Open Access

Open Journal of Semantic Web (OJSW)
Volume 7, Issue 1, 2020

http://www.ronpub.com/ojsw
ISSN 2199-336X

NextGen Multi-Model Databases in Semantic Big Data Architectures

Irena Holubová[A], Stefanie Scherzinger[B]

[A]Department of Software Engineering, Charles University, Malostranske nam. 25, 118 00 Praha 1, Czech Republic, holubova@ksi.mff.cuni.cz

[B]OTH Regensburg, Prüfeninger Straße 58, 93049 Regensburg, Germany, stefanie.scherzinger@oth-regensburg.de

Abstract

When semantic big data is managed in commercial settings, with time, the need may arise to integrate and interlink records from various data sources. In this vision paper, we discuss the potential of a new generation of multi-model database systems as data backends in such settings. Discussing a specific example scenario, we show how this family of database systems allows for agile and flexible schema management. We also identify open research challenges in generating sound triple-views from data stored in interlinked models, as a basis for SPARQL querying. We then conclude with a general overview of multi-model data management systems, to provide a wider scope of the problem domain.

Type of Paper and Keywords

Visionary Paper: *semantic data management, schema evolution, data architecture, data integration, schema management, multi-model DBMS.*

1 Introduction

Ideally, an enterprise information system (EIS) provides a 360^o view on corporate data. However, tapping new data sources usually involves long-running and costly data integration projects. One reason is that the underlying data backend is commonly a relational database management system: Evolving the relational database schema in production systems is a real-world challenge [3, 4].

In this paper, we view the task of building an enterprise information system as a semantic big data project, where we want to query a triple view of the data using SPARQL, and that we can evolve over time. Ideally, we can flexibly integrate new data sources with little impedance overhead.

The research communities focusing on semantic data and on database architecture have been building powerful triple stores for managing RDF data [43, 1, 29, 33]. However, *native* triple stores may not be suitable for *big data* scenarios, due to the up front costs of converting data from its original format (most likely, not triples). Ideally, a new data source can be ingested *as is*, with little data integration overhead. Moreover, the raw triple format is often perceived as unwieldy for certain data types, such as geospatial data, the integration of which is becoming ever more important [30].

Non-native triple stores with SPARQL endpoints [9, 35, 32] keep the data in its original format, most commonly, as relations. Along these lines, manifold contributions on publishing triple-views over relational, XML, or JSON data, have been made [26, 17], even to the point of reaching the status of W3C recommendations for relational data [6, 11] and CSV files [39], or being discussed by a W3C working group,

such as the JSON data format [25].

The popular, industrial-strength and non-native triple store Virtuoso[1] supports relational, XML, and the triple format. By now, Virtuoso looks back on over 20 years of development. Built on top of a traditional relational database management system (RDBMS), like many databases of its generation, it is *schema-full*, meaning that the schema must be declared before a single record can be stored. The database system then manages the schema in its internal catalog, and ensures that all read and write accesses are valid.

Yet in enterprise settings, large volumes of data accumulate and new data sources are added over time. As in other application domains, the database schema is no longer something that we can fix in the early phase of a project. Instead, we may have to repeatedly integrate new data sources, and therefore to evolve the database schema.

We summarize our desiderata for a database system at the heart of our IT architecture as follows:

1. In handling big data, we cannot afford to pre-process and translate each record into RDF triples. Ideally, new data can be ingested *as-is*, with little or no impedance overhead.

2. New data sources will be added over time and structural changes to the data are inevitable. Thus, we are not able to fix a stable schema in the early phases of the project.

3. The data is not static; thus, we need a backend where records may be updated, rather than an append-only data warehouse.

4. We require built-in support for popular data formats, e.g., for managing geospatial data.

5. We need to be able to expose a homogenized triple-view, as a 360^o view on the entire data instance, allowing for SPARQL querying.

In this paper, we put an alternative backend technology up for discussion, and assess the potential of a new generation of database systems that can handle several data models, and where the schema is often managed flexibly.

Of course, the idea of supporting several data models is not new, and there are over 20 representatives of multi-model databases (MM-DBs) [23], including well-known products such as Oracle DB[2] or IBM DB2[3]. Yet there are new players in this market, such as OrientDB[4] or ArangoDB[5]. Although there is no exact definition of a multi-model database, the intuitive understanding of this term assumes the support of *several* data models as *first-class citizens* with efficient support of respective storing and querying, allowing both structured data (such as key/value and graph data) and semi-structured data (XML and JSON). Thus, these systems can seamlessly integrate document collections, social network graphs, or ontologies, which may be interlinked. (This in turn requires sophisticated multi-model transaction management, query evaluation, query optimization, etc.). At the same time, they allow for more flexible schema management, where some are even schema-free, as we will illustrate.

With this paper, we would like to invite the semantic big data community to explore these *NextGen multi-model databases* [22].

Contributions: This vision paper gives an overview over a new generation of multi-model databases, in particular focusing on flexible schema management. Walking through a running example, we demonstrate basic capabilities and motivate research questions. Our overview can be useful for researchers looking for new research opportunities in the field of semantic big data.

The paper is an extended version of the SBD@SIGMOD 2019 workshop paper [15]. The main extensions involve: a new and more complex running example with a focus on ontology evolution; further, a general classification and description of approaches to multi-model data management, and a more detailed discussion of the challenges related to semantic web.

Structure: In Section 2 we characterize the family of NextGen multi-model databases, in particular regarding different levels of schema support. In Section 3, we envision an example of a semantic big data project evolving over time, backed by a NextGen multi-model database. Section 4 discusses research challenges in this context. To ensure a broader view of the target problem domain, in Section 5 we provide a general overview of approaches to multi-model data management. With Section 6, we conclude.

2 Database Schemas in NextGen MM-DBs

In general, the term multi-model can have different meanings in the context of database management

[1] https://virtuoso.openlinksw.com/
[2] https://www.oracle.com/database/
[3] http://www.ibm.com/analytics/us/en/technology/db2/
[4] https://orientdb.com/
[5] https://www.arangodb.com/

systems.[6] We give an overview over the landscape of polyglot persistence and multi-model databases in Section 5. For now, we refer to *NextGen MM-DBs* to denote a particular family of systems.

Since this area is relatively new, there is no rigorous definition. However, we can characterize these systems as ***DBMSs which support more than one data model, where all these models are first-class citizens and can be mutually interlinked, and which support cross-model query evaluation***.

We next clarify our core terminology to avoid confusion when switching between related or even synonymous terms:

- By a *record*, we mean a single data entity that is to be persisted. In the relational model, this is a tuple. In the document model, this is a single document. In the graph model, this is a vertex or an edge.
- By a *kind*, we mean an abstract label that groups related records. In a relational database, this corresponds to a table. Some multi-model databases use the term *class* (as in OrientDB) or *collection* (as in ArangoDB) instead. In the graph model, there are only two kinds, vertices and edges.
- By a *property*, we refer to an attribute in a relational tuple, or in a JSON document. In the graph model, properties may be assigned to vertices or edges.

We assume that the records of a given kind all reside in the same model. This assumption holds in virtually all established multi-model database products today.

While these terms do not sufficiently describe the various data models in their entirety, they establish a common ground for the following discussion.

With multi-model databases, we can distinguish different levels of schema support on the granularity of a single *kind*:

- A *schema-full* kind requires that the properties of all corresponding records are valid w.r.t. the declared schema.
- If a kind is declared as *schema-less* (or *schema-free*), the system does not validate the corresponding records against the schema.[7]
- If a kind is *schema-mixed* (also called *schema-hybrid*), additional properties (not declared by the schema) are allowed.

[6] Note that there also exists the term *multi-modal* which does not mean a combination of multiple data models, but multiple data modalities, e.g., audio, video, eye gaze data, etc.

[7] In theory, no two records of the same kind might even have the same structure. In practice, however, it is likely that there will be some form of agreed structure among records [18].

In the following, we name examples to illustrate the different levels of schema support.

Example 2.1. Relational database systems are traditionally schema-full. □

Example 2.2. The popular (originally NoSQL document, now multi-model) database MongoDB[8] used to be schema-less. Due to a validation feature added later, MongoDB now supports schema-mixed kinds. This provides developers with more type-safety, while still allowing for a certain degree of flexibility in modeling their data. □

Next, we consider two prominent multi-model databases w.r.t. their level of schema support, as well as their strategy for handling several models.

Example 2.3. ArangoDB is a schema-less multi-model database[9], and supports both a graph and a document model. Yet for optimized storage, the graph model is transparently mapped onto an internal document model: Nodes are stored as documents, and edges are stored in a special document collection containing pairs of IDs of the documents corresponding to nodes. Thus internally, the two supported logical models (graphs and documents) are mapped onto the same physical (document) model. □

Example 2.4. OrientDB supports a document, graph, key/value and a designated object model. Internally, all models but the graph model are effectively reduced to the object-model. OrientDB provides all three levels of schema support. □

3 Example Scenario

We envision a semantic big data project where we build an enterprise information system for a pizza delivery franchise. We consider the multi-model database OrientDB as our data backend, since it offers all three levels of schema support. In parts, we contrast this choice of backend with Virtuoso [13], a popular schema-full (non-native) triple store.

3.1 Growing a Pizza Empire

In the early stage of our project, we need to manage our pizza recipes. Over time, we will tap new data sources and grow our data hub.

[8] `https://www.mongodb.com/`

[9] While schema-less, ArangoDB internally tracks the structure of all records and exploits structural similarities to reduce storage costs [18].

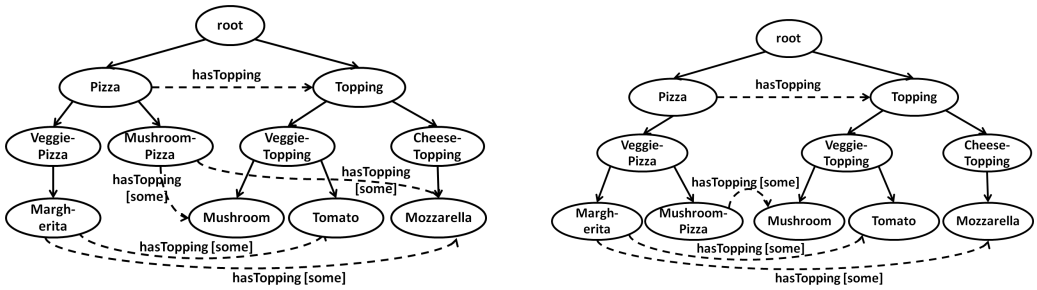


Figure 1: Ontology evolution in the pizza example, taken from [19], with permission by the authors.

```
1  CREATE CLASS Topping;
2  CREATE PROPERTY Topping.name STRING (MANDATORY TRUE, NOTNULL TRUE);
3  CREATE INDEX Topping.name UNIQUE;
4
5  CREATE CLASS VeggieTopping EXTENDS Topping;
6  CREATE CLASS Mushroom      EXTENDS VeggieTopping;
7  CREATE CLASS Tomato        EXTENDS VeggieTopping;
8
9  CREATE CLASS CheeseTopping EXTENDS Topping;
10 CREATE CLASS Mozzarella    EXTENDS CheeseTopping;
11
12 CREATE CLASS Pizza;
13 CREATE PROPERTY Pizza.name STRING (MANDATORY TRUE, NOTNULL TRUE);
14 CREATE INDEX Pizza.name UNIQUE;
15
16 CREATE CLASS VeggiePizza   EXTENDS Pizza;
17 CREATE CLASS Margherita    EXTENDS VeggiePizza;
18 CREATE CLASS MushroomPizza EXTENDS Pizza;
19
20 CREATE PROPERTY Pizza.hasTopping LINKSET Topping (MANDATORY TRUE);
21
22 CREATE PROPERTY Margherita.hasTTopping   LINK Tomato     (MANDATORY TRUE, NOTNULL TRUE);
23 CREATE PROPERTY Margherita.hasMozTopping LINK Mozzarella (MANDATORY TRUE, NOTNULL TRUE);
24
25 CREATE PROPERTY MushroomPizza.hasMTopping   LINK Mushroom   (MANDATORY TRUE, NOTNULL TRUE);
26 CREATE PROPERTY MushroomPizza.hasMozTopping LINK Mozzarella (MANDATORY TRUE, NOTNULL TRUE);
```

Figure 2: Encoding the pizza ontology in the OrientDB schema.

@rid	@class	name
#65:0	TOPPING	seafood
#66:0	TOPPING	ham
#73:0	VEGGIETOPPING	tomato sauce
#75:0	VEGGIETOPPING	oregano
#76:0	VEGGIETOPPING	garlic
#77:0	VEGGIETOPPING	basil
#89:0	TOMATO	tomato slice
#97:0	MUSHROOM	mushroom
#105:0	MOZZARELLA	mozzarella
#106:0	MOZZARELLA	buffalo mozzarella

(a) Toppings.

@rid	@class	name	hasTopping	hasMoz Topping	hasT Topping	hasM Topping
#43:0	VEGGIEPIZZA	bufalina	[#73:0, #75:0, #106:0]			
#44:0	VEGGIEPIZZA	caprese	[#89:0, #105:0]			
#50:0	MUSHROOMPIZZA	prosciutto et funghi	[#66:0,#73:0,#105:0,#97:0]	#105:0		#97:0
#65:0	PIZZA	frutti di mare	[#65:0, #73:0]			
#73:0	VEGGIEPIZZA	marinara	[#73:0, #76:0, #77:0]			
#75:0	MARGHERITA	margherita	[#73:0, #75:0, #89:0, #105:0]	#105:0	#89:0	

(b) Pizzas.

Figure 3: Instances of toppings and pizzas from our example scenario.

3.1.1 Encoding the pizza ontology

Our pizza recipes are at the heart of our business. Figure 1, on the left, shows the popular pizza ontology, taken from [19]. We need to represent the classes `Pizza` and `Topping`, with their various subclasses. The dashed arrow `hasTopping` denotes a relationship between classes, so each pizza can have toppings. There are further constraints, as each pizza margherita must have (at least) some `Tomato` and (at least) some `Mozzarella` topping.

Figure 2 shows the statements in the data definition language (DDL) of OrientDB that encode this ontology as an OrientDB schema. These statements were designed for OrientDB version 3.0.21 (where 3.0.x is the current GA version). We first declare classes (akin to *kinds*) for toppings and pizzas respectively.

In declaring the class hierarchy, we encounter little to no impedance overhead, as OrientDB started out as an object database management system and thus gracefully handles class hierarchies (even multiple inheritance).

A unique record identifier is assigned and managed by OrientDB automatically, replacing the concept of primary keys. Nevertheless, we would like to declare a property `name` as a secondary key: Line 2 declares that the name is mandatory and therefore must exist. In principle, a property may be mandatory and nullable. Yet in our case, we rule out this case. Thus, for each topping, a name must be specified, *null* is not allowed. Both toppings and pizzas are actually identified by their names. Declaring the indices (e.g., in line 3) ensures that names are unique.

Before we remark on `LINK`-typed class properties (c.f. line 20), we consider a specific data instance.[10] The syntax for insertion is familiar from SQL, e.g.,

```
INSERT INTO Mozzarella (name)
VALUES ("mozzarella"),
       ("buffalo mozzarella");
```

To encode the relationship `hasTopping`, we declare OrientDB properties to hold a set of links to `Toppings`. This property must exist, but its value may be an empty set or even *null*.

For margherita and mushroom pizzas, we additionally link to the required ingredients and specify that these links must be specified (and thus not *null*, c.f. the lines 22 and following).

Figures 3a and 3b show instances of toppings and pizzas (in a relational view): Each instance has a unique record identifier `@rid` (which is internally maintained), and belongs to a class (`@class`). Pizzas link to their toppings. Margherita-style pizzas need one link to a mozzarella- and a tomato-topping (of which there are different instances, such as plain and buffalo mozzarella).

[10] Our pizza compositions originate from `https://www.forketers.com/italian-pizza-names-list/`.

name	hasTopping.name
bufalina	["oregano","buffalo mozzarella","tomato sauce"]
frutti di mare	["tomato sauce","seafood"]
marinara	["basil","garlic","tomato sauce"]
margherita	["oregano","tomato sauce","mozzarella", "tomato slice"]
caprese	["tomato slice","mozzarella"]
prosciutto e funghi	["tomato sauce","mozzarella","ham", "mushroom"]

Figure 4: Result of query `SELECT name, toppings.name FROM Pizza` on the pizza instances from Figure 3b.

Links are automatically resolved during query evaluation. For instance, the SQL-like query

```
SELECT name, hasTopping.name
FROM Pizza;
```

returns the result shown in Figure 4, and thus resolves the links to `Toppings`.

3.1.2 Ontology Evolution

We now consider the case that the ontology evolves, replaying the scenario proposed in [19]. In the changed version, shown in Figure 1 to the right, mushroom pizzas have become veggie pizzas. This schema change can be directly expressed in the OrientDB DDL as follows:

```
ALTER CLASS MushroomPizza
SUPERCLASS VeggiePizza;
```

We assume that accidentally, for some reason, the restriction that mushroom pizzas must have at least some mozzarella topping, has been lost, as visualized in Figure 1 (right).

The authors in [19] also propose regression tests to detect problems with evolutionary changes:
t_1: There are at least 3 veggie pizzas.
t_2: Pizza must include mushroom pizza.
t_3: Mushroom pizza has mozzarella topping.
t_4: There must be at least 1 veggie topping.

These tests can be conveniently expressed in the SQL dialect of OrientDB. In our example, all tests pass. Let us consider test t_1, which can be addressed by the query

```
SELECT count(*)
FROM VeggiePizza;
```

counting the number of veggie pizzas (as suggested by [19]). Test t_1 passes, since there are still more than three veggie pizzas. However, the count now yields a different result, since the mushroom pizza "prosciutto et funghi" is included. This observation might trigger

the developers to inspect their data instance, hopefully realizing that this pizza should not have been classified as vegetarian.

Test t_3 yields a false positive, since the schema no longer enforces that mushroom pizzas come with a mozzarella topping. Pizza "prosciutto e funghi" just happens to have a mozzarella topping.

However, these problems are imminent to ontology evolution, and not specific to the underlying database management system.

3.1.3 Generating Triple-Views

The state-of-the-art today in (non-native) triple stores such as Virtuoso is to store data in a relational backend, to avoid expensive data conversions. On demand, we can then generate a triple-view, as sketched in Figure 5 for pizza "bufalina", as a basis for SPARQL querying.[11] (Due to constraints in visualizing the views, we recode the record identifiers, e.g., `#43:0` as "rid43".)

We depict classes as blue ovals, instances as purple ovals, and literals as orange boxes.

Generating triple-views from relational data is well-explored, and several W3C recommendations exist, e.g., [6, 11]. In contrast, NextGen multi-model databases do not fully provide this functionality yet. In the current GA version of OrientDB, yet also in the latest beta-only version (v3.1.0), OrientDB supports Tinkerpop and Gremlin[12], so SPARQL queries[13] may be compiled to the Gremlin graph traversal language[14]. However, we can currently only query data residing in the graph model. Data in other models (like our pizzas and their toppings) cannot be queried with Gremlin or SPARQL.

Yet exposing a triple-view is an indispensable feature from the viewpoint of the semantic big data community, which is why we propose adding this feature to NextGen multi-model databases in Section 4. However, this is more than just a mere engineering problem; as we will discuss, the triple-views need to be generated from *interlinked* data models, which is a nontrivial challenge.

We continue discussing our scenario and tap a new data source, containing relational data.

3.2 Ingesting Relational Data

We next ingest customer data from a relational database, as shown in Figure 6a. Each customer is identified by a customer id. We further know the customer's name and credit limit.

With Virtuoso as our backend, we would declare a relational schema for relation *Customer*. With OrientDB, we manage customer records within a (flat) OrientDB class with mandatory properties, as shown in Figure 7. We specify that the customer ID must not be *null* (line 3). Setting the `STRICTMODE` (line 9) declares this class to be schema-full, so additional properties cannot be added to records.

Again, the syntax for adding cusomer records is straightforward for developers already familiar with SQL:

```
INSERT INTO Customer (CID, CName, CLimit)
VALUES (1, 'Mary', 5000),
       (2, 'John', 3000),
       (3, 'Anne', 2000);
```

Figure 6b shows a triple-view of our three customers. Again, we believe that generating triple-views is a desirable feature for NextGen multi-model databases like OrientDB.

3.3 Managing Graph Data

Later, we purchase amendatory data from a social network provider, to find out which of our customers know each other. Figure 6c shows the social network graph G, where the vertices are labeled with customer identifiers, and edges (labeled "knows") capture when a customer knows another customer.

When working with a NextGen multi-model database that supports the graph model, we can import this data with ease. The DDL statements are shown in Figure 8. First, we register all customers as vertices in the generic class of vertices V (by declaring the customer class a subclass of V). Then, we create a new edge class *knows* (inheriting from the generic edge class E), and add the two instances declaring that Anne knows Mary, and Mary knows John.[15]

A sweet spot here is that we may issue queries across edges; to identify the names of customers who know John, we write

```
SELECT CName
FROM (SELECT EXPAND(IN())
      FROM Customer WHERE CName = 'John');
```

which returns Mary, as John has an incoming edge from Mary in the social network graph.

Naturally, we will want to expose this graph data as a triple-view as well, as sketched in Figure 6d. For graph data, OrientDB Studio already provides a generic graph visualization. Also, OrientDB provides means to

[11] The triple-view visualizations shown here were generated with the tool `https://github.com/usc-isi-i2/ontology-visualization`, in the version from Oct. 2018.

[12] `http://tinkerpop.apache.org/`

[13] To be precise, only a subset of SPARQL 1.0.

[14] `http://tinkerpop.apache.org/docs/current/reference/#sparql-gremlin`

[15] Not shown: To add edges, we need to temporarily disable the `STRICTMODE` declared on class `Customer`.

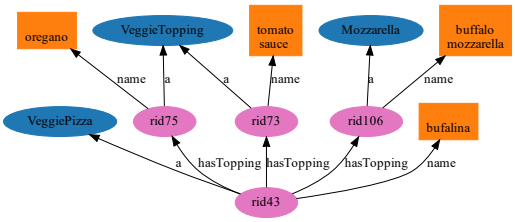
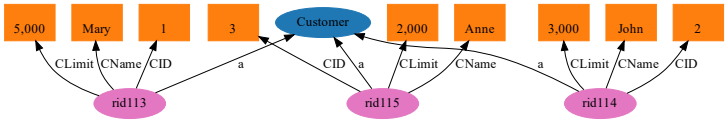


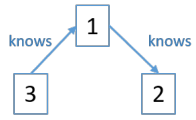
Figure 5: Triple-view for pizza "bufalina" with the internal record identifier `#43:0` (c.f. "rid43").

CID	CName	CLimit
1	Mary	5,000
2	John	3,000
3	Anne	2,000

(a) Relation *Customer*.



(b) Triple-view of customers Mary, Anne, and John.



(c) Social graph G.

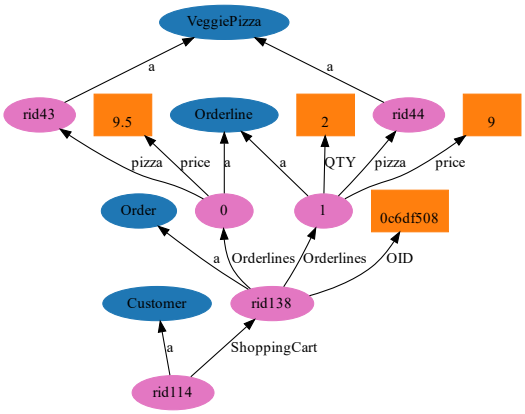
(d) Triple-view of the social graph.

key	value
1	34e5e759
2	0c6df508

(e) Key/value *ShoppingCart*.

```
{ "OID": "0c6df508",
  "Orderlines": [
    { "pizza": "bufalina",
      "Price": 9.5 },
    { "pizza": "caprese",
      "price": 9,
      "QTY": 2} ] }
```

(f) Document *Order*.



(g) Triple-view of John's shopping cart.

Figure 6: Customer-related data in several linked data models and their triple-views.

```
1  CREATE CLASS Customer;
2
3  CREATE PROPERTY Customer.CID INTEGER    (MANDATORY TRUE, NOTNULL TRUE);
4  CREATE INDEX Customer.CID UNIQUE;
5
6  CREATE PROPERTY Customer.CName  STRING  (MANDATORY TRUE);
7  CREATE PROPERTY Customer.CLimit INTEGER (MANDATORY TRUE);
8
9  ALTER CLASS Customer STRICTMODE TRUE;
```

Figure 7: Encoding relational customer data in the OrientDB schema.

```
1  ALTER CLASS Customer SUPERCLASS V;
2  CREATE CLASS knows EXTENDS E;
3
4  CREATE EDGE knows
5    FROM (SELECT FROM Customer WHERE CID = 3) TO (SELECT FROM Customer WHERE CID = 1);
6  CREATE EDGE knows
7    FROM (SELECT FROM Customer WHERE CID = 1) TO (SELECT FROM Customer WHERE CID = 2);
```

Figure 8: Encoding a social network graph in the OrientDB schema.

query graphs (but not records from other data models) via Gremlin and SPARQL.

3.4 Managing Document Data

From our web shop, we next integrate data on the customers' shopping carts and orders. Figure 6f shows a JSON document with John's order.

Each order is identified by its order ID, and consists of an array of orderlines. An orderline, in turn, specifies the pizza ordered, and the price for this item. Note that the first orderline does not state the quantity explicitly, rather, we assume by default that a single pizza bufalina has been ordered. The second orderline explicitly states a quantity. The key/value mappings in Figure 6e associate the customers with their shopping cart.

In OrientDB, the schema for order documents can be declared as nested OrientDB classes, as shown in Figure 9.

Since these classes are not declared as strict, order documents need at least the specified properties, but additional properties are allowed (such as the quantity of products in the second orderline). Thus, orders are schema-mixed, and we can insert our JSON document from Figure 6f, as shown below. This can be done near-verbatim, with only a minor adaption, specifying the class and type in lines 4, 5 and 8, 9.

```
1  INSERT INTO Order SET
2    OID = "0c6df508",
3    Orderlines = [
4      { "@type":"d",
5        "@class":"Orderline",
6        "pizza": #43:0,
7        "price": 9.5 },
8      { "@type":"d",
9        "@class":"Orderline",
10       "pizza": #44:0,
11       "price": 9,
12       "QTY": 2} ];
```

Note that in lines 6 and 10, we hard-coded the record identifiers of pizza bufalina and caprese.

Again, OrientDB can resolve links during query evaluation. So even though *Order* documents only store links to pizzas, we can nevertheless access the pizza names. The following query produces the names of all pizzas ever ordered.

```
SELECT DISTINCT Orderlines.pizza.name
FROM Order;
```

In Figure 10, we implement the key/value mappings from the shopping cart. Now, John can place his order:

```
INSERT INTO ShoppingCart SET
  key   = (SELECT @rid FROM Customer
           WHERE  CName = "John"),
  value = (SELECT @rid FROM Order
           WHERE  OID   = "0c6df508");
```

When OrientDB resolves links, this is a powerful alternative to joins and allows for compact queries:

```
SELECT value.OID
FROM ShoppingCart
WHERE key.CName = 'John';
```

Note that John's name is not a property of the shopping cart, but of the linked customer record.

Again, we want to be able to generate a triple-view across all data models, upon the push of a button. Figure 6g shows how we envision this for customer John's shopping cart and order. Several algorithms for publishing triple data from NoSQL data models (such as XML and JSON documents) [26, 17] have been published, yet they are not integrated in NextGen multi-model databases, where we have the particular challenge of considering links: Ideally, we'd like the key-value encoding of the shopping cart to be realized as an OWL object property (rather than an instance), linking one instance to another (as depicted in the triple-view).

```
1  CREATE CLASS Orderline;
2  CREATE PROPERTY Orderline.pizza LINK Pizza (MANDATORY TRUE);
3  CREATE PROPERTY Orderline.price DOUBLE (MANDATORY TRUE);
4
5  CREATE CLASS Order;
6  CREATE PROPERTY Order.OID STRING;
7  CREATE PROPERTY Order.Orderlines EMBEDDEDLIST Orderline;
```

Figure 9: Encoding order documents in the OrientDB schema.

```
1  CREATE CLASS ShoppingCart;
2
3  CREATE PROPERTY ShoppingCart.key   LINK Customer (MANDATORY TRUE, NOTNULL TRUE);
4  CREATE PROPERTY ShoppingCart.value LINK Order    (MANDATORY TRUE, NOTNULL TRUE);
```

Figure 10: Encoding the shopping cart mappings in the OrientDB schema.

3.5 Managing Geospatial Data

The amount of linked open data with an inherent spatial context is increasing. However, using SPARQL to query specific geospatial relationships (e.g., finding objects within a particular distance from a given point) is not generally perceived as elegant or easy. In consequence, a number of related vocabularies have been proposed, as well as query languages strengthened by the OGC standard GeoSPARQL [30].

Let us imagine that we further integrate the parking opportunities for bicycles from the New York City open data collection[16]. This information will help us in planning our pizza deliveries. Figure 11 shows an excerpt of this data, derived from a CSV file. Each parking site has a unique site id, and is located in a borough. Its location is described by a house number and street name, as well as by latitude and longitude. There are different asset types (commonly, bike racks).

With its GeoSpatial Module, OrientDB has dedicated support for geospatial points, lines, or polygons. In Figure 12, we declare the OrientDB schema for the bicycle parking data. In line 11, we aggregate the values for latitude and longitude as a geospatial point, which is natively supported in OrientDB. To insert the first record from the CSV file in OrientDB, we write:

```
INSERT INTO Parking SET
  SITE_ID = 18941,
  Borough = "Brooklyn",
  House   = 15,
  Street_Nam = "LAFAYETTE AV",
  Asset_Type = "Bike Rack",
  Location = {"@class": "OPoint",
              "coordinates": [-73.97851,
                              40.68679]};
```

As OrientDB supports geospatial queries, we may identify suitable delivery routes, given our customers' delivery addresses.

[16] https://data.cityofnewyork.us/Transportation/Bicycle-Parking/yh4a-g3fj

3.6 Evolving the Schema

As user requirements change, we may need to add a new, optional property to one of the models (e.g., a delivery address to the orders). Such an *intra-model* schema change is restricted to a single model:

```
CREATE PROPERTY Orderline.Address STRING;
```

Naturally, this change will have to be reflected in the triple-view, producing new nodes and edges (i.e., a *monotonic evolution*).

Now, let us perform an *inter-model* schema change to merge the `ShoppingCart` key/value mappings with `Customers`. The DDL statements are shown in Figure 13[17].

While this change affects two data models, it should (ideally) not affect the triple-views shown in Figure 6g. After all, while the schema of the data has changed, its semantics has not. Thus, we aim at a new level of *logical data independence*, where semantics-preserving schema changes at the level of the logical database schema should not affect the generated triple-view.

3.7 Summary

Throughout this chapter, we have grown our pizza empire over time, adapting the OrientDB schema along the way. In the final version, we can now ask new queries, thanks to integrating data from different sources, and inter-linking records. For instance, we can identify customers who have only ordered vegetarian pizzas in the past, or identify groups of friends who are vegetarians. This can help us target our advertising campaigns.

Already today, these queries may be formulated in the SQL-dialect of OrientDB. Along our vision of generating triple-views, we hope to be able to formulate these queries in SPARQL in the future. This is particularly appealing, as we would have a uniform view

[17] Note that the `UPDATE` command should also be applied to all other key/value records.

SITE_ID	Borough	House	Street_Nam	Asset_Type	Latitude	Longitude
18941	Brooklyn	15	LAFAYETTE AV	Bike Rack	40.68679	-73.97851
18658	Brooklyn	24	4 AV	Bike Rack	40.684012	-73.978633
18995	Brooklyn	65	LAFAYETTE AV	Bike Rack	40.687072	-73.975805
19421	Brooklyn	104	BEDFORD AV	Bike Rack	40.720247	-73.955151

Figure 11: Excerpt of NYC bicycle parking data, in tabular view.

```
CREATE CLASS Parking;

CREATE PROPERTY Parking.SITE_ID INTEGER (MANDATORY TRUE);
CREATE INDEX Parking.SITE_ID UNIQUE;

CREATE PROPERTY Parking.Borough STRING;
CREATE PROPERTY Parking.House INTEGER;
CREATE PROPERTY Parking.Street_Nam STRING;
CREATE PROPERTY Parking.Asset_Type STRING;

CREATE PROPERTY Parking.Location EMBEDDED OPoint;

ALTER CLASS Parking STRICTMODE TRUE;
```

Figure 12: Encoding bicycle parking data in the OrientDB schema.

of the data, even though each record actually resides in the data model that is closest to its raw and original form.

In the Introduction, we listed five desiderata. Summarizing our observations from our example scenario, already today, NextGen multi-model databases can meet desiderata (1) through (4). However, their support for triple-views is limited. Again, we consider this our biggest open research challenge in the next section.

4 Research Challenges

In this paper, we argue that NextGen multi-model databases are an interesting architectural choice for building scalable non-native triple stores. We next describe key challenges that we believe must be mastered for these systems to be successful in semantic big data scenarios.

4.1 Triple-Views on Multi-Model Data

For the semantic big data community, being able to expose a triple-view of the data stored, and to evaluate SPARQL queries, is a must.

In OrientDB (currently GA version 3.0.x), SPARQL queries are compiled first to the Gremlin graph API, and then executed. However, this is restricted to data that resides in the graph model. Rather, we require algorithms to compute triple-views on linked multi-model data. While there are various approaches for generating triple-views from single-model data (*either* relational [34] *or* JSON [12] *or* XML [17], ...), we are not aware of solutions that work for several data models with interlinked records.

When handling big data, things will not be as easy as merely blending the existing algorithms. We will need to deliver highly scalable solutions. For instance, when generating triple-views, we may build upon existing work on summarizing ontologies, or digests, c.f. [40]. These summaries capture the essence of the knowledge graph and allow for high-level browsing. Likewise, a tool which will help users in the whole process of publishing their multi-model data as Linked Data (such as the ETL tool LinkedPipes [20]) might be an important contribution to the problem domain.

Also, compiling SPARQL queries directly to the native, database-supported query languages is likely to boost performance. For instance, OrientDB resorts to MapReduce processing when in distributed mode, a functionality that we might leverage for SPARQL evaluation.

Last but not least, the correctness of triple-views must be ensured under conflicting requirements that are model-specific. For example, while the relational model in traditional relational databases is closely associated with strong consistency, NoSQL data stores, such as document or key/value, often only implement eventual consistency. Moreover, in the relational model we try to avoid redundancy in data and therefore normalize the schema. On the other hand, typical optimization strategies of distributed NoSQL systems are to introduce redundancy or materialized views, to name two.

4.2 Linked Multi-Model Data and Ontologies

In Section 3, we sketched how the pizza ontology might be implemented as an OrientDB schema. However, in this simplistic example we ignored more advanced

```
ALTER CLASS Customer STRICTMODE FALSE;

CREATE PROPERTY Customer.ShoppingCart LINK Order;

UPDATE Customer SET
  ShoppingCart = (SELECT @rid FROM Order WHERE OID = "0c6df508")
  WHERE CName = "John";

DROP CLASS ShoppingCart;
```

Figure 13: An inter-model schema change.

features of ontologies, such as sub-properties or further cardinality restrictions. As with relational databases, most MM-DBs have only limited support for expressing complex schema constraints. Inevitably, some constraints imposed by the ontology will have to be enforced by the application logic instead. In general, the mapping between multi-model schemas and OWL2 profiles [27] needs to be defined. However, it first requires a formal definition of multi-model schema or constraints, which remains an important challenge of multi-model data, as discussed next.

At the same time, apart from the complex and challenging idea of combining distinct models, the key aspect of multi-model databases are links between distinct models. In single-model systems, the links can have different representations, such as key/foreign key relationships in the relational model, references (pointers) in the object model, embedding/references of the document model or edges in the graph model. If we mix models and their specific notions of links, we get a number of combinations which have so far not been investigated or standardized. Assigning proper semantics to inter-model links is another important challenge that has not yet been addressed. We assume that like in single-model systems, linking will probably be carried out semi-automatically, assisted by suitable tools, such as [5].

4.3 Supporting Ontology Evolution

A database schema declares more than record properties, it also enforces integrity constraints. This is of particular interest when ontologies evolve (near-inevitable in long-running projects), as they need to remain consistent under updates. When it comes to big data, consistency checks need to scale to large volumes of data. It is generally acknowledged that an efficient way to enforce ontology consistence is via schema-declared integrity constraints [42]. For instance, in OrientDB, we may work with indices to enforce the uniqueness of values.

As the need for such constraints may only materialize over time, ontology evolution is related to database schema evolution. At the same time, schema evolution in NextGen multi-model databases has not yet been systematically explored, as discussed next.

Even in schema-less DBMSs (which is the case of, e.g., most of the originally NoSQL systems listed in Table 1), there is typically an "intrinsic" schema, i.e., a kind of agreed structure of data that is expected by the application. When user requirements change, this affects not only the structure of the data, but also all related parts of the system (data instances, integrity constraints, queries, storage strategies etc.). Consequently, the mappings of data in an evolving schema to RDF triples have to account for this. On the other hand, if the schema change does not change the semantics of the data, the triple-views should not be affected either (yet the query plans for evaluating SPARQL queries on the multi-model data will).

A number of papers deals with schema evolution in single-model systems (e.g., relational or XML [28], or aggregate-oriented NoSQL [36]). But apart from a first academic prototype [41] and a recent position paper [14], there are no principled tools supporting schema evolution in multi-model databases in its full complexity. However, the community is presently devising benchmarks for multi-model databases that take schema evolution into account, right from the start [21]. The availability of such benchmarks is vital for evaluating competing solutions w.r.t. the research challenges discussed here.

In addition, carrying out schema changes in a transactionally safe manner in a distributed system is both a research and an engineering challenge. E.g., the Google-internal database F1 [31] scalably implements this for the relational model.

4.4 General Challenges

In discussing our sample scenario, we have carefully avoided certain problems in data integration that are known to be difficult. For instance, in integrating data from different sources, we face the entity resolution problem. This active research area involves techniques for record linkage and deduplication. We refer to a recent survey on entity resolution in big data processing, where we generally face data variety in terms of different data models [10]. We have also glossed over the challenge of mapping and matching existing schemas that use homonymous and synonymous terms (c.f. [8]

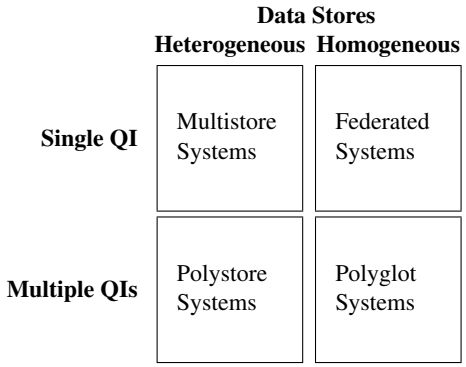


Figure 14: Multi-DBMS distinguished by (1) number of query interfaces (QIs) and (2) the underlying specialized data stores.

for a holistic overview).

Apart from challenges common to data integration tasks, or directly related to building non-native triple stores, there are also general challenges with NextGen multi-model databases. These challenges concern system maturity. On the one hand, multi-model systems involve both traditional relational DBMSs with a long history, as well as newer, but already well established NoSQL systems [22]. On the other hand, the level of support for multiple data models in these systems strongly differs and does not correspond to their general robustness. In addition, there are currently no recognized best practices, or even standards, for modeling, querying, updating, etc. of multi-model data [24]. Similarly, the process of building a strong theoretical foundation for multi-model data management is in its early stages. From the user point of view, we have encountered a number of cases where the documentation does not clearly describe the expected behaviour of the system, or any of the more advanced features. Thus, certain NextGen multi-model database products have not yet reached an appropriate level of applied as well as theoretical maturity.

5 Multi-Model Data Management

To provide a broader scope of the problem domain, we outline existing strategies for multi-model data management. In general, there are two existing approaches to manipulate and query multi-model data [22], namely *multi-DBMSs* systems and *single-DBMSs* systems, both discussed next.

5.1 Multi-DBMSs Systems

The main idea of polyglot persistence is to combine different specialized DBMSs, each with a distinct (native) data model, query language and other capabilities using a middle-ware layer. As defined in [38], the data stores can be either homogeneous or heterogeneous, whereas heterogeneity can be specified at the level of data stores (having different modeling techniques and physical architectures), processing engines (having different processing capabilities when built around arrays, graphs, dictionaries etc.), and/or query interfaces (having various formal algebras and expressive powers). The same paper also classifies existing solutions as listed below, and as visualized in Figure 14.

Federated systems were thoroughly researched during the 1980s and 1990s. They consist of multiple homogeneous data stores and a single query interface. The main strategy is to develop a middleware (called *mediator*) to integrate together multiple, usually relational databases (e.g., Multibase [16] defines a global schema, a mapping language, and a local-to-host translator, whereas users pose the queries against the global schema).

Polyglot systems usually address the need to manage complex data flows in distributed file systems, where data processing can be specified as declarative queries, but also as procedural algorithms. In general, they consist of multiple homogeneous data stores and multiple query interfaces (e.g., Spark SQL [7] provides an API with both relational and procedural access mode).

Multistore systems consist of multiple heterogeneous data stores, including HDFS, RDBMS and NoSQL databases, and one query interface (e.g., HadoopDB [2] integrates a distributed file system with a relational database).

Last but not least, *polystore systems* are built on top of multiple heterogeneous data storage engines, involving relational, array, stream, and key/value stores, generally represented as *islands of information*. Users can choose from a number of query interfaces to process data stored in a variety of data stores.

5.2 Single-DBMSs Systems

Single-DBMSs systems, i.e., the key target of this paper, are usually rather denoted as multi-model databases. They manage different data models within a single, fully integrated backend, to handle the system demands for performance, scalability, and fault tolerance [24]. The idea can be traced back to object-relational database management systems, which extend towards the object-oriented programming model for relational databases, and which can thus store and process various formats, such as relational, text, XML, spatial and object, leveraging domain-specific functions.

Currently, there exist more than 20 representatives of

Table 1: Summary of key features of multi-model databases.

Type	DBMS	Ext.	Models	Query languages
Relational	PostgreSQL[a]	I	R-KJX--O	extended SQL
	Microsoft SQL Server[b]	I	R--JXG-O	extended SQL
	IBM DB2[c]	I	R---XGDO	extended SQL/XML
	Oracle DB[d]	I	R--JX-DO	SQL/XML or JSON extension of SQL
	MySQL[e]	II	R-K----O	SQL, memcached API
	Sinew [37]	III	R-K-----	SQL
Column	Cassandra[f]	I	-C---G-O	SQL-like CQL (Cassandra Query Language)
	CrateDB[g]	I	RC-J-G--	SQL
	DynamoDB[h]	I	-CKJ-G-O	simple API (get/put/update) + simple queries over indices
	Vertica[i]	II	-C-J-G--	SQL-like
Key/value	Riak KV[j]	I	--KJXG--	Solr
	c-treeACE[k]	III	R-K--G--	SQL
	Oracle NoSQL DB[l]	III	R-K--GD-	SQL
Document	Cosmos DB[m]	I	-CKJ----	SQL-like
	ArangoDB[n]	II	--KJ-G--	SQL-like AQL (ArangoDB Query Language)
	MongoDB[o]	II	--KJ---O	JSON-based query language
	Couchbase[p]	III	--KJ----	SQL-based N_1QL (Couchbase query language "nickel")
	MarkLogic[q]	III	---JX-DO	XPath, XQuery, SQL-like
Graph	OrientDB[r]	II	--KJ-G--	Gremlin, extended SQL, SPARQL
Object	InterSystems Caché[s]	III	R--JX--O	SQL with object extensions

[a] https://www.postgresql.org/
[b] http://www.microsoft.com/en-us/server-cloud/products/sql-server/
[c] http://www.ibm.com/analytics/us/en/technology/db2/
[d] https://www.oracle.com/database/index.html
[e] https://www.oracle.com/mysql/index.html
[f] http://cassandra.apache.org/
[g] https://crate.io/
[h] https://aws.amazon.com/dynamodb/
[i] https://www.vertica.com/
[j] https://riak.com/products/riak-kv/
[k] https://www.faircom.com/products/c-treeace
[l] https://www.oracle.com/database/technologies/related/nosql.html
[m] http://www.cosmosdb.com
[n] https://www.arangodb.com/
[o] https://www.mongodb.com/
[p] http://www.couchbase.com/
[q] https://www.marklogic.com/
[r] https://orientdb.com/
[s] https://www.intersystems.com/cz/products/cache/

Legend: I = adoption of a new storage strategy, II = extension of the original storage strategy, III = creation of a new interface, IV = no change; R = relational, C = column, K = key/value, J = JSON, X = XML, G = graph, D = RDF, O = object.

multi-model databases, involving well-known tools from both the traditional relational and novel NoSQL systems. As portrayed in a recent extensive survey [22], they have distinct features and can be classified according to various criteria. The core difference is the strategy used to extend the original model to other models or to combine multiple models. The new models can be supported via (I) adoption of an entirely new storage strategy, (II) extension of the original storage strategy, (III) creation of a new interface, or even (IV) no change in the original storage strategy (which is used for trivial cases).

In Table 1, partially borrowed from [22], we capture prominent systems by their key features, classified according to the original or core model (i.e., relational, column, etc.). It includes a reference to a web page or a core paper devoted to the system, the strategy for multi-model extension, as well as supported models and query languages.

For example, even though currently both ArangoDB and OrientDB support the same set of models (i.e., key/value, document, namely JSON, and graph), they belong to different groups with regards to the original model: ArangoDB started with the document model which was extended towards graphs using a special edge collection. OrientDB was originally a graph database which was soon extended to support documents, thanks to its object-oriented features allowing to define

hierarchies. So in both cases we can say that the original storage strategy was only extended towards the new models. Like most systems, both support an SQL-like query language, usually with proprietary extensions.

6 Conclusion

With this paper, we hope to entice the semantic big data community to consider NextGen multi-model databases as backends for non-native triple stores that can scale to big data. We believe that this new technology is a major stepping point towards unlocking enterprise data, building 360^o views on data otherwise locked away in data silos.

Acknowledgements. The authors are supported by the Czech Science Foundation (GAČR) project number 19-01641S (I. Holubová) and the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), grant number #38580880 (S. Scherzinger). We thank Martin Nečaský and Haridimos Kondylakis for their comments on an earlier version of this paper.

References

[1] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, "A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2049–2060, Sep. 2017.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, Aug. 2009.

[3] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, 1st ed. Wiley Publishing, 2003.

[4] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[5] Y. An, J. Mylopoulos, and A. Borgida, "Building Semantic Mappings from Databases to Ontologies," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, ser. AAAI '06, 2006, pp. 1557–1560.

[6] M. Arenas, A. Bertails, E. Prud'hommeaux, and J. Sequeda, "A Direct Mapping of Relational Data to RDF," W3C, W3C Recommendation, 2012, http://www.w3.org/TR/rdb-direct-mapping/.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, 2015, pp. 1383–1394.

[8] Z. Bellahsene, A. Bonifati, and E. Rahm, *Schema Matching and Mapping*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[9] M. Chaloupka and M. Nečaský, "Efficient SPARQL to SQL Translation with User Defined Mapping," in *Proceedings of the 7th International Conference on Knowledge Engineering and Semantic Web*, ser. KESW '16, 2016, pp. 215–229.

[10] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis, "End-to-end entity resolution for big data: A survey," *CoRR*, vol. abs/1905.06397, 2019. [Online]. Available: http://arxiv.org/abs/1905.06397

[11] S. Das, S. Sundara, and R. Cyganiak, "R2RML: RDB to RDF Mapping Language," W3C, W3C Recommendation, 2012, http://www.w3.org/TR/r2rml/.

[12] A. Dimou, M. V. Sande, J. Slepicka, P. Szekely, E. Mannens, C. Knoblock, and R. V. d. Walle, "Mapping Hierarchical Sources into RDF Using the RML Mapping Language," in *Proceedings of the 2014 IEEE International Conference on Semantic Computing*, ser. ICSC '14, 2014, pp. 151–158.

[13] O. Erling and I. Mikhailov, "RDF Support in the Virtuoso DBMS," in *Networked Knowledge - Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems*, T. Pellegrini, S. Auer, K. Tochtermann, and S. Schaffert, Eds. Springer Berlin Heidelberg, 2009, pp. 7–24.

[14] I. Holubová, M. Klettke, and U. Störl, "Evolution Management of Multi-Model Data," in *Proceedings of the VLDB Workshop on Polystores That Manage Multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data*, ser. Poly '19, 2019.

[15] I. Holubová and S. Scherzinger, "Unlocking the Potential of NextGen Multi-model Databases for Semantic Big Data Projects," in *Proceedings of the International Workshop on Semantic Big Data*, ser. SBD '19, 2019, pp. 6:1–6:6.

[16] J. Huang, "MultiBase: a Heterogeneous Multidatabase Management System," in

Proceedings Eighteenth Annual International Computer Software and Applications Conference, ser. SOMPSAC '94, 1994, pp. 332–339.

[17] J.-Y. Huang, C. Lange, and S. Auer, "Streaming Transformation of XML to RDF Using XPath-based Mappings," in *Proceedings of the 11th International Conference on Semantic Systems*, ser. SEMANTICS '15, 2015, pp. 129–136.

[18] J@ArangoDB. (2014) Schema Handling in ArangoDB. [Online]. Available: http://jsteemann.github.io/blog/2014/06/03/schema-handling-in-arangodb/

[19] M. Kim, J. Cobb, M. J. Harrold, T. Kurc, A. Orso, J. Saltz, A. Post, K. Malhotra, and S. B. Navathe, "Efficient regression testing of ontology-driven systems," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA '12, 2012, pp. 320–330.

[20] J. Klímek and P. Škoda, "LinkedPipes ETL in Use: Practical Publication and Consumption of Linked Data," in *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*, ser. iiWAS '17, 2017, pp. 441–445.

[21] J. Lu, "Towards Benchmarking Multi-Model Databases," in *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*, ser. CIDR '17, 2017.

[22] J. Lu and I. Holubová, "Multi-Model Databases: A New Journey to Handle the Variety of Data," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 55:1–55:38, Jun. 2019.

[23] J. Lu, I. Holubová, and B. Cautis, "Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges," in *Proc. CIKM '18*, 2018, pp. 2301–2302.

[24] J. Lu, Z. H. Liu, P. Xu, and C. Zhang, "UDBMS: Road to Unification for Multi-model Data Management," in *Advances in Conceptual Modeling*, C. Woo, J. Lu, Z. Li, T. W. Ling, G. Li, and M. L. Lee, Eds. Springer International Publishing, 2018, pp. 285–294.

[25] W3C JSON-LD Working Group. (2019) JSON for Linking Data. Https://json-ld.org/.

[26] F. Michel, L. Djimenou, C. Faron Zucker, and J. Montagnat, "xR2RML: Relational and Non-Relational Databases to RDF Mapping Language," CNRS, Research Report ISRN I3S/RR 2014-04-FR, Oct. 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01066663

[27] B. Motik, "Owl 2 web ontology language profiles (second edition)," W3C, W3C Recommendation, 2012, https://www.w3.org/TR/owl2-profiles/.

[28] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková, "Evolution and Change Management of XML-based Systems," *J. Syst. Softw.*, vol. 85, no. 3, pp. 683–707, Mar. 2012.

[29] M. T. Özsu, "A Survey of RDF Data Management Systems," *Front. Comput. Sci.*, vol. 10, no. 3, pp. 418–432, Jun. 2016.

[30] M. Perry and J. Herringi, "GeoSPARQL - A Geographic Query Language for RDF Data," OGC, Open Geospatial Consortium, 2012, https://www.opengeospatial.org/standards/geosparql.

[31] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek, "Online, Asynchronous Schema Change in F1," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1045–1056, Aug. 2013.

[32] M. Rodriguez-Muro and M. Rezk, "Efficient SPARQL-to-SQL with R2RML mappings," *J. Web Semant.*, vol. 33, pp. 141–169, 2015.

[33] S. Sakr and E. Pardede, *Graph Data Management: Techniques and Applications*, 1st ed. Information Science Reference - Imprint of: IGI Publishing, 2011.

[34] J. F. Sequeda, M. Arenas, and D. P. Miranker, "On Directly Mapping Relational Databases to RDF and OWL," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12, 2012, pp. 649–658.

[35] J. F. Sequeda and D. P. Miranker, "Ultrawrap: SPARQL execution on relational data," *J. Web Semant.*, vol. 22, pp. 19–39, 2013.

[36] U. Störl, D. Müller, M. Klettke, and S. Scherzinger, "Enabling Efficient Agile Software Development of NoSQL-backed Applications," in *Proceedings of Datenbanksysteme für Business, Technologie und Web*, ser. BTW '17, 2017, pp. 611–614.

[37] D. Tahara, T. Diamond, and D. J. Abadi, "Sinew: A SQL System for Multi-structured Data," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, 2014, pp. 815–826.

[38] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson, "Enabling Query Processing Across Heterogeneous Data Models: a Survey," in *Proceedings of the 2017 IEEE International Conference on Big Data*, ser. BigData '17, 2017, pp. 3211–3220.

[39] J. Tennison, "CSV on the Web: A Primer," W3C, W3C Recommendation, 2016, http://www.w3.org/TR/tabular-data-primer/.

[40] G. Troullinou, H. Kondylakis, E. Daskalaki, and D. Plexousakis, "RDF Digest: Efficient Summarization of RDF/S KBs," in *Proceedings of the 12th European Semantic Web Conference on The Semantic Web. Latest Advances and New Domains*, ser. ESWC '15, 2015, pp. 119–134.

[41] M. Vavrek, I. Holubová, and S. Scherzinger, "MM-evolver: A Multi-model Evolution Management Tool," in *Proceedings of the 22nd International Conference on Extending Database Technology*, ser. EDBT '19, 2019, pp. 586–589.

[42] J. Wu and F. Lecue, "Towards Consistency Checking over Evolving Ontologies," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, ser. CIKM '14, 2014, pp. 909–918.

[43] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr, "RDF Data Storage and Query Processing Schemes: A Survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 84:1–84:36, Sep. 2018.

AUTHOR BIOGRAPHIES

Irena Holubová is an Associate Professor at the Charles University, Prague, Czech Republic. Her current main research interests include Big Data management and NoSQL databases, evolution and change management of database applications, analysis of real-world data, and schema inference. She has published more than 80 conference and journal papers; her works gained 4 awards. She has published 2 books on XML and NoSQL databases.

Stefanie Scherzinger is a professor at OTH Regensburg, Germany. Her research is strongly influenced by her industry experience as a former software engineer at IBM and Google. Currently, she focuses on maintaining applications backed by NoSQL data stores, and systematic support for database schema evolution.