:16:49A Mon 7/15	34.
http://cruiser.syncromatics.com/Route/330/Vehicles/	257	524	false	0		0	34.019235	-118.277247	0	SE	10:08:24A	34.
http://cruiser.syncromatics.com/Route/330/Vehicles/	258	201	false	0		0	34.021132	-118.291493	30	N	9:50:42P	34.
http://cruiser.syncromatics.com/Route/330/Vehicles/	259	589	false	0		0	34.027292	-118.273498	3	NE	9:43:10P	34
http://cruiser.syncromatics.com/Route/330/Vehicles/	260	592	false	0		0	34.019303	-118.277285	0	NW	12:33:23P	34

Fig. 3. Screenshot of Karma showing the model of the campus cruiser source.

Karma supports the rapid modeling of both static and dynamic sources. In the case of a static source, a sample of the records are loaded into Karma and then the system can proceed to generate a model based on the data. Similarly, Karma handles dynamic sources, such as a REST-based services [17]. Instead of providing sample records, the user provides sample HTTP GET queries as shown in the first column of Figure 3. The source would then be invoked to generate the resulting output, which is JSON in case of the USC campus cruiser source. Then the system would automatically produce a model of the hierarchical JSON output by first applying a learned classifier to recognize the attributes of the output and then generate an overall model that describes the relationships between the attributes. The resulting model is shown in the top of the Figure. Once the user has verified the model, the final source description is generated and saved in R2RML[4], which we extended to support hierarchical data.

The second task is to use the model on the fly to automatically produce RDF data in terms of the shared domain ontology for a given input to a service. This task is referred to as lifting since we have to take the JSON output and produce a corresponding set of RDF data that uses the vocabulary from the ontology. Karma is able to perform the lifting task automatically using the R2RML model built in the first task. When Karma built the model in the first task, it parsed the invocation URLs extracting the input arguments, and recorded this information in the model. In the second task, when Karma receives a service request, it automatically composes the invocation URL from the input arguments, invokes the service, receives the JSON output, uses the model to automatically map the service output into RDF in terms of the domain ontology, and finally returns the RDF.

Please note that Karma supports the modeling of tabular sources. Thus, tabular formats such as CSV and TSV can be directly loaded into Karma. Hierarchical formats such as JSON are transformed to a tabular format before modeling.

5 Executing Integration Programs: Data-Fu

Having access to all sources via a Linked Data interface, we can devise a Data-Fu program [16] that collects the data and evaluates queries over the data, while taking into account the semantics of ontology constructs. Consider the LA Metro API: there are Linked Data resources about the routes, that contain links to the relevant bus stops and the buses currently running on the route. To be able to derive query results on the moving objects, we require to follow links from the routes to the bus stops, and to the buses. We specify the method of traversal in rules. Data-Fu programs live under a URI; programs are started via POSTing to the program URI, optionally providing parameters that are used during program evaluation. Once the program evaluation finishes, the query results are available for GETing under the query URI. In addition to the research presented in [16], we introduce a concrete rule syntax based on Notation3[5] and a web interface for Data-Fu programs in a realistic scenario.

A Data-Fu program consists of i) facts and actions, ii) queries and iii) rules. Facts are RDF triples; actions are HTTP operations, such as HTTP GET or PUT. In our current scenario, we limit ourselves to read-only (GET) operations. Data-Fu supports conjunctive queries that either return variable bindings (select queries) or triples (construct queries). Rules consist of a conjunctive query in the antecedent and either triple patterns (for deduction rules) or actions (for interaction rules) in the consequent.

[4] http://www.w3.org/TR/r2rml/

[5] http://www.w3.org/DesignIssues/Notation3

The state manipulation abstraction and the declarative specifications describing the interplay between resources bring the following major benefits:

- Scalable execution: declarative specifications can be automatically parallelized more easily than imperative programs.
- Uniform and consistent error handling: instead of being confronted with source-specific error messages, universal error handlers can be realized.
- Substitution of resources: replacing a source only requires modeling the source in Karma in terms of the domain ontology. Such flexibility is required as in networked applications the underlying base resources may become unavailable and thus may put the entire application at risk.
- More flexible and cleaner specifications of interactions: the specifications can concentrate on the business logic of the intended interaction, while the operational interaction between components can be automated due to their standardized interfaces.

The Data-Fu interpreter executes resource interactions specified in Data-Fu programs. The engine holds the current state of the interactions as well as the functionality to invoke interactions with resources as defined in the rules. In practice, we translate a Data-Fu program into a logical dataflow network, which is then transformed into an evaluator plan that actually implements the dataflow network. For performing the HTTP operations, we use a multi-threaded lookup component that carries out the HTTP requests and streams the resulting triples into the dataflow network. The dataflow network then processes the triples and provides information about further HTTP requests to the lookup component. The division into dataflow network and lookup component is similar to the one in [10].

The following program collects information on busses from the wrapped LA Metro API. We omit prefix declarations which can be found on prefix.cc[6]. First, we specify an HTTP GET request which retrieves RDF describing the bus routes.

```
[] http:mthd httpm:GET ;
   http:requestURI <http://openeanwrap.appspot.com/route/> .
```

Then, we specify rules that trigger GET requests on `:Route` and `geo:Point` URIs.

```
{ ?route rdf:type <http://km.aifb.kit.edu/people/aha/2013/d3/cruiser#Route>.
} log:implies {
  [] http:mthd httpm:GET ;
     http:requestURI ?route . } .

{ ?stop rdf:type <http://www.w3.org/2003/01/geo/wgs84_pos#Point>.
} log:implies {
  [] http:mthd httpm:GET ;
     http:requestURI ?stop . } .
```

[6] `http://prefix.cc/`

We also may specify deduction rules that allow for the encoding of the semantics of vocabulary terms. One can also use pre-defined rule-sets that specify the entailment of certain OWL constructs[7].

Finally, we specify a query that returns an identifier, a label, and latitude/longitude information for all of the data that flows through the network.

```
<q1> qrl:select ( ?x ?label ?lat ?lon ); qrl:where {
   ?x rdfs:label ?label .
   ?x geo:long ?lon .
   ?x geo:lat ?lat . } .
```

The only thing left to do is to invoke the program and convert the query results to a format that our visualization toolkit understands. Thus, we request the query results in SPARQL XML Query Result format, and apply an XSLT which transforms the query results into KML for display.

The amount of data fetched during the evaluation of Data-Fu programs can range in the tens of MBs. As fetching all the data at query time is infeasible, we decouple evaluation of programs from the access to query results. The evaluation of the program can be triggered with a HTTP POST request (optionally including input data such as bounding box); the program then evaluates and the query results are updated. User agents can then GET the query results under the query's URI. One particular aspect we want to exploit in future versions are update mechanisms based on headers for cache control and expiry of documents. Thus, the interpreter can use the server-provided expiry headers to update query results automatically once the validity of one of the sources expires.

Data-Fu integration programs can be used for slowly changing sources (update intervals measured in months) and frequently changing sources (update intervals measured in seconds). Initial experiments on running Data-Fu programs in a realistic settings with real data are described in the following section.

6 Experiments and Evaluation

We implemented a prototype data integration system to test the feasibility of our method. We first describe the domain ontology $\mathcal{O}$ for the scenario, then we explain how we constructed Linked Data interfaces $\mathcal{L}$ to the original data sources $\mathcal{D}$, and finally we show performance results for evaluating the integration programs $\mathcal{P}$.

The domain ontology consists of several terms from `dc`, `dct`, `foaf`, `geo` and others. In the following we describe how we model moving objects. There is a class `:Vehicle`; each location reading is part of an `:Observation`, which connects to a `:Vehicle` and contains a `geo:Point`, and optionally additional information such as heading or a timestamp. Such a representation is sufficient to query the moving objects and display the results on a map.

The types of relevant original data sources $\mathcal{D}$ include:

[7] e.g., `http://semanticweb.org/OWLLD/`

– Static sources such as 2D maps, 3D models and point-of-interest (POI) data from files in XML, JSON or CSV, Linked Data, and web APIs.
– Dynamic sources producing state updates (e.g. of moving objects or event information), continuously or periodically, possibly with additional spatial and temporal properties, through web APIs.

All sources with a Linked Data interface $\mathcal{L}$ support dereferencable URIs. However, we require means to query for sets of URIs with certain properties. Thus, we arrive at additional lookup or query possibilities for sources:

– Keyword lookup, which returns a list of URIs matching the keyword
– Point lookup, which returns a list of location URIs near the specified point
– Bounding box lookup, which returns a list of location URIs within the bounding box

Table 1 lists properties of the sources as Linked Data $\mathcal{L}$ and shows the query time for retrieving the information necessary to render points on the map. The table also contains the lines of code and the number of rules per integration program for each source. One objective of our approach is that we are able to quickly add new sources: the lines of code for data integration programs $\mathcal{P}$ give an indication as to the effort required to add sources.

Table 1. Data sources.

Data Source	**Input**	**Size**	**Triples**	**Runtime**	**LoC**	**Rules**
Marine vessels (AIS)	bounding box	39 KB	224	0:01 min	33	2
Campus Cruiser	-	65 KB	406	0:01 min	19	1
Crunchbase	keyword	14 MB	87,477	12:08 min	31	2
Eventful	keyword	55.3 MB	293,975	57:13 min	46	5
GADM	bounding box	10.8 MB	68,640	7:08 min	33	2
GeoNames (Wikipedia)	bounding box	2.3 MB	14,746	1:00 min	32	2
GeoNames (cities)	bounding box	304 KB	2,629	0:04 min	32	2
LastFM	point	10.1 MB	77,431	13:39 min	34	2
LA Metro (vehicles)	-	130 KB	860	0:01 min	39	5
LA Metro (routes)	-	2.5 MB	17,349	1:03 min	43	6
OpenStreetMap	bounding box	8.4 MB	57,904	8:29 min	36	2
Wikimapia	bounding box	8.8 MB	51,714	0:19 min	41	3

The table also shows that the Data-Fu interpreter scales from small sources (hundreds of triples) to sources of moderate size (hundreds of thousands of triples). Small sources (such as marine vessels and LA Metro vehicles) can be repeatedly polled to retrieve their current state. The number of moving objects changes depending on the time of day; on a weekday noon we have seen 56 buses and 72 vessels in the current configuration.

We have anecdotal evidence that our infrastructure is capable of much faster polling than supported by the sources. Our infrastructure can handle all LA Metro vehicles; however, we restrict the number of tracked bus lines in the prototype to avoid overloading the source. We additionally throttle lookups and restrict the number of parallel lookups to four. Future work will have to include further investigation of performance and stability of sources, and mechanisms to alleviate their load, as we had repeatedly to deal with APIs that were down.

We present initial measurements of three configurations for transforming $\mathcal{D}$ to $\mathcal{L}$: transformation via procedural wrappers on Google App Engine (GAE) and on a local Tomcat installation, and transformation via Karma. The Crunchbase wrapper (JSON to RDF in Java) deployed GAE has an overhead of -29 %, that is, the wrapped source is actually responding faster than the original source. Our hypothesis is that the wrapper on GAE is closer in the network (12 hops/6.1 ms measured with `traceroute`) than the original API (22 hops/161.5 ms), while the remaining data transfer is routed via Google's optimized internal infrastructure.The AIS wrapper (XML to RDF via XSLT) deployed on Tomcat incurs an overhead of 246 % (76 ms average for direct API access vs. 263.8 ms for wrapped access). The campus cruiser source modelled and transformed via Karma (JSON to RDF in Karma) incurs an overhead of 29 %. Further studying the overhead for transforming original sources is part of future work.

7 Related Work

We survey related work in two areas: sensor systems and web systems. In these areas we differentiate between two fundamental architectures for accessing dynamic data: pull-based (polling) and push-based (streaming) data acquisition. We distinguish between source (i.e., the site where the data originates, such as the sensor) and target (i.e., the site where data and queries are processed). We further present related work in the area of declaratively specified networking.

There has been a considerable amount of research in the area of streaming data, i.e. data that arrives at intervals. The goal of stream data management systems is to evaluate queries over high-velocity data, often in the context of physical sensors. Systems are able to process aggregation queries (often including query clauses with a temporal component) over time-stamped data.

With pull-based (polling) data acquisition, the target accesses the sources in intervals and coordinates data access centrally. Pull-based access is used, for example, in TinyDB [12], which refreshes data from sources in specified intervals. With push-based (streaming) data acquisition, the data sources autonomously decide when to communicate data.

We also distinguish between pull-based and push-based systems in related work in the area of query evaluation over web sources. Polling is the predominant mechanism used in systems based on RESTful architecture, such as the web, and is supported in HTTP. Hartig et al. describe a system that allows for query evaluation over Linked Data [8]. Fionda et al. describe a graph query language which supports the combined querying and navigation of Linked Data [7]. Umbrich et al. describe a live query system which includes reasoning to improve the recall of query results [20]. All of these systems use polling via HTTP GET. We combine several features of these systems, with the addition of interaction rules and generic rule-based reasoning.

In contrast to the pull-based approaches, there are several proposals for streaming query processing over RDF. Notable references in the area of Linked Data include Linked Stream Data [15] and streaming SPARQL systems such as C-SPARQL [3], SPARQLStream [6] and CQUELS [11]. C-SPARQL has been used in a system which includes means for inferencing [4]. Unlike these and other streaming approaches (see, for example, [14] who also address a scenario combining static and dynamic sources), we solely build on the Linked Data abstraction and polling to gain (near) real-time access to data.

Although there are historical datasets which provide time-stamped observations of e.g. weather phenomena, and the W3C Semantic Sensor Networks XG[8] specified a vocabulary for describing sensors and observations, to date there is no established protocol for streaming over the web, and the availability of streaming sources online is scarce. Finally, there has been work on declaratively specifying data-centric programs in a cloud setting [2] which encompasses batch processing on large amounts of data. We use rule-defined integration programs for targeted access to web data sources in real-time.

8 Conclusion and Future Work

We have shown the feasibility of on-the-fly integration of static and dynamic sources for applications that consume Linked Data. We believe that current web architecture offers the right abstraction and allows for the cost-effective implementation of such systems. Linked Data already supports the integration of static data and the same mechanism can be used to achieve real-time functionality. We have shown how to provide interactive access to data, which requires to execute integration pipelines at query time within seconds for real-time sources.

We can envision to extend our research in the following directions:

- Temporal aspects: in the prototype, we are concerned with showing the current state of the tracked objects. We would like to extend the temporal model to include past states of the objects, and potentially future (predicted) states.
- Fault handling: although our current system is robust in terms of failing APIs (the failing source is just not displayed in the refresh cycle), we need to further investigate fault handling and error reporting.
- Reducing network traffic: we plan to experiment with HTTP expires and caching mechanisms to reduce the amount of data transferred while keeping the query results up to date.

References

1. I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine*, 40(8), 2002.
2. P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: datalog in time and space. In *Proc. of the 1st Int'l Conference on Datalog Reloaded*, Datalog '10, 2011.

[8] `http://www.w3.org/2005/Incubator/ssn/XGR-ssn/`

3. D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Record*, 39(1), 2010.
4. D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, Y. Huang, V. Tresp, A. Rettinger, and H. Wermser. Deductive and inductive stream reasoning for semantic social media analytics. *IEEE Intelligent Systems*, 25(6), 2010.
5. C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3), 2009.
6. J.-P. Calbimonte, Ó. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proc. of the 9th Int'l Semantic Web Conference*, ISWC '10, 2010.
7. V. Fionda, C. Gutierrez, and G. Pirró. Semantic navigation on the web of data: specification of routes, web fragments and actions. In *Proc. of the 21st Int'l Conference on World Wide Web*, WWW '12, 2012.
8. O. Hartig, C. Bizer, and J.-C. Freytag. Executing sparql queries over the web of linked data. In *Proc. of the 8th Int'l Semantic Web Conference*, ISWC '09, 2009.
9. C. Knoblock, P. Szekely, J. L. Ambite, A. Goel, S. Gupta, K. Lerman, M. Muslea, M. Taheriyan, and P. Mallick. Semi-automatically mapping structured sources into the semantic web. In *Proc. of the 9th Extended Semantic Web Conference*, ESWC '12, 2012.
10. G. Ladwig and T. Tran. Sihjoin: Querying remote and local linked data. In *Proc. of the 8th Extended Semantic Web Conference*, ESWC '11, 2011.
11. D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proc. of the 10th Int'l Semantic Web Conference*, ISWC '11, 2011.
12. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the 2003 ACM SIGMOD Int'l Conference on Management of Data*, SIGMOD '03, 2003.
13. E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A domain-specific language for web apis and services mashups. In *Proc. of the 5th Int'l Conference on Service-oriented Computing*, ICSOC 2007, 2007.
14. E. Ruckhaus, J.-P. Calbimonte, R. Garcia-Castro, and Ó. Corcho. Short paper: From streaming data to linked data - a case study with bike sharing systems. In *Proc. of the 5th Int'l Workshop on Semantic Sensor Networks*, SSN '12, 2012.
15. J. F. Sequeda and O. Corcho. Linked stream data: A position paper. In *Proc. of the 2nd Int'l Workshop on Semantic Sensor Networks*, SSN'09, 2009.
16. S. Stadtmüller, S. Speiser, A. Harth, and R. Studer. Data-fu: a language and an interpreter for interaction with read/write linked data. In *Proc. of the 22nd Int'l Conference on World Wide Web*, WWW '13, 2013.
17. M. Taheriyan, C. A. Knoblock, P. Szekely, and J. L. Ambite. Rapidly integrating services into the linked data cloud. In *Proc. of the 11th Int'l Semantic Web Conference*, ISWC '12, 2012.
18. M. Taheriyan, C. A. Knoblock, P. Szekely, and J. L. Ambite. A graph-based approach to learn semantic descriptions of data sources. In *Proc. of the 12th Int'l Semantic Web Conference*, ISWC '13, 2013.
19. J. Umbrich, C. Gutierrez, A. Hogan, M. Karnstedt, and J. Xavier Parreira. The ace theorem for querying the web of data. In *Proc. of the 22nd Int'l Conference on World Wide Web*, WWW '13 Companion, 2013.
20. J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Improving the recall of live linked data querying through reasoning. In *Proc. of the 6th Int'l Conference on Web Reasoning and Rule Systems*, RR 2012, 2012.