Managing Polyglot Systems Metadata with Hypergraphs

Moditha Hewasinghage[1], Jovan Varga[1], Alberto Abelló[1], Esteban Zimányi[2]

[1] Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain
{moditha,jvarga,aabello}@essi.upc.edu
[2] Université Libre de Bruxelles, 1050 Bruxelles, Belgium
ezimanyi@ulb.ac.be

Abstract. A single type of data store can hardly fulfill every end-user requirements in the NoSQL world. Therefore, polyglot systems use different types of NoSQL datastores in combination. However, the heterogeneity of the data storage models makes managing the metadata a complex task in such systems, with only a handful of research carried out to address this. In this paper, we propose a hypergraph-based approach for representing the catalog of metadata in a polyglot system. Taking an existing common programming interface to NoSQL systems, we extend and formalize it as hypergraphs for managing metadata. Then, we define design constraints and query transformation rules for three representative data store types. Furthermore, we propose a simple query rewriting algorithm using the catalog itself for these data store types and provide a prototype implementation. Finally, we show the feasibility of our approach on a use case of an existing polyglot system.

Keywords: Metadata Management, NoSQL, Polystore

1 Introduction

With the dawn of the big data era, the heterogeneity among the data storage models has expanded drastically, mainly due to the introduction of NoSQL. There are four primary data store models in NoSQL systems: (i) Key-value stores, which perform like a typical hashmap, where the data is stored and retrieved through a key and an associated value; (ii) Wide-column stores, that manage the data in a columnar fashion; (iii) Document stores, represent data in a document like structure, which can become increasingly complex with nested elements; (iv) Graph stores, are instance based and store the relationships between those instances. The heterogeneity is not only limited to the data models, but also various implementations of the same data model can be entirely different from one to another due to the lack of a standard.

Heterogeneous systems can be useful in different scenarios, because it is highly unlikely that a single data store can efficiently handle all the requirements of the end-user. Therefore, it is common to use different ones to manage different portions of the data. This allows to control the storage and retrieval

more efficiently for different requirements. Hence, polyglot systems were introduced, similar to traditional Federated Database Systems (FDBMS), but more complex considering the need to handle semistructured data models. Due to the heterogeneity at different levels, most of the work on polyglot systems [4, 7] suggests the implementation of wrappers or interfaces for each participating data store. However, this becomes more complex as the number of participating data store types grows.

The catalog (see [9]) maintains the meta information of the data store. Having one for a polyglot system enables end users to have a clear view of the complex system. Its metadata plays a significant role in understanding the overall picture of the underlying infrastructure and, as well, helps to improve the design of the polyglot system and determine the access patterns needed for different query requirements. It is essential to answer questions such as: What is the structure of the data being stored? Where is a piece of data stored? Is it duplicated in another store? What is the best way to retrieve this data? Nevertheless, little research addresses the managing of metadata in polyglot systems. This is mainly due to the lack of a design construct that can represent heterogeneous, semistructured data. In this paper, we address the metadata management in polyglot systems by extending an already existing NoSQL design method [3, 2] and formalizing the constructs through hypergraphs.

The SOS Model [2] claims to capture the NoSQL modeling structures in data design for key-value stores, document stores, and wide-column stores utilizing three main constructs: attributes, structs, and sets. These constructs and their interactions allow to represent the physical storage of above NoSQL systems. The fact that the model is simple makes it compelling in representability, but the lack of formalization leaves space for ambiguity and hinders the automation of metadata management in such settings. Instead, it is simply used as a common programming interface for data exchange.

In this paper, we formalize SOS using a hypergraph-based representation, and extend it to maintain the catalog that manages the metadata of a polyglot system. RDF is considered to be able to represent any kind of data and is often used as a data interchange format. Therefore, we make the assumption that we have exemplars of the data in the polyglot system in RDF. Then, we build a hypergraph that maps to different data design constructs, representing the SOS model over the information. We represent the catalog of the polyglot system using these constructs, and introduce a simple query generation algorithm to show the usefulness of our approach. Next, we explore different data store models, identify their design constructs, introduce their design constraints, and define query generation rules for each of them. Finally, we show the feasibility of our approach using a use case of an existing polyglot system by representing its metadata catalog through our constructs.

The simple, yet powerful hypergraph-based approach presented in this paper is a step towards representing heterogeneous, semistructured data in a formal manner as well as managing the corresponding metadata of a polyglot system. It proves to be useful concerning (i) expressiveness: the ability to express different

representations, regardless of their complexity and (ii) semantic relativism: the ability to accommodate different representations of the same data, as defined in [18].

This paper is organized as follows: First, in Section 2, we introduce some background. We present and formalize our data model in Section 3. Afterwards, we discuss the managing of the metadata through the model in Section 4. Then, we present an application of the constructs in Section 5. Finally, we introduce the related works in Section 6, and conclude our work with future work in Section 7.

2 Preliminaries

In this section, we introduce the basic concepts about RDF [14] and SOS Model [2, 3] that will be used in our approach.

2.1 Resource Description Framework (RDF)

The Resource Description Framework [14] is a Wide Web Consortium (W3C) specification for representing information on the Web. It is a graph-based data model that helps to serve information by making statements about available resources.

RDF represents data as triplets consisting of subject, predicate, and object (s, p, o). These can be resources that are identified by an Internationalized Resource Identifier (IRI), which is a unique Unicode string within the RDF graph. An object can also be a literal, which is a data value. An example of RDF is shown in Listing 2.1, written in Turtle notation. The example contains information about music albums, artists, and songs and is used throughout the paper to illustrate our approach.

```
@prefix foaf: <http://xmlns.com/foaf/0.1> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://dbtune.org/jamendo/artist/dylan> rdf:type mo:MusicArtist;
  foaf:name "Bob Dylan"^^xsd:string;
  foaf:made <http://dbtune.org/jamendo/record/emp>.

<http://dbtune.org/jamendo/record/emp> rdf:type mo:Record;
  dc:title "Empire Burlesque"^^xsd:string;
  foaf:maker <http://dbtune.org/jamendo/artist/dylan> ;
  mo:track <http://dbtune.org/jamendo/track/Seeing>, <http://dbtune.org/
       jamendo/track/Tight> .

<http://dbtune.org/jamendo/track/Tight> rdf:type mo:Track;
  dc:title "Tight Connection to My Heart"^^xsd:string .
<http://dbtune.org/jamendo/track/Seeing> rdf:type mo:Track;
  dc:title "Seeing the Real You at Last"^^xsd:string .
```

Listing 2.1: Example RDF dataset [1]

2.2 SOS Model

The high flexibility of NoSQL systems gives the freedom to have multiple designs for the same data. A particular data design built focusing on a specific scenario

[1] http://dbtune.org

can result in adverse performance when applied in a different context. Most of the data design for NoSQL is carried out based on concrete guidelines for different datastores and access patterns. Nevertheless, recent approaches propose generic design constructs for NoSQL systems. For our approach, we decided to use the SOS model [2, 3] as a starting point.

The SOS model introduces a basic common model (or a meta-layer) which is a high-level description of the data models of non-relational systems. This model helps to handle the vast heterogeneity of the NoSQL datastores and provides interoperability among them, easing the development process. The primary objective of the meta-layer is to generalize the data model of heterogeneous NoSQL systems. Thus, it allows standard development practices on a predefined set of generic constructs. The meta-layer reconciles the descriptive elements of key-value stores, document stores, and record stores. The different data models exposed by NoSQL datastores are effectively managed in the SOS data model with three major constructs: *Attribute*, *Struct*, and *Set* [3].

A name and an associated value characterize each of these constructs. The structure of the value depends on the type of construct. An *Attribute* can contain a simple value such as an Integer or String. *Struct*s and *Set*s are complex elements which can contain multiple *Attribute*s, *Struct*s, *Set*s or a combination of those. SOS Model mainly addresses data design on document stores, key-value stores, and wide-column stores [2]. Each of the datastore instances is represented as a set of collections. There can be any arbitrary number of *Set*s depending on the use case. Simple elements such as key-value pairs or single qualifiers can be modeled as *Attribute*s and groups of *Attribute*s, or a simple entity such as a document, can be represented as a *Struct*. A collection of entities is represented in a *Set*, which can be a nested collection in a document store or a column family in a wide-column store.

3 Formalization

In this section, we introduce and formalize our data model, which is based on representative exemplars in RDF format of each kind of instances in the underlying data stores. This RDF graph contains the classes and user-defined types of the polyglot system. Previous work has shown that this global schema can be obtained by extracting the schema of each data store and reconciling them [8, 20].

Building on top of the RDF data model discussed in Section 2.1, we introduce our design constructs based on the SOS Model. Figure 1 shows the overall class diagram of our constructs, where thicker lines represent the elements already available in the SOS (namely *Set*s, *Struct*s, and *Attribute*s), and the relationship multiplicities defined in SOS are preserved. On top of that, we introduce additional constructs to aid the formalization process and manage metadata. From here on, we use letters in blackboard font to represent sets of elements (e.g, $\mathbb{A} = \{A_1, A_2 ... A_n\}$).

We rely on the concept of hypergraph, which is a graph where an edge (aka hyperedge) can relate any number of elements (not only two). This can be fur-

ther generalized so that hyperedges can also contain other hyperedges (not only nodes).

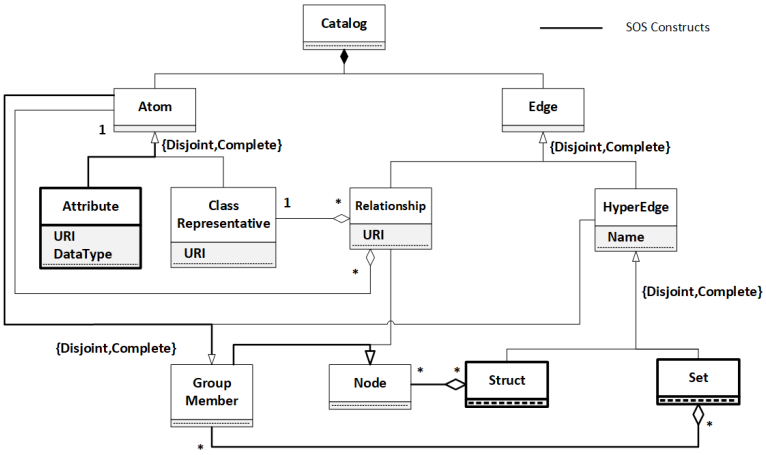


Fig. 1: Class diagram for the overall catalog

We define the overall polyglot system catalog as composed by the schema and the essential elements that support a uniquely accessible terminology for the polyglot system.

Definition 1. *A polyglot catalog* $C = \langle \mathbb{A}, \mathbb{E} \rangle$ *is a generalized hypergraph where* $\mathbb{A}$ *is a set of atoms and* $\mathbb{E}$ *is a set of edges.*

Definition 2. *The set of all atoms* $\mathbb{A}$ *is composed of two disjoint subsets of class atoms* $\mathbb{A}_C$ *and attribute atoms* $\mathbb{A}_A$*. Formally:* $\mathbb{A} = \mathbb{A}_C \cup \mathbb{A}_A$

Atoms are the smallest constituent unit of the graph and carry a name. Moreover, every A_C contains a URI that represents the class semantics, while every A_A carries the datatype and a URI for the user-defined type semantics.

Definition 3. *The set of all edges* $\mathbb{E}$ *composed of two disjoints subsets of relationships* $\mathbb{E}_R$ *that denote the connectivity between* $\mathbb{A}$*, and hyperedges* $\mathbb{E}_H$ *that denotes connectivity between other constructs of* C*. Formally:* $\mathbb{E} = \mathbb{E}_R \cup \mathbb{E}_H$

Definition 4. *A relationship* $E_R^{x,y}$ *is a binary edge between two atoms* A_x *and* A_y *and a URI* u *that represents the semantics of* E_R*. At least one of the atoms in the relationship must be an* A_C*. Formally:* $E_R^{x,y} = \langle A^x, A^y, u \rangle | A^x, A^y \in \mathbb{A} \wedge (A^x \in \mathbb{A}_C \vee A^y \in \mathbb{A}_C)$

This graph $G = \langle \mathbb{A}, \mathbb{E}_R \rangle$ is a representation of the available data, i.e., an RDF translation of the original representatives of the data contained in the polyglot system, that we assume to be given. This G immutable as it contains the knowledge about the data. Figure 2 shows the graph G of the original RDF example in Listing 2.1.

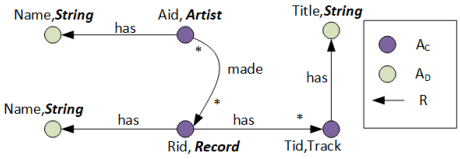


Fig. 2: Translated graph built from the RDF

We build our data design on top of G, based on the constructs introduced in SOS model. Thus, we make use of the *Hyperedge*s in G and give rise to our hypergraph-based catalog C. An incidenceSet of an *Atom* or a *Hyperedge* contains the immediate set of E that *Atom* or *Hyperedge* is part of, respectively.

Definition 5. *The transitive closure of an edge E is denoted as E^+, where* $E \in E^+,\ \forall e \in E^+ : e \in e'.incidenceSet \implies e' \in E^+$

Definition 6. *A hyperedge E_H is a subset of atoms $\mathbb{A}$ and edges $\mathbb{E}$ and it cannot be transitively contained in itself. Formally :* $E_H \subseteq \mathbb{A} \cup \mathbb{E} \wedge E_H.incidenceSet \cap E_H^+ = \emptyset$

Definition 7. *A struct E_{Struct} is a hyperedge that contains a set of atoms $\mathbb{A}$, relationships $\mathbb{E}_\mathbb{R}$, and/or hyperedges $\mathbb{E}_\mathbb{H}$. All atoms within E^+_{Struct} must be connected by a set of E_R that also belong to E^+_{Struct}. Formally :* $E_{Struct} \subseteq \mathbb{E}_H \cup \mathbb{A} \cup \mathbb{E}_R | \forall A^x, A^y \in E^+_{Struct} : \exists \{E_R^{x,x_1}, E_R^{x_1,x_2}, .., E_R^{x_n,y}\} \in \mathbb{E}^+_{Struct}$

Definition 8. *A set E_{Set} is a hyperedge that contains a set of arbitrary hyperedges $\mathbb{E}_\mathbb{H}$ or/and atoms $\mathbb{A}$. Formally :* $E_{Set} \subseteq \mathbb{E}_H \cup \mathbb{A}$

For the ease of illustration, we have also used two additional constructs in Figure 1: A group member $M \in \mathbb{A} \cup \mathbb{E}_H$ is an element of an E_{Set}. In this context, it is an atom or a hyperedge; A node $N \in \mathbb{M} \cup \mathbb{E}_R$ is an element of an E_{Struct}, which is either an M or an E_R.

4 Metadata Management

One crucial aspect of a metadata management system is the ability to represent different data store models. In our work, we exemplify it on traditional RDBMS, document stores, and wide-column stores. Figure 3 extends our original diagram of constructs to support those.

The E_{Set} are specialized into two types: *Data Store* E_D and *First Level* E_F. E_D represents a data store instance of the polyglot system. E_F denotes a collection of instances in the particular data store. All the allowed kinds of data stores are a subclass of E_D. Moreover, $E_D.incidenceSet = \emptyset$.

Thus, we define three E_D (namely *Relational* E_D^{Rel}, *Document Store* E_D^{Doc}, and *Wide-Column* E_D^{Col} in Figure 3), which are the participants of our polyglot system. There can be multiple E_F within each E_D adhering to the number of collections that participate in the polyglot system.

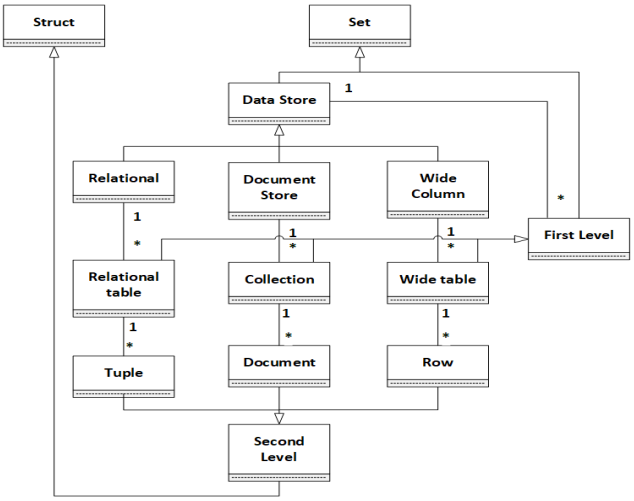


Fig. 3: Class diagram for metadata with *Hyperedge* composition

All the *Atom*s and *Edge*s of the polyglot system belong to the transitive closure of one or more of these E_D, $\mathbb{A} \cup \mathbb{E} = \bigcup E_D^+$. Therefore, we can deduce that the entire polyglot system catalog C can be represented by the participating E_D .

The *Second Level* (E_S), is a *Struct* that represents a kind of object residing directly in E_F. These E_S should align with the type of E_D where it is contained. It is a tuple for E_S^{Rel}, a document stored directly in the collection for E_S^{Doc}, and a row for E_S^{Col}.

Each E_H carries a name which is interpreted depending on the context. In E_D, it can represent the physical location of the underlying data store. In E_F, it can be the collection name or table name. Depending on the type of E_D that represents the data store, we can identify specific constraints and transformation rules for the queries over the representatives.

The *Edge*s of the catalog can carry much more information than just the name. For example, an E_R can indicate the multiplicity between *Atom*s. Likewise, an E_H can carry information like the size of a collection, percentage of null values, maximum, minimum, and average of values. However, in this work, we focus mainly on the structural metadata. This catalog differs from a relational catalog in the expressiveness that allows to represent heterogeneous data models. By modeling the catalog of a polyglot system through a hypergraph, it is possible to retain the structural heterogeneity thanks to its high expressiveness and flexibility. Leveraging this information, it is interesting to see how we can retrieve the data from the polyglot system. Thus, our goal is to transform the formulation of a query over G into a query over the underlying data stores. Inter-data store data reconciliation or merges are out of the scope of this paper.

4.1 Query Representation

We assume that any query over the original RDF dataset or an equivalent query over the graph G corresponds to a query over the polyglot system. Hence, this query needs to be transformed into sub-queries that are executed on the relevant underlying data stores. For this, we introduce Algorithm 1 which builds an adjacency list for all the E_H whose closure contains *Atom*s of the query, aided by the incidence sets. Once all those E_H are generated and corresponding E_D identified, a simple projection query can be composed recursively with Algorithm 2 for each of the E_F according to different rules depending on the kind of data store. Algorithm 2 uses a prefix, a suffix and the path relevant for each of the constructs of the data stores.

We are only generating projection queries, but selections can be considered a posteriori by pushing down the predicates over the query path. Also, there can be cases where the same information is available in multiple data stores. If this happens, this overlap can be easily identified as the considered *Atom* will be contained in more than one E_D^+. Then a join should be performed in the corresponding mediator.

Algorithm 1 Query over polyglot system algorithm

Input: A Subgraph of G including the query Atoms and Relationships
Output: A set of multi language queries $\mathbb{Q}$ corresponding to data store queries

$\mathbb{Q} \leftarrow \emptyset$
$M \leftarrow newHashmap() < N, Set >$
$Q \leftarrow newQueue()$
for each $Atom\ a \in G$ **do**
  for each $E_H\ i \in a.incidenceSet$ **do** // *hyperedges containing an Atom*
    $Q.enqueue(< i, a >)$
  end for
end for
while $Q \neq \emptyset$ **do**
  $temp \leftarrow Q.dequeue$
  $current \leftarrow temp.first$
  $M.addToSet(current, temp.second)$ // *adds the second parameter to the set*
  for each $E_H\ j \in current.incidenceSet$ **do**
    $Q.enqueue(< j, current >)$
  end for
end while
for each $E_F\ f \in M.keys$ **do**
  $\mathbb{Q}.add(CreateQuery(f, ""))$
end for
return $\mathbb{Q}$

Algorithm 2 Create Query algorithm

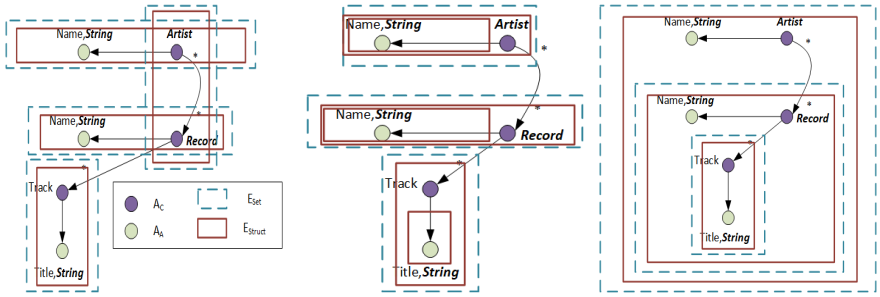
Input: *source* E_H, *path of* E_H (adjacency list M from Algorithm 1 is also available)
Output: A data store query q

$q \leftarrow prefixOf(source, path)$
for each $child \in M.get(source)$ **do**
  $q \leftarrow q + CreateQuery(child, pathOf(source))$
end for
$q \leftarrow q + suffixOf(source)$
return q

4.2 Constraints and Transformation Rules on Data Stores

Considering the constructs and the query generation algorithm mentioned earlier, each of the data store models would have its own rules and constraints on

the data. Therefore, in this section, we analyze the constraints and transformation rules for 3 of them: relational stores, document stores, and wide-column stores.



(a) for RDBMS (b) for Wide-Column Store (c) for Document Store

Fig. 4: Alternative data design representations

Relational Database Management Systems A typical example of the type of data design in RDBMS is shown in Figure 4a. The constraints and the mappings on E_H can be represented in a grammar as follows:

$$E_D^{Rel} \implies E_F^{Rel}{}^*,\ E_F^{Rel} \implies E_S^{Rel},\ E_S^{Rel} \implies A_C A^*$$

A traditional RDBMS data storage system consists of tables, tuples, and simple attributes. The data store can have multiple tables, which are represented by E_F^{Rel}. Within a table, the schema of the tuple E_S^{Rel} is fixed. Therefore, there can only be a single E_S^{Rel} inside a E_F^{Rel} . Finally, the tuple contains at least one A_C, which is the primary key, or multiple A_C in case of compound keys. The E_H containing an E_R that crosses two E_F^Rel (only one of both) is the one corresponding to the relation that has the attribute.

The RDBMS design of Figure 4a represents the following tables:
$Artist[A\_id, name], Record[R\_id, name],\ Artist\_Record[A\_id(FK), R\_id(FK)],$
$Track[T\_id, title, R\_id(FK)]$.

The following prefix and suffix are used in Algorithm 2 to generate the corresponding queries. Note that $deleteComma()$ is an operation that deletes the trailing comma of a string.

Symbol	prefix	suffix	path
E_F^{Rel}		"$FROM$" + E_F^{Rel}.name	
E_S^{Rel}	"$SELECT$"	$deleteComma()$	
A	A.name	","	

Table 1: Symbols for Algorithm 2 in RDBMS

Wide-Column Stores In a wide-column store, the data is stored in vertical partitions. A key and fixed column families identify each piece of data. Inside a column family, there can be an arbitrary number of qualifiers which identify values.

$$E_D^{col} \implies E_F^{col}{}^*,\ E_F^{col} \implies E_S^{col},\ E_S^{col} \implies A_C E_{Struct}^{col}{}^+,\ E_{Struct}^{col} \implies A^+$$

The outer most E_F^{col} represents the tables. E_F^{col} contains an E_S^{col}, which represents the rows. The A_C inside this E_S^{col} becomes the row key. E_S^{col} con-

tains several E^{col}_{Struct}, which represent the column families. The A inside the E^{col}_{Struct} represents the different qualifiers. The relationships between A_C can be represented as reverse lookups. In our example scenario, the hypergraph in Figure 4b contains one column family per table. This can be mapped into $Artist[A\_id, [name, \{R\_id\}]], Record[R\_id, [name]], Track[T\_id[title, R\_id]]$.

Wide-column stores generally support only simple get and put queries and require the row key to retrieve the data. We use HBase query structure to demonstrate the capability of simple query generation. Table 2 depicts the translation rules for simple queries in wide-column stores used in Algorithm 2.

Symbol	prefix	suffix	path
E^{Col}_F	"$scan'$ " + $E^{Col}_F.name$ + " ', {$COLUMNS$ => ["		
E^{Col}_S		$deleteComma()$ + "]}"	
E^{Col}_{Struct}			$path + E^{Col}_{Struct}.name$ + "."
A	"'" + $path + A.name$ + "'"	","	

Table 2: Symbols for Algorithm 2 in Wide-Column Stores

Document Stores Document stores have the least constraints when it comes to the data design. They enable multiple levels of nested documents and collections within. Figure 4c shows a document data store design of our example scenario. The constraints and mappings in a document store design are as follows:

$E^{Doc}_D \implies E^{Doc}_F{}^*$, $E^{Doc}_F \implies E^{Doc}_S{}^+$, $E^{Doc}_S \implies A_C(A|E^{Doc}_{Set}|E^{Doc}_{Struct})^*$,
$E^{Doc}_{Set} \implies E^{Doc}_{Struct}{}^+$, $E^{Doc}_{Struct} \implies (A|E^{Doc}_{Set}|E^{Doc}_{Struct})^+$

E^{Doc}_F represents the collections of the document store. E^{Doc}_S inside the E^{Doc}_F represents the documents within the collection, which must have an identifier A_C. Apart from that, E^{Doc}_S can have $Atom$s, or E^{Doc}_{Set}, which represents nested collections, or E^{Doc}_{Struct}, or a combination of any of them. E^{Doc}_{Set} represents a nested collection which contains documents E^{Doc}_{Struct}.

The design in Figure 4c can be mapped into a document store design as $Artist\{A\_id :, name :, Records : [\{R\_id :, name :, Tracks : [\{T\_id :, title : \}]\}]\}$

We use MongoDB syntax for the queries as it is one of the most popular document stores at the moment. Table 3 identifies the symbols for the queries.

Symbol	prefix	suffix	path
E^{Doc}_F	"db." + $E^{Doc}_F.name$ + "$.find$({}",{	"})"	
E^{Doc}_S		$deleteComma()$	
$E^{Doc}_{Struct/Set}$			$path + E^{Doc}_{Struct/Set}.name$ + "."
$A(path \neq \varnothing)$	""+$path + A.name$ + "": 1"	","	
$A(path = \varnothing)$	$A.name$ + " : 1"	","	

Table 3: Symbols for Algorithm 2 in Document Stores

Other Data Stores As discussed above, we have managed to model and infer constraints and transformation rules for RDBMS, wide-column stores, and document stores which cover most of the use cases. However, it is also interesting to see the capability of the approach to represent other data stores. Since our model is based on graphs, we can simply conclude that it can express graph data stores. We only need to map the data into G, and define a single set with all $Atom$s. The key-value stores do not have sophisticated data structures, and

it can be considered as a single column in a column family. Thus, since we are disregarding the storage of complex structures in the values that are not visible to the data store itself, we can state that our model covers key-value stores as well.

5 Use Case

In this section, we showcase our technique applied on an already available polyglot system. We base the example on the scenario used for ESTOCADA [5]. This involves a typical transportation data storage for a digital city open data warehousing. It uses RDBMS, document stores, and key-value stores. Figure 5 shows the graph G corresponding to ESTOCADA use case.

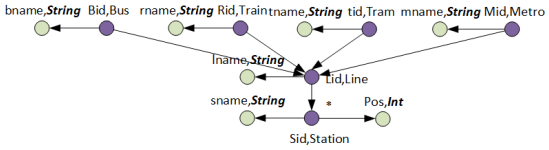


Fig. 5: Graph representation of ESTOCADA

The ESTOCADA system is used to store train, tram, and metro information in a RDBMS, the train and metro route information in a document store, and bus route together with the buses information in a key-value store (see [5] for more details). This information can be represented in our polyglot catalog[2] as follows (shown as containment sets):

$$C = \{E_D^{Rel}, E_D^{Kv}, E_D^{Doc}\}$$
$$E_D^{Rel} = \{E_{F\_Train}^{Rel}, E_{F\_Merto}^{Rel}, E_{F\_Tstat}^{Rel}, E_{F\_Mstat}^{Rel}, E_{F\_Station}^{Rel}\},$$
$$E_{F\_Train}^{Rel} = \{E_{S\_Train}^{Rel}\},\ E_{S\_Train}^{Rel} = \{A_{C\_rid}, A_{A\_rname}\},$$
$$E_{F\_Metro}^{Rel} = \{E_{S\_Metro}^{Rel}\},\ E_{S\_Metro}^{Rel} = \{A_{C\_mid}, A_{A\_mname}\},$$
$$E_{F\_Tstat}^{Rel} = \{E_{S\_Tstat}^{Rel}\},\ E_{S\_Tstat}^{Rel} = \{A_{C\_rid}, A_{C\_sid}, A_{A\_pos}\},$$
$$E_{F\_Mstat}^{Rel} = \{E_{S\_Mstat}^{Rel}\},\ E_{S\_Mstat}^{Rel} = \{A_{C\_mid}, A_{C\_sid}, A_{A\_pos}\},$$
$$E_{F\_Station}^{Rel} = \{E_{S\_Station}^{Rel}\},\ E_{S\_Station}^{Rel} = \{A_{C\_sid}, A_{A\_sname}\},$$
$$E_D^{Doc} = \{E_{F\_Metros.Trams}\},\ E_{F\_Metros.Trams}^{Doc} = \{E_{S\_Metros.Trams}^{Doc}\},$$
$$E_{S\_Metros.Trams}^{Doc} = \{A_{C\_mtid}, A_{A\_lname}, E_{Set\_route}^{Doc}\},$$
$$E_{Set\_route}^{Doc} = \{E_{Struct\_Station}^{Doc}\}, E_{Struct\_Station}^{Doc} = \{A_{A\_sname}\},$$
$$E_D^{Kv} = \{E_{F\_Station}^{Kv}, E_{F\_Bus}^{Kv}\},\ E_{F\_Station}^{Kv} = \{E_{S\_Station}^{Kv}\},$$
$$E_{S\_Station}^{Kv} = \{A_{A\_sname}, E_{Set}^{Kv\_loc}\},\ E_{Set}^{Kv\_loc} = \{A_{A\_lname}\},\ E_{F\_Bus}^{Kv} = \{E_{S\_Bus}^{Kv}\},$$
$$E_{S\_Bus}^{Kv} = \{A_{A\_lname}, E_{Set}^{Kv\_st}\},\ E_{Set}^{Kv\_st} = \{A_{A\_sname}\}$$

Our goal was to store the metadata of the ESTOCADA polyglot system with a hypergraph. Thus, we used HyperGraphDB[3] to save the entire catalog information including the *Atom*s, *Relationship*s, and *Hyperedge*s for the structures. With this catalog, one can quickly detect where each fragment of the polyglot system lies by merely referring to *Hyperedge*s and the content within.

[2] The implementation of the catalog is available in https://git.io/vxyHO
[3] http://www.hypergraphdb.org

6 Related Work

There are few polyglot systems already available to support heterogeneous NoSQL systems. In BigDAWG [7] different data models are classified as islands. Each of the islands has a language to access its data, and the data stores provide a shim to the respective islands it supports for a given query. Cross-island queries are also allowed, provided appropriate query planning and workload monitoring. Contrastingly, ESTOCADA [4, 5] enables the end user to pose queries using the native format of the dataset. In this case, the storage manager fragments and stores the data in different underlying data stores by analyzing the access patterns. These fragments may overlap, but the query executor decides the optimal storage to be accessed. ODBAPI [19] introduces a unified data model and a general access API for NoSQL and heterogeneous NoSQL systems. This approach supports simple CRUD operations over the underlying systems, as long as they provide an interface adhering to the global schema. SQL++ [16] introduces a unifying query interface for NoSQL systems as an extension of SQL to support complex constructs such as maps, arrays, and collections. This is used as the query language for the FORWARD middleware that unifies structured and non-structured data sources. [21] uses a similar approach to SQL++. Katpathikotakis et al. [13] introduces a monoid comprehension calculus-based approach which supports different data collections and arbitrary nestings of them. Monoid calculus allows transformations across data models and optimizable algebra. This enables the translation of queries into nested relational algebra, that can be executed in different data stores through native queries.

Using different adapters or drivers for heterogeneous data stores is a common approach used in polyglot systems. The aforementioned systems use the same principal. Liao et al. [15] uses adapter-based approach for RDBMS and HBase. The authors introduce a SQL interface to RDBMS and NoSQL system, a DB converter that transforms the information with table synchronization, and three mode query approach which provides different policies on how applications access the data. The Spring framework [12] is one of the most popular softwares used to access multiple data stores, as it supports different types by using specific drivers and a common access interface. Apache Gremlin [17] and Tinkerpop[4] follow a similar approach but particularly for graph data stores. The main drawback in this approach is that each and every implementation needs to adhere to a common interface, which is difficult due to the vast number of available data stores.

Standalone data stores have their own metadata catalogs. For example, HBase uses HCatalog[5] (for hive) to maintain the metadata. They are strictly limited to the respective data models involved. In our work, we introduce a catalog to handle heterogeneous data models. MongoDB, on the contrary, does not maintain any metadata but rather handles the documents themselves (without any schema information). But some work has been carried out in managing schema externally [22].

[4] http://tinkerpop.apache.org
[5] https://cwiki.apache.org/confluence/display/Hive/HCatalog

Several work has been carried out on data design methodologies for NoSQL systems. NoSQL abstract model (NoAM) [1,6] is designed to support scalability, performance, and consistency using concepts of collections, blocks and entries. This model organizes the application data in aggregates. It defines the four main activities: conceptual modeling, aggregate design, aggregate partitioning, and implementation. The aggregate storage in the target systems is done depending on the data access patterns, scalability and consistency needs. Similarly, a general approach for designing a NoSQL system for analytical workloads has been presented in [10]. It adapts the traditional 3-phase design methodology of conceptual, logical, and physical design, and integrates the relational and co-relational models into a single quantitative method. At the conceptual level, the traditional ER diagram is used and transformed into an undirected graph. Nodes denote the entities, and the edges represent their relationships, tagged with the relationship type (specialization, composition, and association). In cases where an entity can become a part of several different hypernodes, it is replicated in each of them. The Concept and Object Modeling Notation (COMN) introduced in [11] covers the full spectrum of not only the data store, but also the software design process. COMN is a graphical notation capable of representing the conceptual, logical, physical and real-world design of an object. This helps to model the data in NoSQL systems where the traditional Entity-Relationship (ER) diagrams fail in representing certain situations, such as nesting.

7 Conclusions and Future Work

In this paper, we introduced a hypergraph-based approach for managing the metadata of a polyglot system. We based our work on an already existing data exchange model (SOS). First, we formalized the design constructs, and extended them to support metadata management through a catalog. Next, we defined the constraints and the rules for simple query generation on heterogeneous data store models. Finally, we implemented a simple use case to showcase our approach on an existing polyglot system. We showed the expressiveness of hypergraphs, which is the essential feature of a canonical model of a federated system [18]. Then, effectively introduced a metadata management approach for polyglot systems leveraging this expressiveness.

In addition to the structural metadata, the statistical metadata can be easily included by extending the information kept in the *Hyperedge*s. Moreover, this work allows us to identify restrictions and limitations of different underlying data store models that can aid in data design decisions. Thus, this formalization of the data design can be extended to make data design decisions on NoSQL systems as well as optimizing an existing design by transforming the data using the hypergraph representation.

Acknowledgments

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate Information Technologies for Business Intelligence - Doctoral College (IT4BI-DC)

References

1. Atzeni, P., Bugiotti, F., Cabibbo, L., Torlone, R.: Data modeling in the NoSQL world. Computer Standards & Interfaces (2016)
2. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to non-relational database systems: The SOS platform. In: International Conference on Advanced Information Systems Engineering, CAiSE (2012)
3. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to NoSQL systems. Information Systems **43** (2014)
4. Bugiotti, F., Bursztyn, D., Deutsch, A., Manolescu, I., Zampetakis, S.: Flexible hybrid stores: Constraint-based rewriting to the rescue. In: IEEE 32nd Int. Conf. on Data Engineering, ICDE (2016)
5. Bugiotti, F., Bursztyn, D., Diego, U.C.S., Ileana, I.: Invisible glue : Scalable self-tuning multi-stores. CIDR (2015)
6. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for NoSQL systems. In: Int. Conf. on Conceptual Modeling. ER (2014)
7. Duggan, J., Zdonik, S., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T.: The BigDAWG polystore system. ACM SIGMOD Record **44**(2) (2015)
8. Euzenat, J., Shvaiko, P., et al.: Ontology matching, vol. 18. Springer (2007)
9. Garcia-Molina, H., Ullman, J., Widom, J.: Database systems: the complete book. Pearson Education India (2008)
10. Herrero, V., Abelló, A., Romero, O.: NoSQL design for analytical workloads: Variability matters. In: 35th Int. Conf. Conceptual Modeling:, ER (2016)
11. Hills, T.: NoSQL and SQL Data Modeling. Technics Publications (2016)
12. Johnson, R., Hoeller, J., Donald, K., Pollack, M., et al.: The spring framework–reference documentation. Interface **21** (2004)
13. Karpathiotakis, M., Alagiannis, I., Ailamaki, A.: Fast queries over heterogeneous data through engine customization. Proc. of the VLDB Endowment **9**(12) (2016)
14. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and abstract syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, accessed: 2018-02-16
15. Liao, Y.T., Zhou, J., Lu, C.H., Chen, S.C., Hsu, C.H., Chen, W., Jiang, M.F., Chung, Y.C.: Data adapter for querying and transformation between SQL and NoSQL database. Future Generation Computer Systems **65** (2016)
16. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: 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)
17. Rodriguez, M.A.: The Gremlin graph traversal machine and language. In: 15th Symposium on Database Programming Languages. ACM (2015)
18. Saltor, F., Castellanos, M., García-Solaco, M.: Suitability of data models as canonical models for federated databases. ACM Sigmod Record **20**(4) (1991)
19. Sellami, R., Bhiri, S., Defude, B.: Supporting multi data stores applications in cloud environments. IEEE Trans. on Services Computing **9**(1) (2016)
20. Shvaiko, P., Euzenat, J.: Ontology matching: state of the art and future challenges. IEEE Trans. on knowledge and data engineering **25**(1) (2013)
21. Vathy-Fogarassy, Á., Hugyák, T.: Uniform data access platform for SQL and NoSQL database systems. Information Systems **69** (2017)
22. Wang, L., Hassanzadeh, O., Zhang, S., Shi, J., Jiao, L., Zou, J., Wang, C.: Schema management for document stores. PVLDB **8**(9) (2015)