DataBearings: An Efficient Semantic Approach to Data Virtualization and Federation

Artem Katasonov
VTT Technical Research Centre of Finland
Tampere, Finland
e-mail: artem.katasonov@vtt.fi

***Abstract*—In this paper, we describe and evaluate DataBearings, which is a lightweight platform for heterogeneous data integration from various sources such as databases, Web services, and files. DataBearings is based on an efficient semantic data virtualization and federation mechanism. We demonstrate that DataBearings is as fast as non-semantic data integration solutions such as Denodo Platform, making it the first practical semantic alternative to those, given that the comparable semantic solutions such as Virtuoso and TopBraid Composer fall well behind in terms of their run-time performance. We also demonstrate that DataBearings is very lightweight, as well as provides some unique functional features allowing easier and cheaper development and maintenance of data integration systems.**

Keywords–semantic web; enterprise information integration; data virtualization; data federation; internet of things

I. INTRODUCTION

Enterprises own an ever-growing number of databases with heterogeneous data originating from different business functions or processes. Emerging Internet of Things technologies (wireless sensors, etc.) enable enterprises to collect a variety of real-time data from the physical world, pushing the number of heterogeneous datasets even further. In addition, due to globalization and the pervasiveness of the Internet, different supply chains are increasingly integrated with each other and transforming into supply networks, requiring the information systems of different enterprises to work together, with this issue being increasingly significant not only for large scale enterprises but for companies of all sizes [1]. In other words, data relevant to an enterprise operation are often found not only in in-house databases but also in external data sources, which can be the business partners' sources (usually exposed as Web services) or even Open Data sources on the Internet. In the market, there is a great need for novel applications and better capability to provide services to customers in order to differentiate and compete. As a result, enterprises are seeking possibilities to exploit ever-growing and diverse data efficiently and dynamically to provide new and better services.

Several approaches to tackling the data integration problem have been developed, including integrated packages (e.g., SAP), messaging (e.g., WS-* services), data warehouses (also known as Extract-Transform-Load, ETL), and the Enterprise Information Integration (EII) approach [2, sec.7]. The two former approaches require implementing a custom software adapter or wrapper for each constituent data source, while the two latter approaches aim at providing a generic platform which can be configured for a particular integration case without a programming effort.

The vision underlying EII is to provide tools for integrating data from multiple sources without having to first load all the data into a central warehouse [2, sec.1]. The two central problems in EII are data virtualization and data federation. The former is about accessing data without requiring knowledge of how they are formatted or where they are physically located. The latter is about retrieving data from multiple non-contiguous data sources with a single query, even if the constituent sources are heterogeneous. EII generalizes on the principles of federated databases [3], that is, it involves creating a unified data model that encompasses the schemas of participating data sources. Users or applications formulate their queries in terms of this unified model, and each query is automatically reformulated into one or more queries to the data sources.

Data virtualization provides the business benefits of reducing the integration costs by allowing leveraging existing data sources in new ways without data replication or software development expenses, enabling new applications on the intersection of existing data sources, including external ones, as well as access to live data. When considering integration with external data sources, especially when their interfaces constantly and independently evolve, virtualization may be the only approach viable. Replicating all external data into own warehouse may just not be possible, while hard-coded adapters to external sources are expensive to maintain.

In this paper, we describe and evaluate DataBearings, a lightweight data integration platform that is based on an efficient semantic data virtualization and federation mechanism. The evaluation is done comparatively to three commercial data integration products, non-semantic Denodo Platform by Denodo Technologies, and semantic Virtuoso by OpenLink Software and TopBraid Composer by TopQuadrant. This evaluation is concerned with the run time performance, memory footprint, as well as virtualization-related functional features.

Denodo is a leading tool in data virtualization. It is based around the relational data model. Both Virtuoso and TopBraid are semantic solutions that enable virtualization of non-semantic data, in principle. Both realize it via a query-time ETL, where all source data is transformed into Resource Description Framework (RDF) and loaded into a temporary RDF storage, just to be read from there in the next step that is the execution of a SPARQL query. In contrast, DataBearings realizes a more pure data virtualization approach. It does not transform the source data into RDF, but rather searches for the answer to the target semantic query directly from non-semantic source data. To the best of our knowledge, DataBearings is the only semantic data virtualization solution available at present that is not based on ETL while also being capable of working with Web data and not only relational databases.

We demonstrate that DataBearings is very fast, running as fast or even faster than the non-semantic Denodo, and much faster than the comparable semantic solutions such as Virtuoso and TopBraid. Moreover, DataBearings is very lightweight, with a significantly smaller memory footprint than other systems. Finally, in addition to providing known evolution-related benefits of the semantic technology, DataBearings enables even easier and cheaper development and maintenance of data integration systems through a set of advanced features not available in Virtuoso, TopBraid, or Denodo.

The comparative evaluation of systems, reported in this

 ISBN: 978-1-61208-420-6

paper, uses a very simple and understandable data integration case. A description of more complex and practical cases that were realized with DataBearings for the parking domain can be found in [4], [5].

The rest of the paper is structured as follows. Section II describes a simple data integration scenario that we use as a running example, as well as an evaluation case. Section III analyses the existing data virtualization approaches and example systems, both non-semantic and semantic, including how our running example is handled in these. Section IV describes the DataBearings platform, while Section V provides a comparative evaluation of DataBearings in terms of its run time performance. Finally, Section VI concludes the paper.

II. Running Example

As a running example, as well as a comparative evaluation case, we use the following simple data integration scenario that involves two data sources.

The Finnish state-owned railway monopoly, VR, publishes data on their trains via a feed at *http://188.117.35.14/TrainRSS/TrainService.svc/AllTrains ?showspeed=true*. The content is XML, with a record about a train found at XPath */rss/channel/item*. A record includes such elements as the train identifier, category, origin, destination, current location and speed. A specific complication comes from the fact that a location is given within a single XML element as a whitespace-separated string of a latitude and a longitude, e.g., *<georss:point>60.91658 26.17051</georss:point>*, instead of two separate elements for the latitude and the longitude. Henceforth, we refer to this data source as *DS1*.

The second data source, *DS2*, contains data, which we collected ourselves, on all major cities of Finland and the approximate bounding rectangles of their metro areas. The data is published in a simple comma-separated-values (CSV) format, with a row, e.g., *Tampere, 61.615563, 23.424657, 61.378988, 24.145634* (name, north, west, south, east).

The data integration task is then to extend the train records with an additional attribute containing the name of the city, in the metro area of which the train is currently located. That is, a join of two data sources is to be performed with the following condition: *ds1.lat>=ds2.south & ds1.lat<=ds2.north & ds1.lng>=ds2.west & ds1.lng<=ds2.east*.

III. Related Work

EII industry was born in late 90's and branded as a market category in 2002 [2, ch.6]. In the present, a number of big IT companies provide a data virtualization solution, with notable examples being IBM Cognos, Cisco Composite Information Server, and Denodo Platform by Denodo Technologies. Some products, e.g., IBM Cognos, only support databases but not Web services, while others, e.g., Denodo, are able to virtualize data from a variety of sources including relational and NoSQL databases, Web Services, files including CSV and MS Excel, and even some semi-structured and non-structured sources. Due to its rich feature set and the availability of an evaluation version (Denodo Express), we use in this paper Denodo Platform as a representative example of this category of products.

Denodo, as other traditional EII products mentioned above, works within the relational data model. This means that every constituent data source is represented by a relational view (a virtual relational database table) and all the following operations including data integration are performed as Structured Query Language (SQL) commands (Denodo defines an SQL extension called Virtual Query Language, VQL).

As to our running example, the *DS2* CSV data source on cities, after connecting it to Denodo, is straightforwardly represented by a virtual table view *ds2* with five columns. The *DS2* XML data source on trains is also automatically given a virtual table view *ds1* with seventeen columns, of which twelve correspond to the elements of a train record and the other five repeat for each record the values of the elements and attributes of encompassing *rss* and *channel* XML tags. As the current location of a train is given with a single whitespace-separated value, we need to define a secondary projection/selection view *p_ds1*, in which we define two new columns: 'lat' as *cast('float', substring(ds1.point, 0, instr(ds1.point, ' ')))* and 'lng' as *cast('float', substring(ds1.point, instr(ds1.point, ' ')+1))*, as well as preserve only the columns of interest. Finally, we define a join view *p_join*, with inner join conditions *p_ds1.lat>=ds2.south*, *p_ds1.lat<=ds2.north*, *p_ds1.lng>=ds2.west* and *p_ds1.lng<=ds2.east*. An execution of this view produces the result we seek.

While providing an efficient solution to our integration problem, the main disadvantage of this approach is low modifiability. This is due to the fact that the relational model always requires an explicit schema (even if it is automatically produced by Denodo) and the links between views are hard-coded to that schema via SQL constructs. In fact, in Denodo, after renaming a few output fields in *ds1*, we were just not able to fix the case without completely removing and re-doing *p_ds1* and then *p_join*.

Several authors in [2] argued for a need to exploit the benefits of the semantic technologies in EII. One reason for applying semantics to the data integration problem is a 'softer' nature of links in semantic models, as links and entities can be added or removed without breaking the rest of the model. This enables an agile and interactive evolution of data integration and integrated data analytics cases, with a faster return on investment [6].

In [2, ch.6], it was stated that none of the existing at the time EII tools used formal semantics, but predicted that EII will adopt the foundational technologies of the Semantic Web. Efforts towards semantic EII were reviewed later in [7], referencing, however, only a handful of research projects. Even at the time of writing this paper, to the best of our knowledge, the only available practical semantic data virtualization solutions are those that only support working with relational databases as virtual RDF graphs and cannot be used for access to Web data, such as D2RQ [8]. All solutions that support a variety of data source types rely on ETL instead, that is extraction of non-semantic data from their original data sources, explicit transformation of those data into RDF, and loading it into an RDF data warehouse. Notable commercial products include Data Unleashed Federator by Blue Slate Solutions, Virtuoso by OpenLink, and TopBraid Composer by TopQuadrant, with Linked Stream Middleware [9] and the ontology-based mediator in [10] deserving a mention on the research side.

We use in this paper Virtuoso and TopBraid as representative examples of the state-of-art in semantic data integration of heterogeneous data. Both come the closest to being data virtualization products as they support query-time ETL. That is, after a SPARQL query is received, they access relevant data sources, transform received data into RDF, load RDF into an in-memory RDF storage, and then execute the query on that storage. The repetition of the ETL step is avoided for static sources that did not change since the last query.

Let us use TopBraid to explain the specifics. As to our running example, the *DS2* CSV data source is mapped to

 ISBN: 978-1-61208-420-6

RDF via SemTables, which is TopBraid's own simple ontology consisting of three properties: sheetIndex, rowIndex, and columnIndex. With this approach, mapping data onto an arbitrary semantic structure is not supported, but only onto the most straightforward one: each data sheet corresponds to a class, every row to an instance of that class, and every column to a property of that class. Each data row in *DS2* is, thus, transformed into RDF (Turtle notation) as follows: *[a c:City] c:name "Tampere"; c:north 61.615563; c:west 23.424657; c:south 61.378988; c:east 24.145634*. *DS1* XML data source is mapped to RDF via SXML, which is also TopBraid's own ontology for describing the structure of XML documents. The resulting semantic structure is rather complex and not flexible, with train attributes represented as classes rather than properties, as follows (only the id and the location attributes included): *[] a vr:item; composite:child [a vr:guid; composite:child [sxml:text "IC10"]]; composite:child [a vr:point; composite:child [sxml:text "60.37992 25.09723"]]*.

After both data sources are mapped to RDF, the target integration case is realized via the multi-graph SPARQL query in Figure 1 (simplified here by selecting only the id and the location of a train plus omitting the full URIs of the two graphs). Submitting this query to the TopBraid's SPARQL endpoint produces the result we seek.

```
SELECT *
WHERE {
  GRAPH <ds1> {
    [ ] a vr:item;
      composite:child [a vr:guid; composite:child [sxml:text ?id]];
      composite:child [a vr:point; composite:child [sxml:text ?loc]].
  }.
  BIND (xsd:decimal(strbefore(?loc,' ')) as ?lat) .
  BIND (xsd:decimal(strafter(?loc,' ')) as ?lng).
  GRAPH <ds2> {
    [ ] c:name ?name; c:north ?n; c:south ?s; c:west ?w; c:east ?e
  }.
  BIND (xsd:decimal(?n) as ?no). BIND (xsd:decimal(?s) as ?so).
  BIND (xsd:decimal(?w) as ?we). BIND (xsd:decimal(?e) as ?ea).
  FILTER (?lat<=?no && ?lat>=?so && ?lng>=?we && ?lng<=?ea).
}
```

Figure 1. SPARQL query for the running example.

The main disadvantages of this approach are low performance and scalability (due to the nature of ETL) and a need to mix all processing and integration steps in a single SPARQL query. Note the explicit instructions for splitting the location into the latitude and the longitude, which are now part of the final query, while in the case of a relational tool like Denodo hidden into an intermediary projection view. For data statically residing in the TopBraid RDF storage, one could implement this step with inference rules, but this option is not available for data loaded on-demand from external non-semantic sources. The need to explicitly address the graphs corresponding to each data source is also a disadvantage as it does not allow protecting a user from data distribution details.

IV. DataBearings Approach

To the best of our knowledge, DataBearings is the only available at present semantic solution for data integration which is not based on an explicit extract-transform-load of data as RDF while also being capable of working with Web data and not only relational databases. Another distinctive feature is that, instead of restricting itself to the capabilities of standard semantic technologies such as RDF and SPARQL, DataBearings uses a more expressive and powerful data model, Tim Berners-Lee's Notation3 (N3) [11]. N3 can be easiest explained as RDF with nesting. Each N3 statement is necessarily an RDF triple, but the subject and/or object of it are allowed to be nested N3 models containing other statements (see Figure 3 for an example of N3 data). An important convention is that only statements at the top level are treated as facts, while statements in a nested model are considered only in the context set by the containing statement.

DataBearings features a fast in-memory N3 data storage, which can contain RDF or N3 data, data source annotations, as well as production rules and other imperative constructs in N3-based Semantic Agent Programming Language (S-APL). S-APL is developed around the central idea of N3Logic [12], that is to have N3 as a single data model for all of data, queries, and rules. However, S-APL drops the monotonicity assumption of N3Logic and, similarly to SPARQL, includes constructs for negation, solutions aggregation, as well as for facts removal. S-APL was introduced in [13] and later formalized in [14].

An overview of the DataBearings' semantic data virtualization and federation approach is given in Figure 2. In [5], we provided a detailed description of how this approach is realized exploiting the capabilities of S-APL. In this paper, we focus rather on the practical aspects and benefits of using DataBearings for implementing data integration cases.

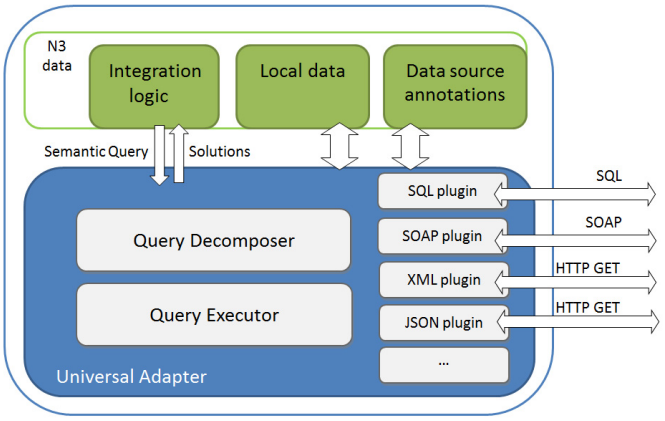

Figure 2. Semantic data virtualization in DataBearings.

Not unlike other data virtualization systems, DataBearings features a central component, we refer to as *Universal Adapter*, with dynamically loaded adapter plugins for different types of data sources. DataBearings currently comes with plugins for SQL databases, SOAP Web services, XML/JSON/CSV Web services or local files, as well as MS Excel files, and provides an API for developing additional plugins. A plugin is instantiated using an explicit N3-based *data source annotation*, an example of which, for *DS1* from our running example, is given in Figure 3. Such an annotation specifies the class of the plugin (*o:type*), class-specific connection parameters (*o:service*), class-specific data syntax (*d:tree* in *o:semantics*), data mapping to an ontology (the rest of *o:semantics*), an applicability-to-query pattern (*o:getPattern*) and, optionally, an applicability precondition (*o:precondition*). Instantiated adapter plugins that act as ontological virtualizations of data sources we refer to as *ontonuts*, a concept we first introduced in [15].

The data source annotation for *DS2* is done in a similar fashion. It uses sapl.shared.eii.CsvOntonut plugin, describes the syntax via d:table, d:row, and d:value properties and maps data to the following semantic structure: *[a ex:City] ex:hasName ?name; ex:hasBounds [ex:north ?n; ex:south ?s; ex:west ?w; ex:east ?e]*.

This data source mapping approach alone offers a number of advantages over the rigidness of mapping in existing

 ISBN: 978-1-61208-420-6

```
ex:VR_Ontonut a o:Ontonut
  ; o:type "sapl.shared.eii.XmlOntonut"
  ; o:service [ o:uri "http://%%ip%%/TrainRSS/..." ]
  ; o:precondition { ex:VR ex:hasIP ?ip }
  ; o:semantics {
    {
      * d:tree { * d:row {
        [ d:element "rss"] d:branch {
          [ d:element "channel" ] d:branch {
            [ d:element "item" ] d:branch {
              [ d:element "guid"] d:branch {* d:value ?id}.
              [ d:element "point"] d:branch {* d:value ?loc}.
        } } } }.
        ?lat s:expression "substring(?loc,0,indexOf(?loc,' '))".
        ?lng s:expression "substring(?loc,indexOf(?loc,' ')+1)".
    } => {
      [a ex:Train] ex:hasID ?id; ex:hasLocation [ex:lat ?lat; ex:lng ?lng]
    }
  }
  ; o:getPattern { * a ex:Train } .
```

Figure 3. A data source annotation for the running example.

products like TopBraid, Virtuoso, or Denodo (see Section III):

- The explicit variable-based mapping allows data to be mapped to arbitrary semantic structures as dictated by target ontologies. This is in contrast to being forced to deal with over-simplified (for *DS2*) and over-complex (for *DS1*) temporary semantic structures, created just for the integration job, in TopBraid or Virtuoso.
- Transformations like splitting a location into latitude and longitude can be handled already at the data source mapping level. Note the two *s:expression* operations in Figure 3. This, again, allows mapping data to existing ontologies, as well as in contrast to having to carry these operations into the final SPARQL query in TopBraid or handling them in an intermediary projection view in relational systems like Denodo.
- Data source requests can be parametrized with values obtained from local data (via a precondition as in Figure 3 or via a local starter query, see below). This is in contrast to always having to specify the URIs statically in all of TopBraid, Virtuoso, and Denodo, even while most practical cases require parametrization.

After providing the data source annotations, the S-APL production rule that obtains the result we seek in our running example is as in Figure 4. As can be seen from Figure 4, an additional advantage of DataBearings is that, unlike in TopBraid, a data user is completely insulated from the data distribution details. A person or system specifying a query or a production rule as above does not need to know whether all needed data is found from a single data source or is distributed among two sources. If the situation changes in this regard (data sources are combined or split), only the ontonuts' definitions have to be updated while the queries and rules are not affected.

```
{
  [a ex:Train] ex:hasID ?id; ex:hasLocation [ex:lat ?lat; ex:lng ?lng].
  [a ex:City] ex:hasName ?name; ex:hasBounds
    [ ex:north ?n; ex:south ?s; ex:west ?w; ex:east ?e ].
  ?lat <= ?n. ?lat >= ?s. ?lng >= ?w. ?lat <= ?e.
} => { ... }
```

Figure 4. S-APL query for the running example.

The Query Decomposer part of the Universal Adapter analyses a query found in the head of a production rule against data source annotations, pre-selected using their applicability-to-query patterns (*o:getPattern*). Based on this analysis, all query statements are split into the following groups:

1) Statements covered by one or more ontonuts.
2) Statements which are covered by an ontonut, but could not be handled by that ontonut (e.g., a filter on a property value can always be handled by an SQL source, but by a Web service only if its request interface includes a corresponding parameter).
3) Inter-ontonut join conditions.
4) Local starters: statements not matching any ontonut's semantics that will be run as a query against the local data at the beginning of the execution. The obtained solutions may be used by ontonuts for parametrization.
5) Local join conditions: explicit conditions for performing join of ontonuts-produced solutions with solutions obtained at the beginning via local starters.
6) Local finalizers: statements that will be handled at the end of the execution. These are either solution aggregators (count/min/max/sum) or selection statements (from local data) that depend on variable values produced in ontonuts.

The Query Executor part of the Universal Adapter performs a query evaluation process, in which all the involved statements are handled in the following order: (1) local starters, (2) ontonuts' preconditions, (3) covered statements combined and translated into ontonut-specific form, e.g., SQL, SOAP, HTTP GET, (4) ontonuts' post-processing operations (e.g., splitting a location into latitude and longitude), (5) covered but not handled statements, (6) join conditions (also implicit join by a common variable value is supported, as well as the union operation), (7) local join conditions (again, including implicit), (8) local finalizers. Note that, in this process, at no point external non-semantic data is transformed into RDF. Rather, a semantic query is answered on the combination of external non-semantic and local semantic data. The step 3 is based on a relevant sub-query transformation, not data transformation. This step outputs directly a set of solutions, i.e., a list of variable-value mappings. For SQL sources, this step actually involves translating an S-APL sub-query into SQL. For simple Web services like in our running example, this step involves matching the sub-query with the data source syntax, requesting data, and then picking up relevant values from data and assigning them to variables as needed.

The simple example in Figure 4 does not include any local or not-handled statements, but only ontonut-covered selection statements (group 1) and inter-ontonut join statements (group 3). However, local statements appear in most practical integration cases. Based on a specific question in hand, or the current situation, some initial solutions (via local starters) may need to be obtained and used to determine (via preconditions) what exactly external data sources have to be contacted. Solutions from external sources may need to be filtered locally if the corresponding source cannot do it (covered but not handled statements). Finally, after solutions from external sources are obtained and joined or unionized, one may still want to extend them with extra attributes from the local data or to group solutions by a value and, e.g., count. An ability to flexibly combine virtualized and local data, and, thus, handle these practical cases, is a powerful feature of DataBearings giving it an advantage over most other data integration solutions, both semantic and non-semantic.

V. EVALUATION

A. Integration run-time

In this section, we report on an evaluation of the performance of DataBearings, in terms of the integration run-time, in

comparison to related commercial integration products, namely non-semantic Denodo 5.5 and semantic Virtuoso 7.2, and TopBraid 4.6 (see Section III).

For this evaluation, we created files with snapshots of data from *DS1* and *DS2* data sources of our running scenario (see Section II). The *DS1* snapshot contains 75 train records, while *DS2* snapshot contains 25 city records. This gives 1875 join pairings to examine and results in 27 solutions (trains currently in a city area). Then, we created copies of the *DS1* snapshot and, by copy-paste of existing data, increased the number of records in each by a factor from 2 to 200. The largest file thus contains 15000 train records encoded in 4.5 megabytes of XML content, and results in 375000 join pairings and 5400 solutions. The same *DS2* snapshot was used in all cases.

Denodo run-times are obtained from its execution trace view. For TopBraid, the integration SPARQL query was used as a SPIN framework inference rule, because the execution times of such rules are reported in TopBraid's SPIN statistics view. To check whether SPIN results in a significant additional overhead, we were also submitting the query directly via the TopBraid's SPARQL endpoint and observed the response time (time to first byte) in Chrome browser's development tools. To avoid an overhead created by formatting a large response, the SPARQL query was modified to return only the number of results instead of the results themselves. We did not observe any significant deviation between such a response time and the corresponding SPIN statistics number for any of the input file sizes, and report the SPIN statistics numbers here. Virtuoso does not seem to have performance self-reporting, so we measured its SPARQL endpoint's response times, same way as described above for TopBraid.

Both TopBraid and Virtuoso use query-time ETL, that is, when receiving a SPARQL query, they access and transform input data into RDF, load it into temporary RDF storage, and then run the query. If the files, however, did not change since the last query, the ETL step is skipped. Therefore, we recorded separately also the run-time of TopBraid and Virtuoso on previously loaded data, and report these numbers as 'TopBraid (QL)' and 'Virtuoso (QL)'. This case only involves executing a SPARQL query on the RDF storage, but requires extracting all trains and all cities and then doing the cross-join.

All experiments were performed on the same Windows 7 PC with 2.3 GHz CPU and 4GB RAM. All the run-time numbers reported are averages over 10 execution runs. Table I and Figure 5 provide the results. The columns correspond to different replication factors of the *DS1* snapshot. Run-times are reported in milliseconds.

TABLE I. INTEGRATION TIME FOR THE RUNNING SCENARIO

	1	10	20	50	100	200
DataBearings	14	42	61	140	265	552
Denodo	41	66	78	178	345	555
TopBraid	124	1322	3892	21212	79284	305832
TopBraid (QL)	78	537	1042	2631	5277	10554
Virtuoso	340	1842	2999	7359	16559	35152
Virtuoso (QL)	25	56	80	156	344	708

As can be seen, DataBearings consistently demonstrates a performance very similar to that of non-semantic Denodo, with a roughly-linear increase in the run-time with the data size growth. In fact, DataBearings even outperformed Denodo in all our test cases, with a bigger gain for smaller data sizes, up to three times for the smallest (original) data. Note that this is achieved even given all the overhead in DataBearings created by flexible query decomposition and match-making to the data sources.

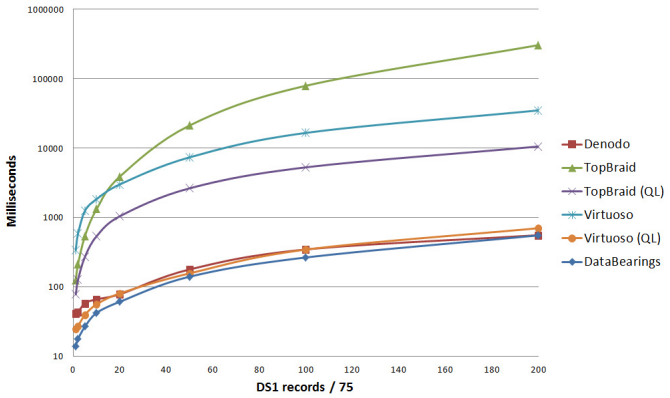

Figure 5. Integration time for the running scenario.

The biggest of the test jobs (15000 train records) is handled by both Denodo and DataBearings in just over 0.5 seconds. On the other hand, Virtuoso needs 35 seconds to do the same job, while TopBraid is out of hand with 5.1 minutes. Note the logarithmic scale in the figure. Considering SPARQL performance on already pre-loaded data, TopBraid still needs 10 seconds with the largest data set, which is surprisingly poor. Virtuoso, however, needs just 0.7 seconds, which is comparable to the total performance of DataBearings and Denodo, but can work only in a static data case.

B. *Memory footprint*

In addition to measuring the run-time performance, we also measured the memory footprints of DataBearings, Denodo, and TopBraid when executing our running example. These three systems are Java-based, which gives us a possibility to precisely measure their footprints. Virtuoso was excluded from this comparison as it is a native Windows application. We can only say that its total memory footprint, as can be observed in the Windows resource monitor, was always rather significant during the tests, with values in the range of 200 to 800 MB.

The total memory footprint includes permanent generation memory (the program code), which is constant regardless of the handled data size, as well as allocated memory (the heap) which grows with the handled data size. Table II and Figure 6 provide the results, in MB. The first column in the Table II shows the permanent generation footprint alone, while the rest of columns are sums of permanent generation and allocated footprints. As before, all the numbers reported are averages over 10 execution runs.

TABLE II. MEMORY FOOTPRINT FOR THE RUNNING SCENARIO

	perm	1	10	20	50	100	200
DataBearings	10	17	26	32	48	84	118
DataBearings (GC)	10	15	20	21	41	60	89
Denodo	68	120	150	183	280	267	280
TopBraid	72	241	648	602	622	674	738

The permanent generation footprint of DataBearings was measured using Java VisualVM tool, which is a part of the Java Development Kit. The allocated memory footprint was measured via calling java.lang.Runtime's *totalMemory()-freeMemory()* from code, for a better precision. Such a reading was performed in multiple points of the code and the maximum value was taken. The memory footprints (both permanent generation and allocated) of Denodo and TopBraid were measured using Java VisualVM. Note that we left the maximum memory settings of Denodo and TopBraid as default in these systems,

 ISBN: 978-1-61208-420-6

which affects the point at which the Java automatic garbage collection starts (visible in the figure).

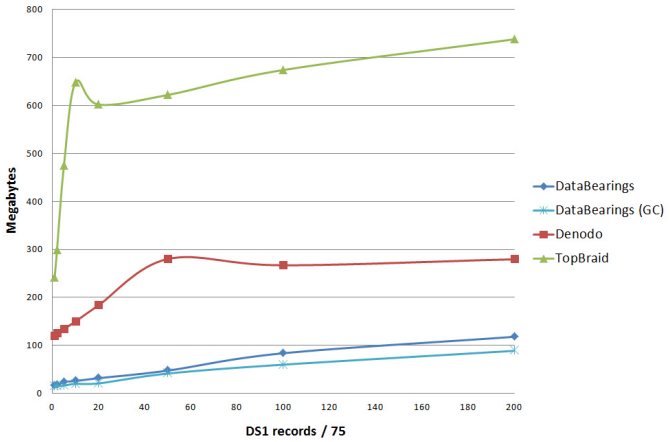

Figure 6. Memory footprint for the running scenario.

As can be seen in Table II, the DataBearings' code (its permanent generation footprint) is seven times lighter than that of Denodo or TopBraid. Also, the total memory footprint of DataBearings was significantly lower than the total footprints of Denodo and TopBraid, in all our experiments.

To further demonstrate the light weight of DataBearings, we used it as a library in an Android application and made a mobile phone to execute the integration job from our running example. The experiments were conducted on a Nexus 5 phone. Table III presents the results, while comparing them to the numbers obtained on a PC (as in Table I).

TABLE III. INTEGRATION TIME ON ANDROID PHONE

	1	2	5	10	20	50	100	200
PC	14	18	27	42	61	140	265	552
Nexus 5	72	122	283	501	1079	2605	5313	10625

Obviously, data integration jobs are better left to be performed on servers. Yet, using DataBearings, small data volumes can be integrated even locally within a smartphone application. So, 1500 trains (factor of 20) x 25 cities are handled in just over a second, which appears to be still an acceptable time for a user to wait.

VI. Conclusions

Supported by the comparative evaluation presented in this paper, we claim that, to the best of our knowledge, DataBearings is (1) the only semantic data virtualization solution available at present, which is not emulating virtualization via query-time ETL while capable of working with Web data and not only relational databases, (2) the only semantic solution to the data integration problem that is as fast as non-semantic ones, and (3) the only data integration solution, semantic or not, that is so lightweight that it can be run on a smartphone.

Being a semantic solution, DataBearings offers a possibility to exploit the evolution-related benefits of the semantic technology. In addition, DataBearings enables even easier and cheaper development and maintenance of data integration systems through a set of advanced features not available in other semantic systems. These include the ability to map data to arbitrary semantic structures as dictated by target ontologies, the ability to perform source data transformations prior to mapping to an ontology, the ability to parametrize data source requests, as well as the ability to flexibly combine virtualized and local data.

In this paper, we did not touch some other advanced features of DataBearings that go well beyond capabilities of existing data integration systems. These include a support for federated data updates (i.e., write not only read), as well as for abstraction of virtualized data. A discussion of these DataBearings' features can be found in [5].

An interesting result in our experiments was that Virtuoso and TopBraid performed worse than DataBearings even when running the job on already transformed into RDF and pre-loaded data. This indicates that, even when dealing with static data, when it would be possible to transform all data into semantic form and store as RDF, it may still be not a good idea if integration (join) operations cannot be also performed statically, but have to be done upon a request. Thus, even in such cases, pure virtualization, as in DataBearings, may be the most efficient and thus recommended choice.

Acknowledgements

This work is performed in the DIGILE Internet of Things project, which is supported by TEKES.

References

[1] L. D. Xu, "Enterprise systems: State-of-the-art and future trends," IEEE Trans. Industrial Informatics, vol. 7, no. 4, 2011, pp. 630–640.

[2] A. Y. Halevy, N. Ashish, D. Bitton, M. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka, "Enterprise information integration: Successes, challenges and controversies," in Proc. ACM SIGMOD International Conference on Management of Data. ACM, 2005, pp. 778–787.

[3] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," ACM Computing Surveys, vol. 22, no. 3, 1990, pp. 183–236.

[4] A. Lattunen, A. Katasonov, and T. Koivuniemi, "Flexible parking data management across enterprise and beyond," in Proc. ITS World Congress, Detroit, 2014.

[5] A. Katasonov and A. Lattunen, "A semantic approach to enterprise information integration," in Proc. 8th IEEE Conf. Semantic Computing, 2014, pp. 219–226.

[6] Introducing Data Unleashed: An Overview of Data Federation Agility, Blue Slate Solutions, 2014, online: http://www.blueslate.net/Dave/DataUnleashedIntroduction_OverviewOfDataFederationAgility/ [retrieved: June, 2015].

[7] J. Zhou, H. Yang, M. Wang, R. Zhang, T. Yue, S. Zhang, and R. Mo, "A survey of semantic enterprise information integration," in Proc. Intl. Conf. Information Sciences and Interaction Sciences (ICIS). IEEE, 2010, pp. 234–239.

[8] C. Bizer and A. Seaborne, "D2RQ – treating non-RDF databases as virtual RDF graphs," in Proc. 3rd International Semantic Web Conference, 2004.

[9] D. L. Phuoc, H. Q. Nguyen-Mau, J. X. Parreira, and M. Hauswirth, "A middleware framework for scalable management of linked streams," J. Web Semantics, vol. 16, 2012, pp. 42–51.

[10] K. Hribernik, C. Hans, C. Kramer, and K.-D. Thoben, "A service-oriented, semantic approach to data integration for an internet of things supporting autonomous cooperating logistics processes," in Architecting the Internet of Things, D. Uckelmann, M. Harrison, and F. Michahelles, Eds. Springer, 2011, pp. 131–158.

[11] T. Berners-Lee, Notation 3: An RDF language for the Semantic Web, online: http://www.w3.org/DesignIssues/Notation3.html [retrieved: June, 2015].

[12] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler, "N3logic: A logical framework for the world wide web," Theory Pract. Log. Program., vol. 8, no. 3, 2008, pp. 249–269.

[13] A. Katasonov and V. Terziyan, "Semantic agent programming language (S-APL): A middleware platform for the semantic web," in Proc. 2nd IEEE Conf. Semantic Computing, 2008, pp. 504–511.

[14] M. Cochez, "Semantic agent programming language: Use and formalization," Master's thesis, University of Jyväskylä, 2012.

[15] S. Nikitin, A. Katasonov, and V. Terziyan, "Ontonuts: Reusable semantic components for multi-agent systems," in Proc. 5th Intl. Conference on Autonomic and Autonomous Systems. IEEE, 2009, pp. 200–207.

 ISBN: 978-1-61208-420-6