An overview of data integration principles for heterogeneous databases

Aleksandar Stojanović[1], Marko Horvat[2], Željko Kovačević[3]
[1]Zagreb University of Applied Sciences, Department of Informatics and Computer Science, Zagreb, Croatia
[2]University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Applied Computing, Zagreb, Croatia
[3]Zagreb University of Applied Sciences, Department of Informatics and Computer Science, Zagreb, Croatia
[1]aleksandar.stojanovic@tvz.hr
[2]marko.horvat3@fer.hr
[3]zeljko.kovacevic@tvz.hr

***Abstract* – Modern large-scale information systems often use multiple database management systems, not all of which are necessarily relational. In recent years, NoSQL databases have gained acceptance in certain domains while relational databases remain de facto standard in many others. Many "legacy" information systems also use relational databases. Unlike relational database systems, NoSQL databases do not have a common data model or query language, making it difficult for users to access data in a uniform manner when using a combination of relational and NoSQL databases or simply several different NoSQL database systems. Therefore, the need for uniform data access from such a variety of data sources becomes one of the central problems for data integration. In this paper we provide an overview of the main problems, methods, and solutions for data integration between relational and NoSQL databases, as well as between different NoSQL databases. We focus mainly on the problems of structural, syntactic, and semantic heterogeneity and on proposed solutions for uniform data access, emphasizing some of the more recent proposals.**

Keywords* – *data integration; database; SQL; NoSQL; heterogeneous

I. INTRODUCTION

Considering the variety of database systems available today, data integration between those systems becomes increasingly important, but also challenging. The goal of data integration is to provide users with homogeneous view of data distributed across different, *heterogeneous* data sources [1], usually relational and/or non-relational databases (Figure 1). The need for data integration arises for the following reasons:

- When databases are designed and implemented independently, but later need to be shared.
- When user requirements increase over time, and it is not possible to predict all future requirements.

As quantity of data and business requirements increase, the complexity of today's information systems increases as well. One consequence of this is that modern organizations often use multiple data sources, both relational and non-relational. Relational database systems are widely used today in a large variety of application areas. Such systems are based on a standardized data model (relational), query language (SQL), and offer capabilities like ACID consistency, schema normalization and transaction support

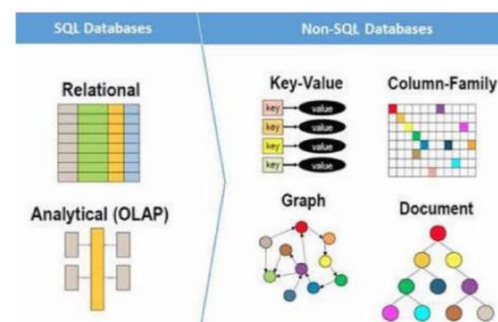

Figure 1. Variety of data models [2].

[1]. The standardized data model and query language allow users to transfer their skills to other such systems because all of them have similar capabilities, offer similar services, and differ only in details.

Regardless of their wide adoption, it has been argued that relational database systems have disadvantages in certain areas such as performance and rigidity of data model [3]. On the other hand, NoSQL database systems (or data sources), of which there are many, do not have a standardized data model or query language; they support variety of data models like document, graph, key-value, and extensible records, with API and the query language that directly support that data model. Because these systems are often made for more specific domains like stream processing, big data, and real-time web systems, the performance requirements for such applications usually necessitate a simpler approach to data modeling and handling [2]. Many NoSQL systems support *semi-structured* data using formats like JSON or XML. Additionally, for many internet-based applications, like social networks, scalability and performance is more important than data consistency and therefore in such cases ACID consistency does not have to be fully supported. For that reason, many NoSQL systems emphasize flexibility of the data model and simplicity of operations while supporting data consistency in a weaker form.

The aim of this paper is to present some recent developments in data integration, specifically between heterogeneous data sources. We describe three of the more recent proposals which the authors of this paper believe are most promising based on a thorough analysis of current trends and patterns of heterogeneous database integration.

Two of the proposals discuss uniform data access while the third one is an extension of the SQL language called SQL++. The rest of the paper is organized as follows. Section II provides some basic background on main aspects of data integration. Section III discusses basic data integration methods including the uniform data access which is the main topic of the paper. In section IV three data integration solutions are described: SOS, the general schema approach and SQL++. Some other approaches are also briefly mentioned in this section. Section V concludes the paper.

II. BACKGROUND

One of the main problems in data integration is *interoperability* [4] – uniform access to data from multiple heterogeneous data sources. Data sources can be heterogeneous from several aspects [5] [6]:

- Structure
- Contents
- Syntax
- Semantics

Structural heterogeneity means that data in different data sources are structured differently [6]. For example, the attribute "name" may be structured differently in different databases, such as first name followed by last name or vice versa. Structural heterogeneity can also be caused by different data models [7]. For example, in relational databases, schemas may be designed differently to show the same type of information.

Content heterogeneity occurs when two or more databases contain one type of data in different forms [6]. For example, a person's age may be given as a date of birth or as a number. Both forms provide the same information but are stored in different ways.

Syntactic heterogeneity exists when different languages are used [5]. For example, most relational database systems use SQL [1] as the standard data manipulation language while in NoSQL databases there are various languages in use and there exists no standard language. As an example, the document database MongoDB [8] uses JavaScript as the language for data manipulation.

The most important problem in data integration is *semantic heterogeneity* [6] [7] [9]. This type of heterogeneity occurs when the meaning of data is different in different contexts of the same domain [7]. Semantic heterogeneity at the schema level is caused by synonyms and homonyms [9]. The problem with synonyms occurs when the same entity is named differently in different databases. An example is the use of phone numbers: one database might call the data "contact number", another "phone number". The problem of homonyms occurs when different entities in different databases have the same name [9]. An example is the attribute "grade". In some educational systems, "grade" can mean the level of elementary or high school education (e.g., fifth grade), but it can also denote performance measured by grades from 1 to 5, 6 to 10, A to F, and so on, depending on the country and the educational system. The main reason for semantic heterogeneity is that databases are developed independently and by different people [10] with different understanding of the data and its meaning. In a database, it is not possible to define exactly what the meaning of the data is [11] in a way that would be valid for all databases.

III. OVERVIEW OF DATA INTEGRATION METHODS

Data integration can be performed at different levels of abstraction [12] where integration techniques can be grouped in the following way [7] [13]:

- Manual integration
- Creating integration application
- Using common user interface
- Using middleware interface
- Creating common database
- Developing a uniform data access

The first four approaches provide certain level of integration but have some serious drawbacks. Manual data integration is time consuming and expensive. An integration application is an application built such that it has access to all databases that need to be integrated and provides users with necessary data [7]. The problem with integration application is that it can access and handle only databases it was built to use and adding additional data sources further complicates the system. The approach of using a common user interface presents users with separate parts of data and the user must manually integrate those parts [7]. Middleware solutions such as OLE DB or ODBC can connect to various data sources, but do not have the ability to resolve structural and semantic differences between them. Creating a common database approach requires all data of interest to be manually migrated to the new database [14]. This is the extraction-transformation-load (ETL) process that is performed programmatically. The most demanding part of the process is the transformation which maps the data from the sources to the target database. Some problems with this approach are that it requires additional storage space, and the migrated data does not get automatically updated with the original data. To keep the new database in sync with the original sources the migration must be performed every time the original data changes. This is time consuming and costly.

Developing a uniform data access does not require manual data integration, migration of data nor additional storage space. Also, it does not suffer from issues related to integration application. Semantic heterogeneity of source databases presents the most important problem for uniform data access. This problem has been researched for decades and most proposed solutions are based on ontologies and can be performed manually or in a semiautomatic way [15]. Some areas where ontology-based data integration methods are used are web search [16] [17], geospatial information systems [18] [19], e-commerce [20] [21] and life sciences [22] [23].

IV. OVERVIEW OF DATA INTEGRATION SOLUTIONS

In this section we describe some of the more recent proposals and wrap up with a brief overview of some other data integration approaches. Data integration solutions can be divided into SQL-only, NoSQL-only and SQL/NoSQL. Since in this paper we are interested only in the last two,

the following approaches describe data integration from that perspective.

A. SOS (Save Our Systems)

The approach overviewed in this section is described in [24]. It is based on specifying a common interface that supports the following operations: put, get, and delete. It uses a data model like document data stores, i.e., it supports collections of nested objects. This model is called *model generic representation* and is exposed through a *hierarchical* path specified on the operations:

- *get(path)*
- *put(path, object)*
- *delete(path)*

The operation *get* returns a set of objects, operation *put* inserts or updates objects and *delete* deletes objects or object fields. The path specifies the tree structure for accessing database objects and has the form

$step_1/step_2/.../step_n$

Each step can represent one of the following:

- A collection
- An object identifier
- A field name

The following are some examples of the paths with *get* operation. The first example shows a *get* operation that retrieves all objects from collection *users*:

```
get("users")
```

The following *get* operation retrieves the object with id 255 that is stored in collection *users* and selects its field *name*:

```
get("users/255/name")
```

Data objects can contain sub-objects that can be retrieved in the same way. For example, if an object contains sub-object that represents tweets the user with id 255 has made, they could be retrieved in the same way as the previous example:

```
get("users/255/tweets")
```

The following example either updates (if the field is already there) or inserts (if it is not) the field *username* of the object with id 123:

```
put("users/123/username", "john69")
```

To support multiple data stores, operations like these need to be translated for each supported data store. This approach supports multiple NoSQL data stores, but here we show an example only for one such data store, HBase. HBase is an extensible record store. It supports a data model based on tables with sparse rows where columns can be grouped into *families*. Figure 2 shows an example of one such table. The table name is *User* and columns families are *Account*, *Personal* and *Friends*. The following

User			
	Account	Personal	Friends
1001	username = “bob1987” password = “thisisapassword”	firstName = “Bob” lastName = “Smith” ssn = “4hfe94”	2004:firstName = “Alice” 2004:lastName = “Smith” 2004:email = “alice@gmail.com” 1714:firstName = “Charlie” ...
2004	username = “alice”	...	...
1714	...	...	...

Figure 2. An example of HBase table [24].

Users			
	_top	Personal	Tweets[]
1001	username = “bob1987” password = “thisisapassword”	firstName = “Bob” lastName = “Smith” country = “Italy” city = “Rome”	[0]:id = “2039485563” [0]:text = “Hello World!” [0]:geo:longitude = “12.44751” [0]:geo:latitude = “41.83764” [1]:id = “4059382747” ...
1002	...	...	...

Usernames	
	_top
bob1987	_value = “1001”
alice	_value = “2004”
...	...

Log	
	_array[]
c1	[0] = “new user created” [1] = “query Q1 successfully applied” ...
c2	...

Figure 3. One translation of model generic representation into an extensible record store structure [24].

is an example of the model generic representation of the data in the table:

```
users: [
    1001: {
        username: "bob1987",
        password: "thisisapassword",
        personal: {
            firstName: "Bob",
            lastName: "Smith",
            country: "Italy",
            city: "Rome" },
    tweets: [{  id: "2039485563",
                text: "HelloWorld",
                geo: {
                    longitude: "12.44751",
                    latitude: "41.83764" }}
             {id: "4059382747", … }
             …
        ]
    },
2004 { username: "alice", … }]
```

To use this data, it first needs to be translated into an extensible record store structure where columns need to be arranged into families. Figure 3 shows one approach. Column family *_top* contains simple fields, *Personal* contains personal data and *Tweets[]* contains set of tweets. Now to perform a query, the system must navigate the tree structure of objects. For this translation example, queries are interpreted as

table/row/columnFamily/qualifier

For example, the query

```
get("users/155/personal/city")
```

would map *users* to table *users*, *personal* to column family *Personal*, and *city* to field *city*.

One strength of this approach is that the SOS code tends to be much shorter than the code that uses the native interface of the underlying data store. The downside of the approach is that it requires some programming knowledge to use it. The performance of SOS was measured on insertions and retrievals using HBase, Redis and MongoDB database systems. On all three systems the difference was almost negligible between issuing commands directly and through the SOS layer.

B. The general schema approach

In this section, the most important elements of data integration solution described in [7] are presented. This approach is based on the following:

- General schemas
- Database adapters
- Data migration
- Query processing

General schemas handle structural and semantic heterogeneities. They contain data model independent descriptions of semantic and structural information of the source database systems in the JSON format (like model generic representation of the SOS system previously described). General schemas are generated automatically and contain their name which comes from the database object (depending on the database type, this can be relation, view, collection or some other such type of object) and the following:

- Information about the source system (like the name of the database).
- Structural description of the database object (like attributes, their types, and keys of a table in a relational database system).
- Semantic mapping of the elements of the database object (creating aliases for database objects so they can be referred to by unique names).
- Relationships between database objects (necessary for connecting related data).

Figure 4 shows an example of a part of a simplified general schema for an address relation. CONNECTION and TYPE tags contain information about the database. The FIELDS tag contains the information about the fields of the relation where the *alias* tag is used for resolving semantic heterogeneities of the fields. Other tags are used for additional and/or optional information. OPTIONS tag contains information about the character encoding while the EMBEDDED tag is reserved for describing embedded documents in document-based database systems.

To access data stored in more than one database object (like table or view) a *connectivity graph* is created. This graph consists of nodes that represent data sources and edges that represent links (joins) between those data sources. Furthermore, each node contains one or more general schemas for that data source. To query data spanning multiple data sources the system needs to find a path in the graph that connects corresponding general schemas (i.e., the nodes in the graph). Figure 5 shows the structure of the relational schema. From that structure the following general schemas can be generated: ADDRESS, CUSTOMER, ORDER, FAVOURITE, and PRODUCT. Suppose we want to get the information about which cities customers who ordered certain products live in. Since this data is stored in two different database tables, ADDRESS and PRODUCT, to avoid having to do the cartesian product between the tables, we need to join the tables. If we look at the connectivity graph in Figure 6 there are two paths that lead from ADDRESS to PRODUCT tables: ADDRESS → CUSTOMER → FAVOURITE → PRODUCT and ADDRESS → CUSTOMER → ORDER → PRODUCT. In this case the user would have to select which path should be taken to query the tables.

Database adapters handle syntactic heterogeneities. Different database systems use different query languages, so a query needs to be translated into the language of the database system where the data is stored. In this approach database adapters are used. They get the query parameters from a function like the one in Figure 7. This function's name specifies the database operation it performs. The parameters specify the name of the general schema, query parameters and the query output parameters. Query parameters are represented as a JSON object. Database adapters translate such description of the query into a specific database system's language.

Data migration is the process of migrating data from different data sources. Since query results are structured as JSON objects (Figure 8) it is easy to merge all such results and migrate the data to different systems.

Query processing is performed using the following steps:

1. Paths containing the involved schemas are generated from the connectivity graph (with user's input if the path is ambiguous).
2. Parameters are translated into JSON format.
3. The adapters translate the query parameters into the native language of the specific data sources.
4. The resulting JSON objects representing the returned data are merged.

```
{
  "NAME"       : "sourcedb_address",
  "CONNECTION": "sourcedb",
  "TYPE"       : "mysql",
  "FIELDS"     : [{"name"   : "CustomerId",
                   "type"   : "text",
                   "length" : 11,
                   "alias"  : "Customer␣identity"
                  },
                  {"name"   : "Zip",
                   "type"   : "text",
                   "length" : 6,
                   "alias"  : "Zip␣Code"
                  },
                  {...}, {...}
```

Figure 4. A general schema example [7].

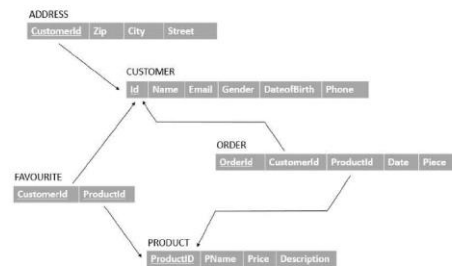

Figure 5. Relational schema for address data [7].

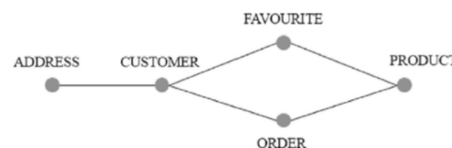

Figure 6. Connectivity graph for schema in Figure 5 [7].

```
dbExample.select(
    "Customer",
    {
       "PROJECTION": {
                       "Customer.Name"   : "Name␣of␣Customer",
                       "Customer.Email"  : "Email␣of␣Customer",
                       "Address.Zip"     : "Postcode",
                       "Address.City"    : "City"
                    },
       "CONDITION" : {
                       "Customer.Gender": { "$eq"   : "male" },
                       "Customer.Email" : { $regex : "mail.co$" },
                    },
       "JOIN"      :  [ "Customer", "Address" ],
       "ORDERBY"   :  { "Customer.Name" : 1 },
       "LIMIT"     :  0,
       "OFFSET"    :  0
    },
    function(error, data) {
                        if ( error ) {
                                     // error handling
                                    } else {
                                     // data handling
                                    }
                        }
)
```

Figure 7. An example of a general query function [7].

The results can then be used for further processing.

One advantage of this approach over the SOS system described previously is that it does not require programming knowledge and it provides data migration capabilities. The authors of [7] developed a web-based application that allows users to interactively work with the system.

Performance tests showed that generating a general query and translating it to the native database was very efficient. Most time is spent transferring the data from the database to the application's user interface and data conversion into JSON.

C. SQL++

Many NoSQL databases support semi-structured data, usually in the form of JSON. Those databases often have their own query language that supports the system's data model. Relational databases use a structured, schema-based data model based on relations. Although it is not a requirement, most relational database systems use SQL as a query language [1]. To bridge the gap between SQL and NoSQL databases and to utilize the existing SQL skills of many database programmers, a query language SQL++ [25] was developed. SQL++ is backward compatible with SQL and supports both relational and semi-structured data models. Its data model is a superset of relational data model and JSON.

```
[
        {
          "Name␣of␣customer"     : "Jonh␣Smith",
          "Email␣of␣customer"    : "john@mail.co",
          "Postcode"             : "W112BQ"
          "City"                 : "London"
        },
        {
          "Name␣of␣customer"     : "Danny␣Prett",
          "Email␣of␣customer"    : "danny@mail.co",
          "Postcode"             : "M2␣4WU"
          "City"                 : "Manchester"
        },
]
```

Figure 8. Query result in JSON format [7].

```
 1 sensors :: {
 2  location: 'Alpine',
 3  readings: [
 4   {
 5    time : timestamp('2014-03-12T20:00:00'),
 6    ozone: 0.035,
 7    no2  : 0.0050
 8   },
 9   {
10    time : timestamp('2014-03-12T22:00:00'),
11    ozone: {{ 0.03, {min: 0.03, max: 0.05}, 0.01 }},
12    co   : 0.4
13   } ]
14 }
```

Figure 9. Sample SQL++ value [25].

```
⟨readings:{{
  { no2: 0.6, co: 0.7,   co2: [0.5, 2] },
  { no2: 0.5, co: [0.4], co2: 1.3      } }} ⟩

FROM   readings AS r

SELECT ELEMENT (
 FROM   r AS {g:v}
 WHERE  g LIKE "co%"
 SELECT ATTRIBUTE g:v )

{{ { co: 0.7,   co2: [0.5, 2] },
   { co: [0.4], co2: 1.3      } }}
```

Figure 10. Sample SQL++ query [25].

Figure 9 shows a sample SQL++ value named *sensors* which contains data about sensor readings of environmental pollutants. As shown in the figure, the syntax of SQL++ values resembles that of JSON. Tuples are written inside curly braces. The tuple *sensors* contains two attributes: *location* and *readings*.

The *readings* attribute is an array of two tuples. We can see that these two tuples are heterogeneous: the *ozone* attribute in the first one is a number while in the second one it is a bag (unordered collection of elements which may contain duplicates). Additionally, the two tuples contain different sets of attributes: the first one contains *time*, *ozone* and *no2* while the second one contains *time*, *ozone* and *co*. A sample SQL++ query is shown in Figure 10.

Obviously, SQL++ bridges the gap between database heterogeneity by allowing access to elements of JSON data structures. Such data structures can represent relations of a relational data model, but also documents, rows, columns, or other data structures. There are two main disadvantages of SQL++ as a tool for data integration:

- It is an extension of SQL, so it requires knowledge of programming (therefore it is not intended for nonprogrammers).
- If used to access semi-structured databases, it requires understanding of their semi-structured data model.

Unlike the previous two approaches to data integration, SQL++ does not provide generic data model layer that allows users to work with data at a higher level of abstraction. On the other hand, SQL++ is not designed as just a data integration tool, but primarily as a query language that provides access to both relational and non-relational data sources.

D. Other approaches

Besides the three approaches described previously, here we briefly mention some other approaches. Unified Modelset Definition is described in [26]. It provides a framework for querying SQL and NoSQL database systems. In [27] a unified REST-based API is described. It can be used on both relational and nonrelational databases, but it does not offer data migration facilities. In [28] an extended SQL language is proposed to seamlessly query data in relational and document databases. In [29] a method of translating an SQL query referring to both relational and nonrelational data into NoSQL API is described. This was achieved by using a virtualization layer on top of nonrelational database system. The Partiqle system is described in [30]. It allows transactional SQL

queries over key-value databases. One of the more recent proposals is SQLtoKeyNoSQL [31], a layer for translation between SQL and key-oriented nonrelational database. In [32] a mapping language is defined where schema on which queries are defined corresponds to a relational data model, allowing queries to be specified in SQL, while the source systems can be column, key-value, or document stores.

V. CONCLUSION

This paper presented the main issues, principles, and approaches to data integration between heterogeneous databases. While many solutions are based on components or layers that translate data operations from a common, generic data model into the language of a particular database system, SQL++ is a query language extension solution that supports both relational and semi-structured data models. It remains to be seen what level of industry adoption each of these approaches will achieve over time as there is no supporting data at the time of this writing. Some other approaches to data integration not covered in this paper, such as ontology-based data integration and Datalog, will be the focus of our future work.

REFERENCES

[1] H. Garcia Molina, J. D. Ullman and J. Widom, Database Systems: The complete book, Pearson Prentice Hall, 2009.

[2] A. Messina, P. Storniolo and A. Urso, "Keep it simple, fast and scalable: a Multi-Model NoSQL DBMS as an (eb)XML-over-SOAP service," in *30th International Conference on Advanced Information Networking and Applications Workshops*, Palermo, Italy, 2016.

[3] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem and P. Helland, "The end of an architectural era (it's time for a complete rewrite)," *VLDB,* pp. 1150-1160, 2007.

[4] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann and S. Ubner, "Ontology-Based Integration of Information - A Survey of Existing Approaches," *Proceedings of the IJCAI'01 Workshop on Ontologies and Information Sharing, Seattle, Washington, USA, Aug 4-5,* 2002.

[5] I. Cruz and H. Xiao, "The Role of ontologies in data integration," *Journal of Engineering Intelligent Systems,* vol. 13, no. 12, 2005.

[6] R. Asgari, M. Moghadam, M. Mahdavi and A. Erfanian, "An ontology-based approach for integrating heterogeneous databases," *Open Computer Science,* vol. 5, no. 1, 2015.

[7] A. Vathy-Fogarassy and T. Hugyák, "Uniform data access platform for SQL and NoSQL database systems," *Information Systems,* vol. 69, 2017.

[8] S. Bradshaw, E. Brazil and K. Chodorow, MongoDB: The Definitive Guide, 3rd ed., O'Reilly Media, 2019.

[9] M. Ceruti and M. Kamel, "Semantic Heterogeneity in Database and Data Dictionary Integration for Command and Control Systems," Technical Report, DTIC Document, 1994.

[10] M. Bouzeghoub (ed.), "Semantics of a Networked World: Semantics for Grid Databases: First International IFIP Conference," in *ICSNW 2004, Springer*, 2004.

[11] A. P. Sheth, S. K. Gala and S. B. Navathe, "On automatic reasoning for schema integration," *Internat. J. Cooperative Inf. Syst.,* vol. 02, no. 01, pp. 23-50, 1993.

[12] K. R. Dittrich and D. Jonscher, "All together now: Towards integrating the world's information systems," C. Delgado, E. Marcos, J.M.M. Corral (Eds.), JISBD, 2000.

[13] P. Ziegler and K. R. Dittrich, "Three decades of data integration - all problems solved?," in *18th IFIP World Computer Congress (WCC 2004)*, Kluwer, 2004.

[14] J. Morris, Practical Data Migration, BCS Bookshop, 2012.

[15] N. F. Noy, "Semantic integration: a survey of ontology-based approaches," *SIGMOD,* vol. 33, no. 4, pp. 65-70, 2004.

[16] A. Hajmoosaei and S. Abdul-Kareem, "An ontology-based approach for resolving semantic schema conflicts in the extraction and integration of query-based information from heterogeneous web data sources," *Australian Computer Society,* vol. 85, 2007.

[17] D. M. Herzig, Ranking for web data search using on-the-fly data integration, KIT Scientific Publishing, 2014.

[18] S. Bhattacharjee and S. K. Ghosh, "Advanced computing, networking and informatics-Volume 1: advanced computing and informatics proceedings of the second international conference on advanced computing, networking and informatics (ICACNI-2014)," *Springer International Publishing,* 2014.

[19] T. Zhao, C. Zhang, M. Wei and Z.-R. Peng, "Ontology-based geospatial data query and integration," in *Proceedings of the 5th International Conference on Geographic Information Science, GIScience '08, Springer-Verlag*, Berlin, Heidelberg, 2008370–392, 2008.

[20] A. Malucelli, D. Palzer and E. Oliveira, "Ontology-based services to help solving the heterogeneity problem in e-commerce negotiations," *Electron. Commer. Res. Appl.,* vol. 5, no. 1, pp. 29-43, 2006.

[21] J. Zhang, G.-m. Zhou, J. Wang, J. Zhang and F. Xie, "Computer and computing technologies in agriculture VII: 7th IFIP WG 5.14 international conference," in *CCTA 2013*, Beijing, China, 2013.

[22] P. Geibel, M. Trautwein, H. Erdur, L. Zimmermann, K. Jegzentis, M. Bengner, C. H. Nolte and T. Tolxdorff, "Ontology-based information extraction: identifying eligible patients for clinical trials in neurology," *J. Data Semant.,* vol. 4, no. 2, pp. 133-147, 2015.

[23] S. Mate, F. Köpcke, D. Toddenroth, M. Martin, H. Prokosch, T. Bürkle and T. Ganslandt, "An ontology-based approach for integrating heterogeneousdatabases," *Open Comput. Sci.,* vol. 5, no. 1, pp. 41-50, 2015.

[24] P. Atzeni, F. Bugiotti and L. Rossi, "Uniform access to nosql systems," *Inf. Syst.,* vol. 43, pp. 117-133, 2014.

[25] K. W. Ong, Y. Papakonstantinou and R. Vernoux, "The SQL++ semi-structured data model and query language: a capabilities survey of sql-on-hadoop, nosql and newsql databases," *CoRR abs/1405.3631,* 2014.

[26] W. Allen, "https://www.infoq.com/articles/unified-data-modeling-for-relational-and-nosql-databases," 2016. [Online].

[27] R. Sellami, S. Bhiri and B. Defude, "Odbapi: a unified rest api for relational and nosql data stores," in *IEEE International Congress on Big Data, IEEE, 2014653–660*, 2014.

[28] J. Roijackers and G. H. Fletcher, "On bridging relational and document-centric data stores," *Big Data, Springer, 2013135–148.*

[29] R. Lawrence, "Integration and virtualization of relational sql and nosql systems including mysql and mongodb," *Computational Science and Computational Intelligence (CSCI), 2014 International Conference on, 1, IEEE, 2014285–290,* 2014.

[30] J. Tatemura, O. Po, W.-P. Hsiung and H. Hacigümüş, "Partiqle: an elastic sql engine over key-value stores," *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012629–632,* 2012.

[31] G. A. Schreiner, D. Duarte and R. dos Santos Mello, "Sqltokeynosql: a layer for relational to key-based nosql database mapping," *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services, ACM, 201574.*

[32] O. Curé, R. Hecht, C. Le Duc and M. Lamolle, "Data integration over nosql stores using access path based mappings," *Database and Expert Systems Applications, Springer, 2011481–495.*