Alrehamy and Walker *J Big Data* (2018) 5:14
https://doi.org/10.1186/s40537-018-0123-x

Journal of Big Data

RESEARCH Open Access

SemLinker: automating big data integration for casual users

Hassan Alrehamy[1,2] and Coral Walker[1*]

*Correspondence:
Huangy@cardiff.ac.uk
[1] School of Computer Science and Informatics, Cardiff University, Queens Building, Cardiff, UK
Full list of author information is available at the end of the article

Abstract

A data integration approach combines data from different sources and builds a unified view for the users. Big data integration inherently is a complex task, and the existing approaches are either potentially limited or invariably rely on manual inputs and interposition from experts or skilled users. SemLinker, an ontology-based data integration system, is part of a metadata management framework for personal data lake (PDL), a personal store-everything architecture. PDL is for casual and unskilled users, therefore SemLinker adopts an automated data integration workflow to minimize manual input requirements. To support the flat architecture of a lake, SemLinker builds and maintains a schema metadata level without involving any physical transformation of data during integration, preserving the data in their native formats while, at the same time, allowing them to be queried and analyzed. Scalability, heterogeneity, and schema evolution are big data integration challenges that are addressed by SemLinker. Large and real-world datasets of substantial heterogeneities are used in evaluating SemLinker. The results demonstrate and confirm the integration efficiency and robustness of SemLinker, especially regarding its capability in the automatic handling of data heterogeneities and schema evolutions.

Keywords: Data integration, Big data, Data lake, Modeling, Schema evolution, Schema mapping, Metadata management

Introduction

Big data is growing rapidly from an increasing plurality of sources, ranging from machine-generated content such as purchase transactions and sensor streams, to human-generated content such as social media and product reviews. Although much of these data are accessible online, their integration is inherently a complex task, and, in most cases, is not performed fully automatically but through manual interactions [1, 2]. Typically, data must go through a process called ETL (Extract, Transform, Load) [3] where they are extracted from their sources, cleaned, transformed, and mapped to a common data model before they are loaded into a central repository, integrated with other data, and made available for analysis. Recently the concept of a data lake [4], a flat repository framework that holds a vast amount of raw data in their native formats including structured, semi-structured, and unstructured data, has emerged in the data management field. Compared with the monolithic view of a *single* data model emphasized by the ETL process, a data lake is a more dynamic

environment that relaxes data capturing constraints and defers data modeling and integration requirements to a later stage in the data lifecycle, resulting in an almost unlimited potential for ingesting and storing various types of data despite their sources and frequently changing schemas, which are often not known in advance [5]. In one of our earlier papers [6], we propose personal data lake (PDL), an exemplar of this flexible and agile storage solution. PDL ingests raw personal data scattered across a multitude of remote data sources and stores them in a unified repository regardless of their formats and structures. Although a data lake like PDL, to some extent, contributes towards solving the big data *variety* challenge, data integration remains an open problem. PDL allows its users to ingest raw data instances directly from the data sources, but the data extraction and integration workflow, without predefined schemas or machine-readable semantics to describe the data, is not straightforward. Often the user has to study the documentation of each data source to enable suitable integration [7]. An enterprise data lake system built with Hadoop [8] would rely on professionals and experts playing active roles in the data integration workflow. PDL, however, is designed for ordinary people, and has no highly trained and skilled IT personnel to physically manage its contents. To this end, equipping PDL with an efficient and easy-to-use data integration solution is essential for casual users and allows them to process, query, and analyze their data, and to gain insights for supporting their decision-making [9].

To support PDL, the big data integration (BDI) system faces the following three challenges:

- The scalability challenge arises from the vast number of data sources that may input to a data lake [10], and the continuous addition of new and varying data sources [2].
- The heterogeneity challenge is the implication of dealing with various types of raw data collected from a large number of unrelated data sources [1]. The data sources of a lake, even in the same domain, can be very heterogeneous regarding how their data are structured, labeled and described (e.g., naming conventions for JSON keys, XML tags, or CSV headers), exhibiting considerable variety even for data with substantially similar attributes [11]. The reconciliation of semantic and structural heterogeneities in raw data is a necessary preparatory step for storing and retrieving data quickly and cost efficiently and aligning raw data from different sources so that all types of data relevant to a single analysis requirement can be combined and analyzed together. Manually handling heterogeneity reconciliations would pose a huge burden on PDL users. Despite efforts in the fields of semantic web and data integration for automating the reconciliation process [12–14], existing approaches, most of which require optimized parameter tuning and expertise-based configurations to cope with the heterogeneities of data [10], cannot be adopted in PDL.
- The schema evolution challenge refers to the problem of handling unexpected changes in the schema and structure of the data ingested [15, 16]. Big data is often subject to frequent schema evolutions, which would cause query executions to crash if not dealt with [17]. Handling schema evolutions is a non-trivial task, and

the common practice normally involves employing skilled manpower. Schema evolution has been a known problem in the database community for the last three decades [18] and has become frequent and extensive in the era of big data, yet it has not been addressed effectively [1, 7, 11].

In this paper, we introduce *SemLinker*, an ontology-based BDI system to address the BDI challenges discussed above. SemLinker, as a principal integrated component in the PDL architecture, adopts an automatic approach that only operates on the schema metadata level without involving physical transformation of data during integration. Thus, it preserves the data in their native formats and structures while, at the same time, allowing the data to be easily analyzed and queried by casual users. In addition, SemLinker also handles frequent schema evolutions automatically and shields analysis processes operating on the schema metadata level from crashing due to unexpected schema changes.

The remainder of this paper is organized as follows: a summary of related work is given in "Related work" section; an overview of the SemLinker architecture is given in "SemLinker architecture overview" section; the implementation of SemLinker is discussed in detail in "Global schema layer", "Local schemata layer", and "Query" sections; the integration of SemLinker into the PDL architecture is described in "Integration with PDL" section; an experimental evaluation is given, and its results are presented and analysed in "Experimental evaluation" section; finally, our conclusions and future work directions are discussed in "Conclusion and future work" section.

Related work

Lenzerini introduces in [19] a theoretical framework for integrating a set of heterogeneous data sources based on their associated schema metadata, more formally, local schemas. The framework's integration workflow is to maintain a mediated schema (i.e., global ontology) and specify relationships, or *mappings*, between the mediated schema and the local schemas of different data sources under integration. The concept of an ontology [20] is used as an efficient description tool for representing the mediated schema and for providing unified views over the data collected from the integrated sources. The user formulates queries by utilizing the terms (concepts) defined by the ontology, and the queries are executed according to mappings between the ontological terms and their corresponding representations in the local schemas. Many current state-of-the-art ontology-based data integration systems follow Lenzerini's framework [19] to integrate structured and/or semi-structured data collected from heterogeneous data sources, such as [7, 21–23]. Although these systems can deliver effective and efficient data integration performance in many use cases, they typically require continuous human intervention to supervise the process of discovering mappings between the global ontology and the local schemas [1, 14, 24], which is a laborious and time-consuming task itself that requires schema matching expertise. Furthermore, these systems favor static integration workflows, where any changes in the global ontology or the local schema metadata implies a high degree of manual processing to (re)configure the mappings [7, 14].

With the increasing popularity of data lakes in the big data landscape, metadata is becoming of immense importance for BDI research [25], and metadata management is currently an active research topic. GEMMS [5] is a Metadata Management

Framework (MMF) that extracts and manages metadata about the data stored in the constance data lake. GEMMS integrates the user's personal data in life science fields by modeling them with a common metadata model. Although GEMMS is theoretically capable of reconciling semantic heterogeneities between the raw data, and tolerating big data volume and variety, its architecture suffers multiple drawbacks: first, it reconciles structural heterogeneities through physical data transformations, which implies altering the native schemas and structures of the data and posing constraints on the ingestion process; secondly, it is very sensitive to emerging changes in the raw data schemas; thirdly, the GEMMS literature does not describe how the integrated data can be systematically accessed and queried. Kayak [26], a generic framework for managing data lake content through metadata-based data preparation and wrangling, is a case similar to GEMMS. Although it promises integration and querying capabilities, its approach has not yet been revealed. Atlas [27] is an agile Apache enterprise framework for data governance and metadata management in Hadoop data lakes. After integration with Apache Avro [28], it became capable of handling schema evolutions in the datasets stored in a Hadoop data lake. In [7], Nadal et al. propose an integration-oriented ontology-based system for integrating heterogeneous JSON data in data lakes, and for governing their schema evolutions. The system has two ontology levels. The top level is modeled as an OWL ontology to offer unified views over local schemas, and the bottom level contains local schemas maintained as RDF graphs, A data steward is responsible to provide mappings between the two levels. If a particular local schema evolves, the data steward is notified, and a remapping then takes place. The shortcoming of current data lake BDI solutions is that they inherently exhibit the same drawbacks found in traditional data integration. For instance, raw data metamodeling remains an expensive task and requires expert user supervision [1, 29]. Furthermore, the schema evolution problem and its impact on data access, processing, integration, and analysis in a data lake is often overlooked, and its solution largely remains manual [7, 28, 30]. Rahm states in [1] that most big data integration proposals are limited to a few data sources, and analytics over a large volume of heterogeneous data ingested from various data sources is only possible with the availability of a *holistic* data integration solution that: (i) should be fully automatic or require only minimal manual interaction, and (ii) should make it easy to add and use additional data sources and automatically deal with frequent changes in these sources.

SemLinker, as a data integration system, shares many features and functionalities with other solutions. However, as a solution for PDL whose users are typically casual and unskilled, SemLinker needs to be in agreement with Rahm's automation proposal and isolate its users from the technical details imposed by the integration workflow. Our proposed automations in SemLinker are implemented in the following operations:

- Metadata extraction and maintenance.
- Schema evolution handling.
- Discovery of mappings between the system's global ontology and the metadata denoting the local schema of a data source.

SemLinker also supports data analysis in PDL by allowing direct queries over its metadata repository. Thus, big data management tasks such as data summarization, analytics, and insight discovery can be readily performed.

SemLinker architecture overview

The SemLinker architecture consists of a global schema layer, a local schemata layer, and the relationships between these layers. The global schema layer consists of the global schema ($\mathcal{G}$), and the query engine for formulating queries over $\mathcal{G}$. The local schemata layer consists of the schemas repository (S), and schema metadata extraction, mapping and management components. As an ontology-based integration system, SemLinker is conceptually based on the theoretical framework proposed by Lenzerini [19], and we formally define the system as follows:

Definition 1 SemLinker is a triple $\langle \mathcal{G}, S, \mathcal{M} \rangle$, where $\mathcal{G}$ is the global schema, S is a set of local schemas corresponding to n data sources, $S = \{S_1, S_2, \ldots, S_n\}$, and $\mathcal{M}$ is a set of mapping assertions, such that, for each S_i there is a set of mappings between g and S_i, $g \in \mathcal{G}, 1 \leq i \leq n$, in the form: p $\rightarrow$ a, where attribute $a \in S_i$ and property $p \in g$.

The system's global schema $\mathcal{G}$ is modeled as a global ontology and is described using web ontology language (OWL) [31]. SemLinker extracts and maintains machine-readable metadata describing the physical schema details of each data source connected with the PDL, and specific semantics about its available data. We refer to such metadata as a local schema. A local schema is described using resource description framework (RDF) [32] and is stored in the schema's repository S. SemLinker is responsible for automatically mapping the local schema S_i of the data source i to a semantically corresponding concept in the global ontology $\mathcal{G}$. Such mappings provide a metadata model that allows SemLinker to systematically annotate data ingested and allows the user to pose queries over $\mathcal{G}$ which serves as an abstraction layer over S and its raw data. With the metamodeling in place, the raw data of PDL are integrated on the metadata level; no manual effort

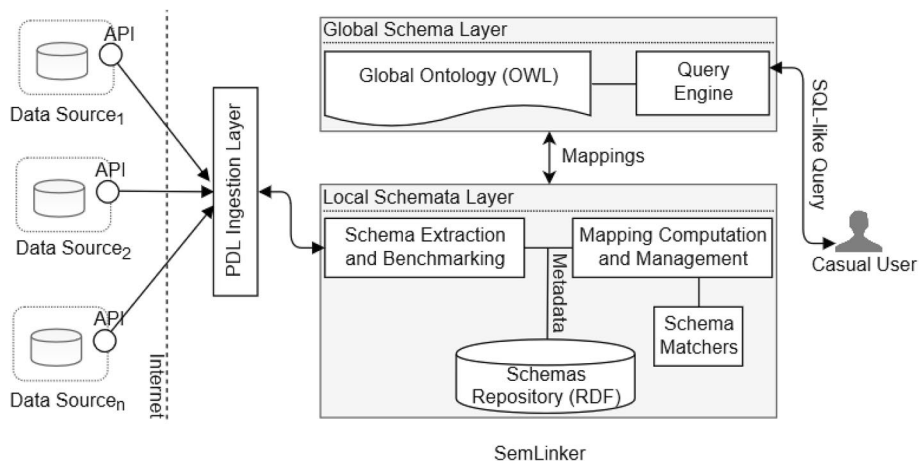

Fig. 1 Overview of SemLinker architecture

Fig. 2 Examples of personal data instances ingested from two data sources. **a** Facebook data in schema v.2.9. **b** Facebook data in schema v.3.0. **c** Twitter data in schema v.1.1. **d** Twitter data in schema v.1.2

is required to reconcile the heterogeneities in the physical schemas and structures of the raw data. Figure 1 gives a high-level overview of the SemLinker architecture.

Here we introduce a personal data example comparable to a real-life scenario to give a realistic view of the challenges that a BDI system like SemLinker must meet. Figure 2 lists four personal data instances representing social media feeds posted by a PDL user on Facebook and Twitter and ingested by the PDL through the available API of each source (Facebook Graph API, and Twitter Streaming API) with evolved schemas. Although these instances exist in self-describing formats and contain abstract schema metadata implicitly encoded in JSON keys and XML tags, they suffer semantic and structural heterogeneities, even for the instances ingested from the same data source. For example, the JSON keys in Facebook data instances (Fig. 2a, b) are expressed with different strings and exist in different structures (see "*location*" and "*geo*" keys). Similarly, Twitter data (Fig. 2c, d) also exist in different data formats. The example serves as a reference point for later sections on SemLinker's implementation.

Global schema layer

The global ontology $\mathcal{G}$ serves two purposes: to tag data sources with *type* semantic information, and to form an indispensable basis in the form of query-able format-agnostic unified views, that allows executing uniform queries over the raw data ingested from different data sources. An ideal global ontology is a comprehensive and standardized ontology that provides semantic coverage and interoperability across a vast range of domains [33]. For this reason, we initiate $\mathcal{G}$ as an OWL implementation of SCHEMA [15]. SCHEMA is a lightweight and well-curated vocabulary which consists of abstract concepts common across many domains and is used as a backbone schema for annotation in many large-scale knowledgebase projects, such as *Wikipedia*, *DBPedia*, and *Google Knowledge Graph*. Such initiation is beneficial in supporting semantic interoperability between a multitude of data sources that possibly exist in different domains;

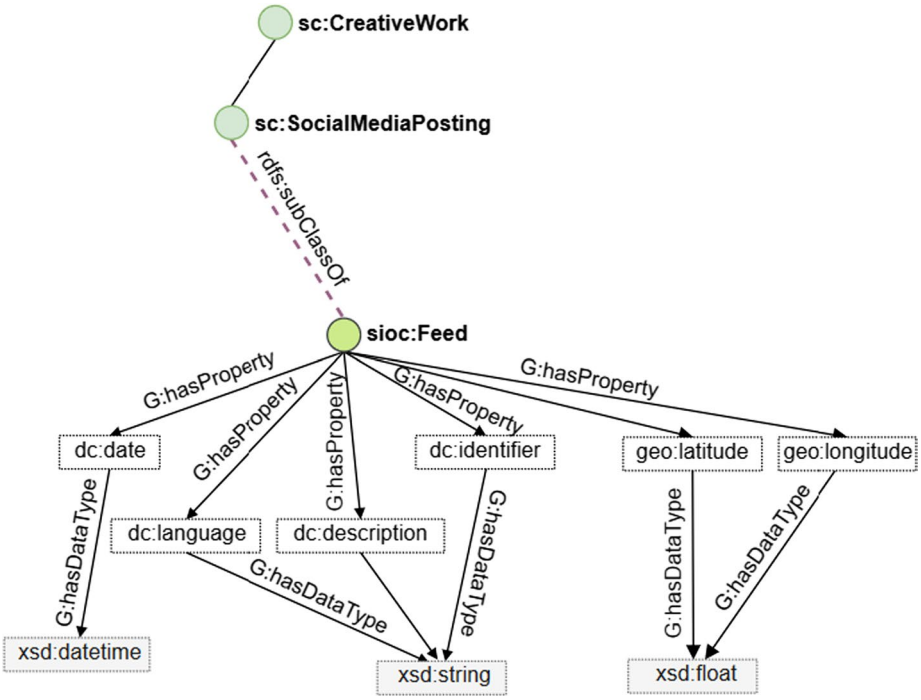

Fig. 3 Example of concept extension in $\mathcal{G}$

the disadvantage, however, is that SCHEMA abstract concepts can be too generic and require more specificity to support concise metamodeling and integration [34]. To balance between the conceptual abstraction and the semantic specificity, we enable $\mathcal{G}$ *extensibility*.

The elements of $\mathcal{G}$ may be extended by adding new properties to the current set of properties of a concept, $g \in \mathcal{G}$, to increase its coverage over elements in local schemas at the local schemata layer. g may also be extended by adding a new subordinate concept to it. *rdf:type* and *rdfs:subClassOf* are used for importing new and more specific concepts. To comply with $\mathcal{G}'s$ structure, the newly added concept must be associated with a set of properties (declared using *G:hasProperty*) and each property is of a certain primitive data type that is strictly reused from the XSD vocabulary [35] (declared using *G:hasDatatype*). Figure 3 depicts an example of extending *SocialMediaPosting*, a concept in $\mathcal{G}$, with *Feed*, a more specific concept imported from the SIOC vocabulary [36]. The extension taking place is to support a unified view over data ingested from social media data sources. *Feed* is linked to *SocialMediaPosting* using *rdfs:subClassOf*, and is described by a set of properties imported from the DCMI [37] and WGS84 [38] vocabularies. The required RDF data to implement such an extension are automatically generated by SemLinker and are added to the $\mathcal{G}$ ontology.

Local schemata layer

Schemas repository

The schemas repository S is the principal component in the local schemata layer. It stores the set of local schemas corresponding to different data sources that are added to SemLinker over time. Each local schema is stored in S as a data graph that contains

machine-readable metadata in the form of RDF triples to describe the physical schema details of the data ingested from the data source.

For each new data source i, SemLinker initializes a new empty RDF graph representing i's local schema, denoted as S_i. Subsequently, SemLinker requires S_i to be tagged with a concept $g \in \mathcal{G}$, so that g reflects the underlying type semantics of the data typically offered by i. Local schema tagging is normally modeled as an RDF triple, and follows the pattern:

$$\langle S_i M\text{: } isInstanceOf\ \ g \rangle$$

For example, to add data source Facebook to SemLinker (Fig. 2), a global concept "*Feed*" is used to tag Facebook, i.e. *Tag*(*Facebook*)=*Feed*, and the RDF interpretation of such tagging is asserted as

$$\langle S_{Facebook} M\text{: } isInstanceOf\ \ \mathcal{G}\text{:Feed} \rangle$$

The physical schema of any data source is subject to changes and updates [15, 39]. In the example depicted in Fig. 2, schema evolution is observed at both the semantic level (data attribute renaming, e.g. *"message"*↦ *"story"*, and *"text"*↦ *"message"*), and the structural level (data format changes, e.g. JSON↦XML, and attribute changes, e.g. casting the JSON object "*location*" in Fig. 2a into the simple attribute "*geo*" in Fig. 2b). SemLinker takes a novel, automatic approach to handle the schema revolution problem. In this approach, the RDF representation of the local schema of a data source is regarded as *dynamic*. It contains a changeable set of subgraphs, each of which represents an evolving version of the schema and is called a source schema. A schema extraction algorithm is used to extract source schemas automatically, and a mapping computation algorithm is responsible for mapping them to the global ontology. A formal definition of the local schema of a dynamic feature is given below.

Definition 2 (*Local schema*) The local schema $S_i \in S$ is a dynamic set of source schemas corresponding to m physical schema evolutions in the data ingested from the data source i, $S_i = \{S_{i1}, S_{i2}, \ldots, S_{im}\}$. For each $S_{ij} \in S_i$, $1 \leq j \leq m$, there is a set of mapping assertions $\mathcal{M}$ between S_{ij} and $g \in \mathcal{G}$ of the form: $p \rightarrow a$, where attribute $a \in S_{ij}$ and property $p \in g$.

The system ingests a data instance from its source's API that is typically associated with a release version. Analysis of the instance's physical schema is needed to obtain its source schema S_{ij}, where i is the data source's unique identifier (typically a URI), and j is i's API release version. SemLinker (fully/partially) then maps S_{ij} to the tagging concept g in the global ontology, stores it in the underlying graph of S_i, and uses it to integrate i's data with other data stored in the PDL. Furthermore, S_{ij} is regarded as a *benchmark* and is used to run schema checks on any new data instances ingested from i. A schema check may fail, and when the number of failures reaches a predefined threshold, the system infers that data source i has released its API with a newer version. Consequently a new evolution has occurred in the physical schema of i's data, and the system must augment the local schema S_i by constructing a new source schema, say S_{ik}, that is also mapped to g and added to S_i, so that S_{ik} is utilized to integrate any new data instances ingested from i

with the release version *k*, meanwhile utilizing S_{ij} to maintain backward integration support for the data instances that have already been ingested from *i* with the release version *j*. The procedure for augmenting the local schemas upon schema evolutions in the APIs of their data sources is automatically repeated to keep up-to-date metadata about the physical schema of the data instances ingested from each data source.

Schema extraction algorithm

The schema extraction algorithm automatically extracts source schemas from data instances (see List 1). It takes as input a data instance *F* ingested from a data source *i*, with release version *j*, and a *mime* string specifying the format type of *F*. *F* is assumed to conform to a known format specification [40], and its structure consists of a mix of flat and complex attributes, each of which has a *label* and a *value.* The algorithm operates on the structure level of *F* and extracts its RDF representation S_{ij} that consists of nodes and relationships between them. Each node in S_{ij} describes a specific element (attribute) in the physical schema of *F* and is associated with three constructs: *Identifier*, *Semantic Type*, and *Relation*. The algorithm assigns a value to each node and constructs *Identifier* using the URI of the data source and the release version *j* as base values. *Semantic Type* specifies the semantic class of the node, and its range is one of the classes, *𝒮:Attribute*, *𝒮:Object*, or *𝒮:Collection*. *Relation* refers to a relation between a pair of nodes, and can be *𝒮:hasAttribute*, *𝒮:hasObject*, *𝒮:hasCollection*, *𝒮:hasFormat*, *𝒮:isComposedBy*, *or 𝒮 :isDecomposedFrom.*

The algorithm has two main procedures: *InitializeGraph()* and *GenerateGraph()*. *InitializeGraph()* starts with specifying the given URI and the release version *j* as the root of S_{ij} (line 2); the auxiliary function *ToRDF()* adds format attributem (the input *mime)* to S_{ij} as one of its nodes (lines 3 and 4); *ToRDF()* then specifies the relationship (*:hasFormat*) between the format node and its parent node (line *5*). At this point, the source schema S_{ij} is initiated. The procedure then invokes *GenerateGraph()* and passes *F* and the root of S_{ij} to it.

```
1   function InitializeGraph(F, i , j , mime)
2     root ← i + '/' + j
3     format ← root + '/' + mime
4     S_ij ← S_ij ∪ ToRDF(format, rdf:type , S :Attribute )
5     S_ij ← S_ij ∪ ToRDF(root, S :hasFormat, format)
6     GenerateGraph(F , Root)
7   End
8   function GenerateGraph(F, ParentId)
9     foreach (label,value) ∈ F do
10      NodeId ← ParentId + '/' + label
11      NodeType ← Type(value)
12      S_ij ← S_ij ∪ ToRDF(NodeId , rdf:type , NodeType)
13      if NodeType = S :Attribute then
14        S_ij ← S_ij ∪ ToRDF(ParentId , S :hasAttribute , NodeId)
15      else
16        if NodeType = S :Object then
17          S_ij ← S_ij ∪ ToRDF(ParentId, S :hasObject , NodeId)
18          GenerateGraph(value,NodeId)
19        else
20          if NodeType = S :Collection then
21            S_ij ← S_ij ∪ ToRDF(ParentId, S :hasCollection , NodeId)
22            GenerateGraph(value,NodeId)
23      end if
24    end foreach
25  end
```

List 1 Schema extraction algorithm

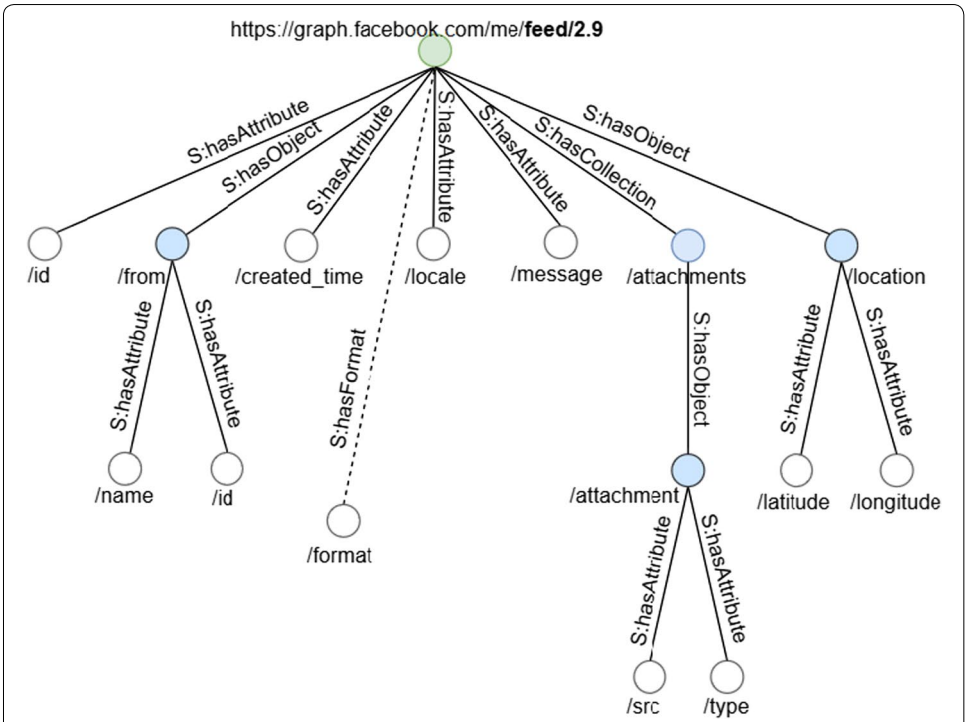

Fig. 4 Example of source schema extracted using the schema extraction algorithm

GenerateGraph() constructs S_{ij} through a series of iterations and recursive calls over the physical schema of F. In each call, the procedure takes a label-value pair from F and *parentId* (the URI) as input, creates a node in S_{ij} corresponding to the label, and links the node to its parent node (*parentId*). The initialization and linking of any node is modeled as the RDF triples *(NodeId rdf:type S:Type)* and *(ParentNodeId S:relation NodeId)*, respectively (line 12). Next *ToRDF()*, based on the type of the node, appends these triples to S_{ij} (lines 6–15). Depending on the complexity of F's structure, a label-value pair may represent a flat attribute in F (e.g. the "id" key in Fig. 2a), in which case, the node type obtained from the auxiliary function *Type()* is $\mathcal{S}$*:Attribute*, and the corresponding node is linked to its parent node using $\mathcal{S}$*:hasAttribute* relation, and the algorithm moves to the next label-value pair. Conversely, the current label-value pair may correspond to a complex attribute (e.g. the embedded object "location" in Fig. 2a). In this case, the type obtained from *Type()* is either $\mathcal{S}$*:Collection* or $\mathcal{S}$*:Object*, and the node is linked to its parent node using one of the relations $\mathcal{S}$*:hasCollection* or $\mathcal{S}$*:hasObject*, and subsequently the node's identifier and value are passed to the recursive procedure *GenerateGraph()*. Figure 4, as an example, depicts the graphical representation of the source schema extracted from the data sample given in Fig. 2a. The first node in the graph is created as a leaf node because the first label-value (JSON key "*id*") is a simple attribute in F. Its identifier is https://graph.facebook.com/me/feed/2.9/id. A node maybe embedded in another node, such is the case with the node labeled '*latitude*', which serves as one of the flat attributes of the object '*location*'.

Mapping computation and management

Once a source schema is constructed, it needs to be mapped to the global ontology. A mapping is a relationship specifying how an element structured under one schema (i.e., the source schema) corresponds to an equivalent element structured under the mediated schema (i.e., the global ontology $\mathcal{G}$) [19]. Mappings may be discovered either *implicitly* or *explicitly*. In SemLinker, because the global concepts of $\mathcal{G}$ are predefined independently from the data sources, it is likely that a source schema is semantically *incompatible* with the concepts of $\mathcal{G}$, and therefore no implicit mappings can be directly discovered between a source schema S_{ij} and a tagging concept g.

Typically, computing mappings between a source schema and a tagging concept involves specifying the semantic types of the source schema elements, i.e., labeling each schema element with a semantically equivalent property in the tagging concept [13]. However, semantic labeling alone is not sufficient [41], and a precise mapping computation process requires an extra step that specifies how the elements of a source schema should be organized in accordance with the structure of its tagging concept so that the two constructs become semantically compatible and ready for mapping. This 'extra step' is often missed in systems that automate the mappings [14, 24, 29, 42, 43] and is expected to be dealt with manually [41]. SemLinker uses a two-step mapping approach that not only does the explicit mappings, but also performs the 'extra step' automatically. The two steps are schema matching (SM) and virtual transformation of source attribute (VTSA).

Mapping algorithm

The mapping algorithm (see List 2) takes as inputs a tagging concept g, a source schema S_{ij}, and a threshold t. It takes two steps, SM and VTSA, to compute mappings between properties and source attributes. Mappings are established as RDF triples, where each mapping triple has the pattern $(p\,M : mapsTo\,a),\ p \in g,\ a \in S_{ij}$. Such modeling offers the flexibility of allowing multiple source attributes of multiple source schemas to be mapped to a single property. The source attributes mapped to the same property are considered semantically equivalent between themselves, so a unified view over them can be automatically represented by the property.

Revisiting the example in Fig. 2, we see that the Twitter data source is tagged with the concept "*Feed*". With the mappings specified below, "*text*" in $S_{Twitter,v1.1}$ (Fig. 2c) is regarded as semantically equivalent to "*message*" in $S_{Twitter,v1.2}$ (Fig. 3d).

$$\langle Feed\colon description\ M\colon mapsTo\ S_{Twitter,v1.1}\colon text\rangle$$
$$\langle Feed\colon description\ M\colon mapsTo\ S_{Twitter,v1.2}\colon message\rangle$$

Such mappings allow SemLinker to automatically reconcile heterogeneous attributes from different source schemas of the same data source, and the reconciliation can be further obtained by a SPARQL query with the pattern $\langle Feed\colon description M\colon mapsTo\ ?a\rangle$. In our running example, the result $?a = \{S_{Twitter,v1.1}\colon text,\ S_{Twitter,v1.2}\colon message\}$ allows an analysis process to access the values of both data attributes from both versions, $S_{Twitter,v1.1}$ and $S_{Twitter,v1.2}$. This can also be applied to unify semantically equivalent attributes in the source schemas of different data sources as long as they are tagged with the same concept. In our example, if we have both Facebook and Twitter data sources tagged

with the same concept *Feed*, then "message" in $S_{Facebook,v2.9}$, "story" in $S_{Facebook,v3.0}$, "text" in $S_{Twitter,v.1.1}$, and "message" in $S_{Twitter,v1.2}$ are all regarded as equivalent.

```
1  function ComputeMap(g , S_ij , t)
2   foreach p ∈ g do
3    result ← Matcher(p, g, S_ij, t)
4    if result(A) ≠ ∅ then
5     if result(|A|) = 1 and result(|P|) = 1 then
6      S_ij ← S_ij ∪ ToRDF(p, M: MapsTo,result(A[0]))
7     else
8      if result(|A|) > 1 and result(|P|) = 1 then
9       parent ← Parent(result(A[0]))
10      newNode ← parent + '/' + Label(p)
11      S_ij ← S_ij ∪ ToRDF(newNode, rdf: Type, S: Attribute)
12      S_ij ← S_ij ∪ ToRDF(parent, S: hasAttribute,newNode)
13      S_ij ← S_ij ∪ ToRDF(p, M: MapsTo,newNode)
14      for h = 0 upto result(|A|) – 1 do
15       DeleteRelation(parent, S_ij, result(A[h]))
16       S_ij ← S_ij ∪ ToRDF(result(A[h]),S: isComposedBy,newNode)
17      end for
18     else
19      if Type(result(A[0])) ≠ S: Object then
20       UpdateType(S_ij, result(A[0]), S: Object)
21      end if
22      for h = 0 upto result(|P|) – 1 do
23       newNode ← result(A[0]) + '/' + Label(result(P[h]))
24       S_ij ← S_ij ∪ ToRDF(newNode,rdf: type, S: Attribute)
25       S_ij ← S_ij ∪ ToRDF(result(A[0]), S: hasAttribute,newNode)
26       S_ij ← S_ij ∪ ToRDF(newNode, S: isDecomposedForm,result(A[0]))
27       S_ij ← S_ij ∪ ToRDF(result(P[h]), S: M: mapsTo,newNode)
28      end for
29      Skip(result(P))
30     end if
31    end if
32   end for
33  S_i ← S_i ∪ S_ij
34 End
```

List 2 Mapping algorithm

Schema matching

For each property p (line 2), the mapping algorithm invokes function *Matcher()* to find the attribute in S_{ij} that is semantically equivalent with p (line 3). *Matcher()* is an interface function that passes the matching task to an external schema matcher that has been plugged into the system. For each attribute a, it computes a score that quantifies the semantic correspondence between a and p. If the score is larger than the threshold t, a and p are regarded as semantically equivalent.

When there is more than one property equivalent to the same source attribute, or more than one source attribute equivalent with the same property, the algorithm, before a mapping is established, adjusts the structure of S_{ij} using VTSA.

Matcher() returns a data structure containing two collection constructs, A and P; while A holds zero or more source attributes, P holds one or more properties. The algorithm decides its next step according to what is returned in the A and P constructs.

If $A = \emptyset$ (line 4), no match is found and the algorithm proceeds to the next p.

If $|A| = 1$ and $|P| = 1$ (line 5), one matching attribute a of the source schema is found, so the algorithm establishes a mapping between p and a (line 6).

If $|A| > 1$ and $|P| = 1$ (line 8), an operation called *Composition* is performed on the attributes of A before establishing mappings (lines 9–17).

If $|A| = 1$ and $|P| > 1$ (line 18), an operation called *Decomposition* is performed on the attribute a stored in A before establishing mappings (lines 19–28). After the operation, P is skipped from g using the auxiliary function *Skip()* for optimization purposes (line 29).

While typically information regarding the concept g is abundant, information regarding a specific input S_{ij} is often inadequate [44]. When a situation like this arises, SemLinker uses matchers from third parties to handle schema matching tasks. Matchers are classified into three groups, schema-level, instance-level, and hybrid matchers [45]. Schema-level matchers utilize the information available in input schemas to find matches between schema elements. Instance-level matchers use statistics, metadata, or trained classifiers to decide if the values of two schema elements match. Hybrid matchers combine both mechanisms to determine match candidates. Schema matching approaches are constantly evolving, and often they apply other techniques such as dictionaries, thesauri, and user-provided match or mismatch information [44].

After every single property is examined, and mappings between g and S_{ij} are established, the underlying RDF data of the newly constructed S_{ij} are added into the local schema S_i (line 33).

Virtual transformation of source attribute

In Fig. 2, "*latitude*" and "*longitude*", the two properties in the concept *Feed*, correspond *directly* to the flat attributes of the embedded object labeled "*location*" in Fig. 2a, but correspond *indirectly* to the flat attribute labeled "*geo*" in Fig. 2b. The relationships between "*latitude*" and "*longitude*" and their indirect corresponding source attribute "*geo*", though apparent, can semantically hold only if "*geo*" is transformed into two new source attributes, i.e., "*geo*" $\rightarrow \langle$"*latitude*", "*longitude*"$\rangle$, or vice versa. To preserve the structure of the raw data stored in the lake, we adopt two virtual transformation operations, *Composition* μ and *Decomposition* γ, to work on the schema of the raw data rather

than on the data themselves. The virtual transformation operations are based on [46, 47], and they allow SemLinker to virtually map an attribute in a source schema to a property in the global ontology.

Definition 3 (*Composition* μ) Given a set of source attributes A, $A = (a_1, a_2, \ldots, a_k)$, $A \subseteq S_{ij}, S_{ij} \in S_i, 1 < k \leq n, n = |S_{ij}|$, the composition operator μ_{A,a_μ} composes A into a single virtual attribute a_μ.

The mapping algorithm uses the condition ($|A| > 1$, and $|P| = 1$) as the heuristic rule to compose a subset of source attributes $A, A = (a_1, a_2, \ldots, a_k)$ as a single new attribute a_μ and adds it to the S_{ij}. Since a_μ is a new source attribute, it must be initialized in the same manner as other source attributes. Two types of mappings are established to activate the composition transformation. Mapping $p_x \rightarrow a_\mu$ is performed by adding the RDF triple $\langle p_x M{:}mapsTo\ a_\mu \rangle$ to S_{ij} (line 13); mapping $A \rightarrow a_\mu$ is performed by adding a set of RDF triples, each following the pattern $\left\langle a_y \mathcal{S}{:}isComposedBy\ a_\mu \right\rangle$ (lines 14–17). Since a_μ is a virtual attribute that has no physical implementation, its data values are dynamically constructed when queried.

Definition 4 (*Decomposition* γ) Given an attribute $a_y \in S_{ij}$, $S_{ij} \in S_i$, the decomposition operator γ_{a_y,A_γ} decomposes the attribute a_y into a set of virtual attributes A_γ, where $A_\gamma = \{a_{\gamma 1} a_{\gamma 2}, \ldots, a_{\gamma k}\}$, $k > 1$.

When $|A| = 1$, and $|P| > 1$ (line 18) a decomposition operation takes place to decompose a source attribute a_y into a set of new virtual attributes A_γ, and adds the set to S_{ij}. In the operation, a_y is modeled as the parent node of the new virtual attributes (lines 23–25). Similar to composition, the algorithm establishes two types of mappings to activate the decomposition transformation. Mapping $p_x \rightarrow a_{\gamma i}$ is materialized through the RDF triple $\langle p_x M{:}mapsTo\ a_{\gamma i} \rangle$, and mapping $a_{\gamma i} \rightarrow a_y$ is materialized through the RDF triple $\left\langle a_{\gamma i} \mathcal{S}{:}isDecomposedFrom\ a_\gamma \right\rangle$. Since A_γ is a set of virtual attributes that have no actual implementation, the value of each attribute $a_{\gamma i}$ in A_γ must be dynamically constructed whenever needed.

Partial unified views

Instead of providing a strictly unified view that requires full mapping between the global schema and local schemas as is normally supported in rigorous data modeling, SemLinker allows a partial unified view and gives users control over the scope of the view. The scope of the partial unified view can be adjusted by adding or removing properties in the global ontology.

Figure 5 depicts a sample of two source schemas and their mappings to properties of a tagging concept in the global ontology. The source schemas are extracted from Facebook data samples given in Fig. 2b, using the schema extraction algorithm, and the mappings are computed using the mapping algorithm. In the figure, red circles indicate normal source attributes mapped to the equivalent properties in

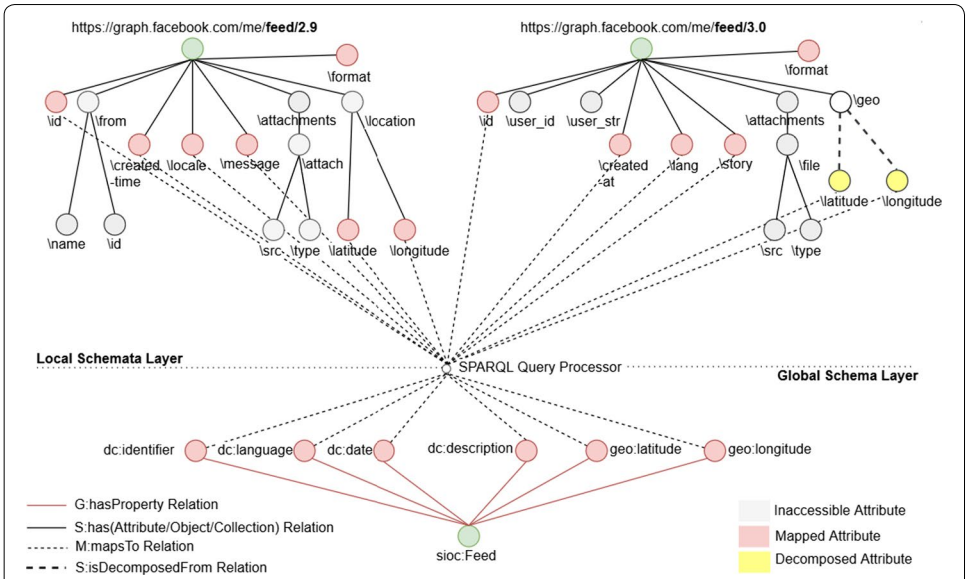

Fig. 5 Mappings between the concept "*sioc:Feed*" and two source schemas

straightforward schema matching operations. The source attribute '*geo*' in the source schema https://graph.facebook.com/me/feed/3.0 is marked by a white circle to indicate that decomposition has taken place, and '*geo*' is decomposed into virtual source attributes, namely, "*latitude*" and "*longitude*". The virtual source attributes (yellow circles) are also mapped to their corresponding global properties "*geo:latitude*" and "*geo:longitude*". Nonetheless, not all source attributes are mapped to properties of the tagging concept. Some source attributes (gray circles) are *inaccessible*, such as "*attachments*" of the second local schema. An inaccessible attribute, without an equivalent property in the global ontology, cannot be accessed through queries.

Query

The Query engine of the global schema layer (see Fig. 1) provides querying services for SemLinker. It serves two purposes: (i) to provide an SQL abstraction for formulating SQL-like queries targeting the unified views over raw data, and (ii) to compile, translate, and execute SQL-like queries and return results to the users. The query engine takes a successfully compiled input query, converts it into a *relevance query* and an *unfolding query*, both of which are internal SPARQL queries. A relevance query is a SPARQL SELECT query derived from the input query based on the concepts embedded in its clause formulation, and its execution over $\mathcal{G}$ returns all conceptually relevant data sources. An unfolding query is the input query unfolded in a SPARQL formulation and is executed iteratively on the underlying RDF graphs of the relevant local schemas that have been found by the relevance query. The result of the iterative execution is a list of source attributes that correspond to properties of the concepts specified in the query. Once the source attributes have been identified, we have all the relevant metadata information regarding the query, and the last phrase of the query is to retrieve data values from data instances stored in the raw data lake

based on the returned metadata and assemble the results into a list before giving it to the user.

Here is a simple query scenario. Suppose a user (the same user of the example in Fig. 2) is interested in retrieving all social feeds, and their geolocations, after a specified date (e.g., 1/10/2017), and so she formulates the following query.

```
SELECT Feed.description,Feed.latitude,Feed.longitude

FROM sioc:Feed Feed WHERE

ParseTime(Feed.date,"dd/mm/yyyy")> "1/10/2017"
```

On receiving the query, the query engine compiles it, and based on the concept (*sioc:Feed*) used in the clause of the query, it forms the following relevance query:

```
SELECT ?s WHERE

{ ?s M:isInstanceOf Feed .}
```

This above relevance query is executed on the global ontology. A successful execution returns all local schemas that are tagged with the concept *Feed*. In our case (assuming Facebook and Twitter are the only local schemas tagged with the concept *Feed*), $S_{Facebook}$, $S_{Twitter}$ are returned. Next, the query engine unfolds the input query and generates the following unfolding query, executing it iteratively on the RDF graphs of each local schema it has found, i.e., $S_{Facebook}$, $S_{Twitter}$.

```
SELECT ?a1 ?a2 ?a3 WHERE

{  Feed:description M:mapsTo ?a1 .

   Feed:latitude M:mapsTo ?a2 .

   Feed:longitude M:mapsTo ?a3 . }
```

Table 1 lists the metadata returned from the execution of the above unfolding query. From the table, we see two matching local schemas, each with two source schemas, and their attributes corresponding to the properties indicated in the query. Two virtual source attributes from decomposition, "*latitude*" and "*longitude*", are among the source attributes returned.

Once the metadata information is obtained, the query engine retrieves and parses the corresponding data instances to retrieve data values matching to the specific source attributes and the filter condition. The nature of PDL, a marriage between data lake and data gravity pull, determines that data are stored in their native formats, and third-party apps and tools can serve the lake as plugins (gravity pull) so

Table 1 Schema metadata results from a query

Local S.	Source S.	Query properties	Result attributes
$S_{Facebook}$	$S_{Facebook,v2.9}$	Description, latitude, longitude	Message, latitude, longitude
$S_{Facebook}$	$S_{Facebook,v3.0}$	Description, latitude, longitude	Story, geo (latitude, longitude)
$S_{Twitter}$	$S_{Twitter,v1.1}$	Description, latitude, longitude	Text, coordinates (latitude, longitude)
$S_{Twitter}$	$S_{Twitter,v1.2}$	Description, latitude, longitude	Message, coordinates (latitude, longitude)

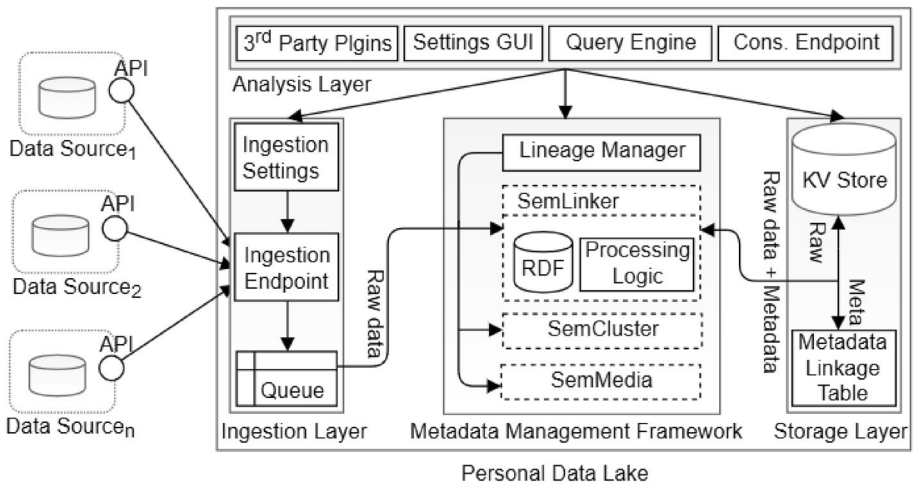

Fig. 6 The architecture of PDL integrated with SemLinker-based MMF

that any special need for a specific data type can be dealt with professionally [6]. For our instance, both the source schemas of $S_{Facebook}$ and $S_{Twitter,v1.1}$ use JSON, whereas $S_{Twitter,v1.2}$ uses XML. A suitable plugin parser for each data instance is chosen in order to parse it.

Integration with PDL

SemLinker, specializing in structured and semi-structured raw data integration, together with SemCluster [48], which tags free-text documents with key phrases that are associated with ontology-based semantics, and SemMedia, which extracts and manages metadata for multimedia data, constitute the MMF for PDL. Figure 6 depicts the architecture of the PDL, in which SemLinker is directly connected to the ingestion, and storage layers of the lake. The ingestion layer temporarily stores incoming data instances in a message queue before being preprocessed and dispatched to the storage layer. Through a cross-layer data pipeline SemLinker pulls data instances from the message queue, extracts their schema metadata, and dispatches the data instances and their associated metadata to the unified storage repository in the storage layer. The storage layer consists of a key-value tuple store and a simple hash table called the *Linkage Table*. The MMF associates each data instance and its metadata with a hash key representing the data instance's unique identifier and stores the hash key with the data instance itself as a key-value pair in the key-value store. Subsequently, the hash key and the data instance's metadata are stored as a key-value pair in the linkage table. Note that the metadata record associated with a data instance includes various information produced from different MMF components, such as lineage metadata (e.g., creation date, last access date), and security metadata (e.g., access control, encryption information). The RDF graph identifier (URI) of the local schema, and the subgraph identifier of the source schema (i.e., i and j) are also stored in the record, whereas the actual schema are stored in the schema repository.

Table 2 Datasets used in the evaluation

Source	Type	Format	#Attr.	#Size	#Evol.
HAR-1 [66]	Scien.	CSV	10	3,540,962	0
HAR-2 [66]	Scien.	CSV	10	3,205,431	0
HAR-3 [57]	Scien.	CSV	4	200,471	0
Facebook [58]	Social	JSON	17	19,770	4
Twitter [59]	Social	JSON	19	169,000	2
Foursquare [60]	Social	JSON	17	15,712	2
Flickr [61]	Social	XML	10	20,000	3
TripAdvisor [62]	Social	Spreads	13	19,998	4
Tourpedia [63]	Social	JSON	7	115,732	3
EnglandPubs [64]	Public	CSV	9	51,566	4
OpenPostCode [65]	Public	CSV	7	2,525,575	1

#Attr: Number of attributes; #Evol: Number of Schema Evolutions

Experimental evaluation

The evaluation of SemLinker is carried out in two phases. The first phase examines the accuracy of SemLinker's mapping computations on data with substantial heterogeneities and frequent schema evolutions. The second phase investigates integration effectiveness and the runtime functional complexity of SemLinker. Because of the difficulty in collecting personal data and privacy concerns, we do not use personal data in the evaluation, but instead use 11 real-world publicly available datasets (see Table 2) that exhibit a high degree of heterogeneity and have frequent schema evolutions. Each dataset consists of many data instances, and each data instance may be of a different release version. The collection include 3 datasets that contain sensor (accelerometer and gyroscope) streams generated by smart phone and smart watches carried by human subjects and are collected from two human activity recognition experiments [49, 50]. Also included are a mix of social data, some of which concerns user opinions, reviews, and ratings of popular places in London such as hotels, restaurants, and pubs, and two public datasets published by UK government agencies.

To integrate the above datasets into the system, we first need to tag the data with the most relevant concept of the global ontology $\mathcal{G}$. For example, "*sc:Review*" is used to tag the TripAdvisor and Tourpedia datasets; "*sc:LocalBusiness*" to tag the EnglandPubs dataset; and "*sc:PostalAddress*" to tag the OpenPostCode dataset. $\mathcal{G}$ is also extended to include more specific concepts, such as "*G:SensorReading*" (extension to "*sc:Dataset*") to tag the HAR-1, HAR-2, and HAR-3 datasets, and "*sioc:Feed*" (extension to "*sc:SocialMediaPosting*") to tag the Facebook, Twitter, Flickr, and Foursquare datasets. Full information about the global ontology and its extension used in the evaluation is available at [51].

Automatic mapping management evaluation

Here we evaluate the accuracy of SemLinker's mapping computations between the schema of each dataset and its tagging concept. Three matcher plugins are used. The first two, SemanticTyper [13] and AgreementMaker [43], are both open source matching approaches [52, 53]. SemanticTyper is an instance-level schema matcher that collects statistical information about the data based on their type and decides if

two schema elements match. AgreementMaker comprises multiple automatic matchers that are grouped into three layers. Each layer uses a different representation and similarity comparison measure, with the third layer being a combination of the other two. For AgreementMaker, because of the data lake lacks structural information in schemas, we use only the first layer, which represents features of schema elements (labels, comments, instances, etc.) in TF.IDF vectors and compares their similarities using the cosine similarity metric or some string-based measures (such as edit distance and substrings). The third plugin, SemMatcher, is the system's default matcher. A detailed discussion of SemMatcher is beyond the scope of the paper, however, SemMatcher is built as a combination of AgreementMaker, the schema-level matcher, and SemanticTyper, the instance-based matcher. It is linguistics-based and measures the similarity between two schema elements based on their textual descriptions retrieved from an external schema dictionary [44].

We measure the *accuracy* of SemLinker mappings against gold standard mappings—the mappings manually obtained using the Karma tool [54] and use the following formula to compute the accuracy score for SemLinker's mapping computations:

$$Acc(S_{ij}, g) = \frac{M_{SemLinker}(S_{ij}, g)}{M_{Gold}(S_{ij}, g)} \tag{1}$$

where $M_{SemLinker}(S_{ij}, g)$ is the number of correct mappings between S_{ij} and g that are automatically computed by SemLinker, and $M_{Gold}(S_{ij}, g)$ is the total number of mappings established using Karma. The following formula is used to obtain an overall accuracy score:

$$Ave(S_i, g) = \sum_{v=1}^{j} Acc(S_{ij}, g) \tag{2}$$

where j is the total number of evolutions in the physical schema of i.

The evaluation was run three times, each using a different schema matcher. Figure 7 displays the comparison results of the overall accuracy score when using different schema matchers. The results demonstrate clearly that the accuracy of a mapping calculation is very much determined by the schema matcher that is used, and that the system's own matcher, SemMatcher, outperforms the other two matchers in most of the datasets. SemanticTyper successfully captures correct matches wherever AgreementMaker fails, and this, to an extent, explains why SemMatcher, which combines

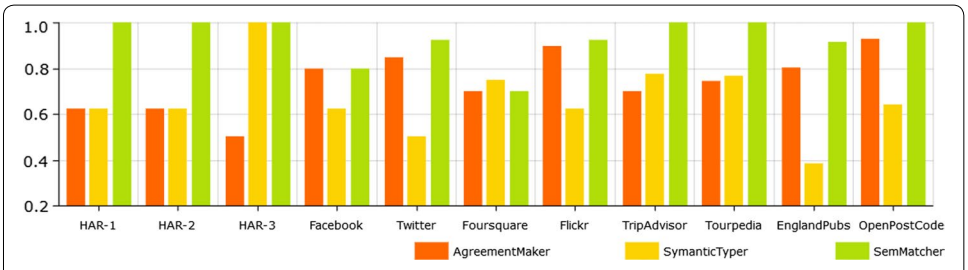

Fig. 7 The overall precision score of mapping calculations using different matchers

the best features of the matchers, gets almost full scores on 6 of the datasets. The fact that SemMatcher is also linguistics-based suggests that, for social and public datasets, by providing proper schema-level linguistic information (e.g., meaningful attribute labels), schema matching may achieve a better precision.

Functional efficiency and query complexity evaluation

To evaluate the functional efficiency of SemLinker in integrating big data with frequently changing schemas and the time complexity of executing queries, we compare SemLinker with a similar integration-oriented and ontology-based prototype system that is used in the SUPERSEDE project and is discussed in [7] (we refer to this as the BDI Ontology system). The BDI Ontology system prototype is implemented using a MongoDB [55] database backend to store JSON data, and SQL to store CSV and XML data. The downside of using the BDI Ontology system is immediately apparent as substantial effort (including manual interactions) is required to maintain its global ontology and to manage the source attributes found in the data collected from data sources. Each schema evolution also requires manual (re)mappings. Two scenarios are used in the evaluation:

Scenario 1 (involving datasets HAR-1, HAR-2, and HAR-3):

It is assumed we want to retrieve all gyroscope readings ingested from gyroscope sensors to pass to a specialized HAR application for HAR analysis. For this purpose, the following query is formed:

```
SELECT Sensor.reading

FROM sc:SensorReading Sensor

WHERE Sensor.sensor-type = "gyroscope"
```

A gyroscope reading, such as *[0.0041656494, − 0.0132751465, 0.006164551] (see HAR-3 dataset),* consists of values corresponding to the x, y and z axes. The global concept "*SensorReading*", which has one property, "*G:reading*", has been used to tag all three HAR datasets. This apparently simple Gyroscope reading retrieval has some complexity: the reading in HAR-1&2 is described by three separate attributes, *X*, *Y*, and *Z*, whereas the reading in HAR-3 is described by only one attribute. However, before the query takes place, this heterogeneity problem has been solved when HAR-2 data are ingested and the schema mapping takes place. In the mapping, the source schema, $S_{HAR\text{-}2,V1.0}$, is virtually transformed into the virtual source attribute "*reading*". Consequently, the query alone is sufficient to retrieve the required data without any extra pre- or post-processing steps.

However, for the BDI Ontology system, since there is no easy solution for structural heterogeneities in the source schemas, it is impossible to execute the query directly. Either we must transform the data so that HAR-1&2 and HAR-3 share the same structure, or we tag them with different concepts and query them separately.

Scenario 2 (involving social and public datasets):

It is assumed we are interested in local businesses in London such as hotels, restaurants, and pubs, and would like to know their full address (including postcode), and reviews and ratings about them. We may also apply sentiment analysis to gauge the polarity in the comments that are retrieved.

Because the data exist in different formats and semantic contexts, a direct query may seem to be complicated. With SemLinker, however, provided the raw data are ingested properly into the system, and predefined inline functions (another kind of plugin of the lake) that fulfill certain relevant tasks are in place, a direct query will return the desired results. In the datasets, there are defects in the data, such as a postcode missing from the reviewed business, or some geolocation is inaccurate, or the name of a business may have different spelling (using "*&*" for "*and*" or "*65*" for "*sixty-five*", and so on). If such problems have been anticipated, as in our case, customized inline functions may be designed and imported in advance to deal with these situations at query time. Here we use *Sentiment(string)* to produce a polarity representing the user opinion (i.e., positive, negative, or neutral), *Normalize(string)* to normalize the business names, *Radius(float,integer)* to generate *x* values around an input spatial coordinate, and *WordNet(string,int)* (a WordNet-based function) to retrieve all the possible synonyms and hyponyms of an input string. In addition, there is also the complication regarding schema evolution that has already been dealt with by SemLinker. Once all elements are in place, the user can retrieve the desired information using the following query:

```
SELECT
  sc:LocalBusiness.name,
  Review.reviewRating,
  Sentiment(Review.reviewBody),
  sc:PostalAddress.postcode
FROM
  sc:Review Review
  JOIN sc:LocalBusiness ON
  sc:LocalBusiness.name = Normalize(Review.name)
  JOIN scPostalAddress ON
  scPostalAddress.latitude =
  Radius(sc:LocalBusiness.latitude,5)
  AND scPostalAddress.longitude =
  Radius(sc:LocalBusiness.longitude,7)
WHERE
  Review.about IN
  (WordNet("hotel",1),WordNet("restaurant",1))
  AND Review.location = "london"
```

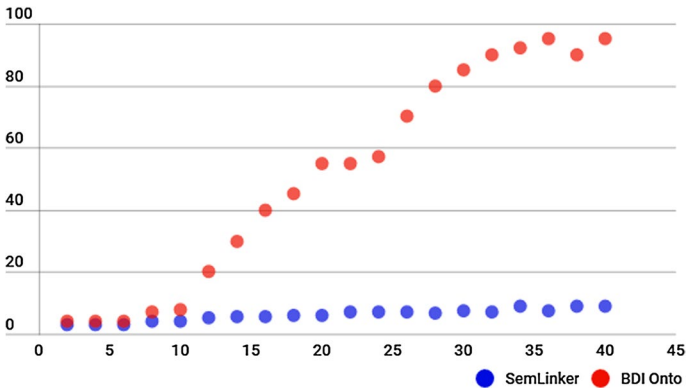

Fig. 8 Real time query execution performance comparison

In the query, the postcode associated with each review is obtained by chaining data instances across multiple datasets.

For both scenarios, we ran 20 queries targeting raw data in the range 0–800K data instances on both SemLinker and the BDI Ontology prototype system, and we measure and compare their query execution time. The recorded time of each query includes input query translation, query unfolding, and data retrieval from the backend. Figure 8 presents the runtime benchmark data recorded for the query executions of each system. We observe that when the number of datasets is small, the difference in the execution times for the two systems is insignificant, but when the retrieved data are moderately large, SemLinker significantly outperforms the BDI Ontology system. For example, SemLinker requires 8 s on average to retrieve and integrate 40K review results, whereas the BDI ontology system requires 96 s on average to perform the same task. SemLinker's significant improvement is mainly due to the following reasons: (i) since SemLinker fully supports the storage, integration, and querying of raw data regardless of their formats and structures, any high-performance key-value store can be adopted as the central unified backend (e.g., Redis [56]). Hence, compared to the data access and query execution overheads of the BDI Ontology system (MongoDB/SQL), SemLinker's backend is conceptually a big hash table with data access complexity *O(1)*; and (ii) in SemLinker the source schemas of each dataset are modeled as subgraphs grouped into one RDF graph (i.e., the local schema), whereas in the BDI Ontology system each source schema is treated as a separate RDF graph. As expected, SemLinker, which executes its internal SPARQL queries on a single graph for each dataset, is much faster than the BDI Ontology system, which executes its queries on several graphs for each data set.

Conclusion and future work

We have presented SemLinker, an ontology-based data integration system for PDL and other similar data lake implementations. SemLinker allows casual users with limited technical background and with minimal effort, to integrate, process, and analyze heterogeneous raw data through a unified conceptual representation of the data schemas regarding a widely used global ontology. To the best of our knowledge, SemLinker is the first domain-agnostic integration system that offers self-adapting capabilities to

automatically integrate big data with frequently evolving schemas based on solid theoretical foundations. SemLinker has been evaluated on large datasets in multiple domains, and the results not only validate its integration effectiveness and functional efficiency, but also indicate that SemLinker's performance is robust and promising, albeit there is still room for improvement in multiple aspects of the system.

Although SemLinker is a generic integration solution, it targets only structured and semi-structured data, and it is, by no means, a holistic integration solution when unstructured data such as free-text documents and multimedia files are also considered. For such data we have proposed, in an earlier paper [48], SemCluster, an automatic key phrase extraction tool that specializes in extracting keyphrases from free text documents and annotating each keyphrase with ontology-based metadata. One of our planned immediate undertakings is to combine SemLinker and SemCluster into a broader integration solution towards an effective and efficient metadata management framework for the personal data lake.

In this paper, we have discussed the importance of automating the tasks of the integration process, thereby building an easy-to-use data lake for casual users. However, SemLinker, though in many aspects it can be regarded as an automatic system, still has two vital tasks that need to be dealt with manually by its users: data source tagging and selecting a schema matcher plugin. Using machine learning based approaches to label data sources with ontological concepts automatically, and thus relieving users of the burden of manual data source tagging, is one of our future research goals. As for schema matcher selection, though the performance of SemMatcher in the evaluation is promising, we intend to extend it by combining a number of other matching approaches, so that it provides a good matching solution for schemas of various characteristics, reducing the need for users to resort to other schema matchers.

Abbreviations

PDL: personal data lake; ETL: extract-transform-load process; API: Application Programming Interface; BDI: big data integration; MMF: metadata management framework; SM: schema matching; VTSA: virtual transformation of source attribute; OWL: Web Ontology Language; RDF: resource description framework; DCMI: Dublin core metadata initiative vocabulary; XSD: XML schema definition vocabulary; RDFS: RDF schema vocabulary; SIOC: Semantically-Interlinked Online Communities vocabulary; SPARQL: recursive acronym for SPARQL Protocol and RDF Query Language; WordNet: the lexical database WordNet; TF: term frequency; IDF: inverse document frequency.

Authors' contributions

Both authors contributed to the investigation of the research hypothesis and the initiation of the research. Hassan Alrehamy is the main force in developing and evaluating the SemLinker system, while Coral Walker took on a supervisory role and oversaw the completion of the work. Both authors contributed significantly in producing the paper. Both authors read and approved the final manuscript.

Author details

[1] School of Computer Science and Informatics, Cardiff University, Queens Building, Cardiff, UK. [2] College of Information Technology, Babylon University, Babylon, Iraq.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

In a GitHub repository associated with our paper, we include all datasets used in our experiments. In addition, we also provide the scripts to reproduce our analysis [51–54, 57–65]. The repository is publicly available on GitHub at [51].

Ethics approval and consent to participate

Not applicable.

Funding
Not applicable.

Publisher's Note

Received: 29 January 2018 Accepted: 14 March 2018
Published online: 26 March 2018

References

1. Erhard R. The case for holistic data integration. In: East European conference on advances in databases and information systems. Berlin: Springer; 2016.
2. Jagadish HV, Gehrke J, Labrinidis A, Papakonstantinou Y, Patel J, Ramakrishnan R, Shahabi C. Big data and its technical challenges. Commun ACM. 2014;57(14):86–94.
3. Ponniah P. Data extraction, transformation, and loading. New York: Wiley; 2001.
4. Dixon J. Pentaho, hadoop, and data lakes. James Dixon Blog. http://www.pentaho.com/blog/2010/10/15/pentaho-hadoop-and-data-lakes. Accessed 25 Dec 2017.
5. Quix C, Hai R, Vatov I. Metadata extraction and management in data lakes With GEMMS. Complex Syst Inf Model Quart. 2016;9(16):67–83.
6. Walker C, Alrehamy H. Personal data lake with data gravity pull. In: 2015 IEEE fifth international conference on Big data and cloud computing (BDCloud); 2015.
7. Nadal S, Romero O, Abelló A, Vassiliadis P, Vansummeren S. An integration-oriented ontology to govern evolution in big data ecosystems. EDBT/ICDT workshops; 2017.
8. Apache Hadoop. http://hadoop.apache.org/. Accessed 25 Dec 2017.
9. Jones W. A review of personal information management. IS-TR-2005-11-01. The information school technical repository. Washington: University of Washington; 2005.
10. Dong XL, Srivastava D. Big data integration. In: 2013 IEEE 29th international conference on data engineering (ICDE); 2013.
11. Abelló A. Big data design. In: Proceedings of the ACM eighteenth international workshop on data warehousing and OLAP. New York: ACM; 2015.
12. Shvaiko P, Euzenat J. Ontology matching: state of the art and future challenges. IEEE Trans Knowl Data Eng. 2013;25(1):158.
13. Ramnandan S, Mittal A, Knoblock C, Szekely P. Assigning semantic labels to data sources. In: European semantic web conference. Cham: Springer; 2015.
14. Peukert E, Eberius J, Rahm E. A self-configuring schema matching system. In: 2012 IEEE 28th international conference on data engineering (ICDE); 2012.
15. Ramanathan V, Brickley D, Macbeth S. Schema. org: evolution of structured data on the web. Commun ACM. 2016;59(16):44–51.
16. Manousis P, Vassiliadis P, Zarras A, Papastefanatos G (2015) Schema evolution for databases and data warehouses. In: European business intelligence summer school. Berlin: Springer; 2015.
17. Curino C, Moon H, Deutsch A, Zaniolo C. Automating the database schema evolution process. VLDB J. 2013;22(13):73–98.
18. Andany J, Léonard M, Palisser C. Management of schema evolution in databases. In: VLDB. 1991. p. 161–70.
19. Lenzerini M. Data integration: a theoretical perspective. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems. New York: ACM; 2002. p. 233–46.
20. Gruber T. A translation approach to portable ontology specifications. Knowl Acquisit. 1993;5(93):199–220.
21. Giese M, Soylu A, Vega-Gorgojo G, Waaler A, Haase P, Jiménez-Ruiz E, Lanti D. Optique: zooming in on big data. Computer. 2015;48(15):60–7.
22. Calvanese D, Cogrel B, Komla-Ebri B, Kontchakov R, Lanti D, Rezk M, Rodriguez-Muro M, Xiao G. Ontop: answering SPARQL queries over relational databases. Semantic Web. 2017;8(17):471–87.
23. Marcos M, Maldonado J, Martínez-Salvador B, Boscá D, Robles M. Interoperability of clinical decision-support systems and electronic health records using archetypes: a case study in clinical trial eligibility. J Biomed Inform. 2013;46(4):676–89.
24. Cate B, Dalmau V, Kolaitis P. Learning schema mappings. ACM Trans Database Syst (TODS). 2013;38(13):28.
25. Varga J, Romero O, Pedersen T, Thomsen C. Towards next generation BI systems: the analytical metadata challenge. In: International conference on data warehousing and knowledge discovery, vol. 8646. Cham: Springer; 2014. p. 89–101.
26. Maccioni A, Torlone R. Crossing the finish line faster when paddling the data lake with kayak. Proc VLDB Endowment. 2017;10(12):1853.
27. Apache Atlas. http://atlas.apache.org/. Accessed 25 Dec 2017.
28. Apache Avro. https://avro.apache.org/. Accessed 25 Dec 2017.
29. Reis D, Cesar J, Pruski C, Reynaud-Delaître C. State-of-the-art on mapping maintenance and challenges towards a fully automatic approach. Expert Syst Appl. 2015;42(15):1465–78.
30. Scherzinger S, Cerqueus T, Cunha de Almeida E. Controvol: a framework for controlled schema evolution in nosql application development. In: 2015 IEEE 31st international conference on data engineering (ICDE). 2015. p. 1464–7.
31. McGuinness D, Van Harmelen F. OWL web ontology language overview. W3C Recommen. 2004;1010(4):2004.
32. Lassila O, Swick R. Resource description framework (RDF) model and syntax specification. W3C Technical Report. 1999. https://www.w3.org/TR/REC-rdf-syntax/

33. Mascardi V, Cordì V, Rosso P. A comparison of upper ontologies. In: WOA; 2007.
34. Heath T, Bizer S. Linked data: evolving the web into a global data space. Synth Lect Semantic Web. 2011;1(11):1–136.
35. XSD Vocabulary. https://www.w3.org/TR/xmlschema11-1/. Accessed 25 Dec 2017.
36. SIOC Vocabulary. http://rdfs.org/sioc/spec/. Accessed 25 Dec 2017.
37. DCMI Vocabulary. http://dublincore.org. Accessed 25 Dec 2017.
38. WGS84 Vocabulary. https://www.w3.org/2003/01/geo/. Accessed 25 Dec 2017.
39. Wang S, Keivanloo I, Zou Y. How do developers react to restful API evolution? In: International conference on service-oriented computing. Berlin: Springer; 2014. p. 245–59.
40. Media types listing by the internet assigned numbers authority. https://www.iana.org/assignments/media-types/media-types.xhtml. Accessed 25 De 2017.
41. Taheriyan M, Knoblock A, Szekely P, Ambite J. A scalable approach to learn semantic models of structured sources. In: 2014 IEEE international conference on semantic computing (ICSC); 2014. p. 183–90.
42. Shen W, Wang J, Han J. Entity linking with a knowledge base: Issues, techniques, and solutions. IEEE Trans Knowl Data Eng. 2015;27(15):443–60.
43. Cruz I, Antonelli F, Stroe C. AgreementMaker: efficient matching for large real-world schemas and ontologies. Proc VLDB Endowment. 2009;2(9):1586–9.
44. Madhavan J, Bernstein P, Doan A, Halevy A. Corpus-based schema matching. In: Proceedings 21st international conference on ICDE 2005 data engineering; 2005. p. 57–68.
45. Bernstein A, Madhavan J, Rahm E. Generic schema matching, ten years later. Proc VLDB Endowment. 2011;4(11):695–701.
46. Xu L, Embley D. Combining the best of global-as-view and local-as-view for data integration. ISTA. 2004;48:123–36.
47. Fagin R, Kolaitis P, Popa L, Tan W. Schema mapping evolution through composition and inversion. In: Schema matching and mapping. Berlin: Springer; 2011. p. 191–222.
48. Alrehamy H, Walker C. SemCluster: unsupervised automatic keyphrase extraction using affinity propagation. In: UK workshop on computational intelligence. Cham: Springer; 2017. p. 222–35.
49. Stisen A, Blunck H, Bhattacharya S, Prentow T, Kjærgaard M, Dey A, Sonne T, Jensen M. Smart devices are different: assessing and mitigating mobile sensing heterogeneities for activity recognition. In: Proceedings of the 13th ACM conference on embedded networked sensor systems. New York: ACM; 2015. p. 127–40.
50. Faye S, Louveton N, Jafarnejad S, Kryvchenko R, Engel T. An open dataset for human activity analysis using smart devices. 2017. hal-01586802, Version 1. https://hal.archives-ouvertes.fr/hal-01586802
51. SemLinker Experimental Evaluation Setup. https://github.com/alrehamy/SemLinker_Evaluation. Accessed 25 Dec 2017.
52. AgreementMaker Source Code Repoistory. https://github.com/agreementmaker/agreementmaker. Accessed 25 Dec 2017.
53. SemanticTyper Source Code Repository. https://github.com/tknandu/SemanticLabelingRepo. Accessed 25 Dec 2017.
54. Karma Web-based Integration Tool Source Code Repository. https://github.com/usc-isi-i2/Web-Karma. Accessed 25 Dec 2017.
55. MongoDB Database Homepage. https://www.mongodb.com. Accessed 25 Dec 2017.
56. Carlson J. Redis in action. New York: Manning Publications Co.; 2013.
57. Human Activity Recognition Dataset (HAR 3). https://hal.archives-ouvertes.fr/hal-01586802. Accessed 25 Dec 2017.
58. Facebook Open Graph API. https://graph.facebook.com. Accessed 25 Dec 2017.
59. Twitter Data Streaming API. https://api.twitter.com. Accessed 25 Dec 2017.
60. Foursquare API. https://api.foursquare.com/v2/. Accessed 25 Dec 2017.
61. Flickr API. https://api.flickr.com/services/rest/. Accessed 25 Dec 2017.
62. London Restaurants Reviews Dataset. https://www.kaggle.com/PromptCloudHQ/londonbased-restaurants-reviews-on-tripadvisor. Accessed 25 Dec 2017.
63. Tourpedia API. http://tour-pedia.org/api/. Accessed 25 Dec 2017.
64. United Kingdom Government open datasets, the food standards agency, food safety and food hygiene ratings dataset. http://ratings.food.gov.uk/open-data/. Accessed 25 Dec 2017.
65. United Kingdom Postal Codes Dataset. https://www.getthedata.com/open-postcode-geo. Accessed 25 Dec 2017.
66. Human Activity Recognition Dataset (HAR 1,2). https://archive.ics.uci.edu/ml/datasets/Heterogeneity+Activity+Recognition. Accessed 25 Dec 2017.