Data & Knowledge Engineering 134 (2021) 101896

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Managing polyglot systems metadata with hypergraphs

Moditha Hewasinghage [a,b,*], Alberto Abelló [a], Jovan Varga [a], Esteban Zimányi [b]

[a] Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain
[b] Université Libre de Bruxelles, 1050 Bruxelles, Belgium

ARTICLE INFO

Keywords:
Metadata management
NoSQL
Polystore

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. Then, we define design constraints and query transformation rules for three representative data store types. Next, we propose a simple query rewriting algorithm from the metadata of the catalog to underlying data store specific ones and provide a prototype implementation. Furthermore, we introduce a storage statistics estimator on the underlying data stores. Finally, we show the feasibility of our approach on a use case of an existing polyglot system, and its usefulness in metadata and physical query path calculations.

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 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 that represent data in a document-like structure, which can become increasingly complex with nested elements; (iv) Graph stores that 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 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 controlling the storage and retrieval more efficiently for different requirements. Hence, polyglot systems were introduced, similar to traditional Federated Database Systems (FDBMS), but with more complexity considering the need to handle semistructured data models. Due to the heterogeneity at different levels, most of the work on polyglot systems [1,2] 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 [3]) 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. Moreover, it also helps to improve the design of the polyglot system and determine the statistics and the access

[*] Corresponding author at: Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain.
E-mail addresses: moditha@essi.upc.edu (M. Hewasinghage), aabello@essi.upc.edu (A. Abelló), jvarga@essi.upc.edu (J. Varga), ezimanyi@ulb.ac.be (E. Zimányi).

https://doi.org/10.1016/j.datak.2021.101896
Received 11 February 2021; Accepted 10 May 2021
Available online 13 May 2021

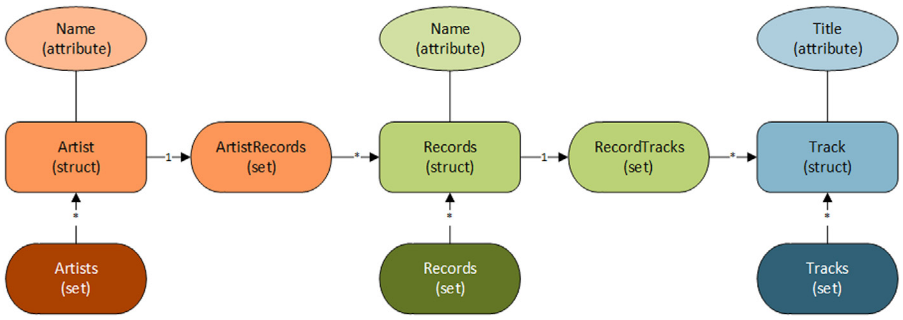

Fig. 1. SOS representation of the example.

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? What would be the storage requirements for a particular data store design? What are the access patterns of a particular query over a particular datastore? 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 [4,5], and formalizing the constructs through hypergraphs.

The Save Our Systems (SOS) Model [5] 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 representing 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, defining a common conceptual model for the metadata of any NoSQL system, which we have formalized through definitions of the concepts based on logics. This work is an extension of our previous conference paper [6] with refinements in the original definitions and improvements carried out to maintain additional statistical metadata calculations. 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. Afterwards, we introduce how our catalog can be used to calculate storage statistics and physical access patterns for queries using document stores as an example. 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 [7]. We used this flexible meta-representation to evaluate design alternatives in document stores in terms of storage space, query cost and access patterns.[1]

This paper is organized as follows: First, in Section 2 we introduce the necessary 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 6. Next, in Section 5, we explain how the metadata representation can be extended to maintain statistical metadata and calculate storage requirements and physical query access patterns. Finally, we introduce the related work in Section 7, and conclude and discuss future work in Section 8.

2. Preliminaries

In this section, we introduce the basic concepts of Resource Description Framework (RDF) [8] and SOS Model [4,5] that are used in our approach.

2.1. Resource Description Framework (RDF)

The Resource Description Framework [8] is a World Wide Web Consortium (W3C) specification for representing information on the Web. It is a graph-based data model that enables sharing of information and 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,

[1] https://vimeo.com/396513259.

which is a data value. An example of RDF is shown in Listing 1, written in Turtle notation.[2] The example contains information about music albums, artists, and songs and is used throughout the paper.

Listing 1 Example RDF dataset

```
@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 .
```

2.2. SOS model

The high flexibility of NoSQL systems gives freedom to have multiple designs for the same data. A particular data design built focusing on a specific scenario 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 [4,5] 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. These different data models exposed by NoSQL datastores are effectively managed in the SOS data model with three major constructs: *Attribute*, *Struct*, and *Set* [4].

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 *Attributes*, *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 [5]. 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 *Attributes* and groups of *Attributes*, 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. A possible SOS representation of the example is shown in Fig. 1.

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. Thus, having the schemas of the underlying data stores and having a global schema is important in our model. However, this is beyond the scope of the current work. The schema of a structured data store such as RDBMS can be extracted through the underlying DDL. Schema inference from semi-structured data has also been carried out [9]. Moreover, previous work has shown that this global schema can be obtained by extracting the schema of each data store and reconciling them [10,11].

Building on top of the RDF data model discussed in Section 2.1, we introduce our design constructs based on the SOS Model. Fig. 2 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 *Attributes*), 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., $\mathbb{A} = \{A_1, A_2 \ldots A_n\}$).

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

[2] http://dbtune.org.

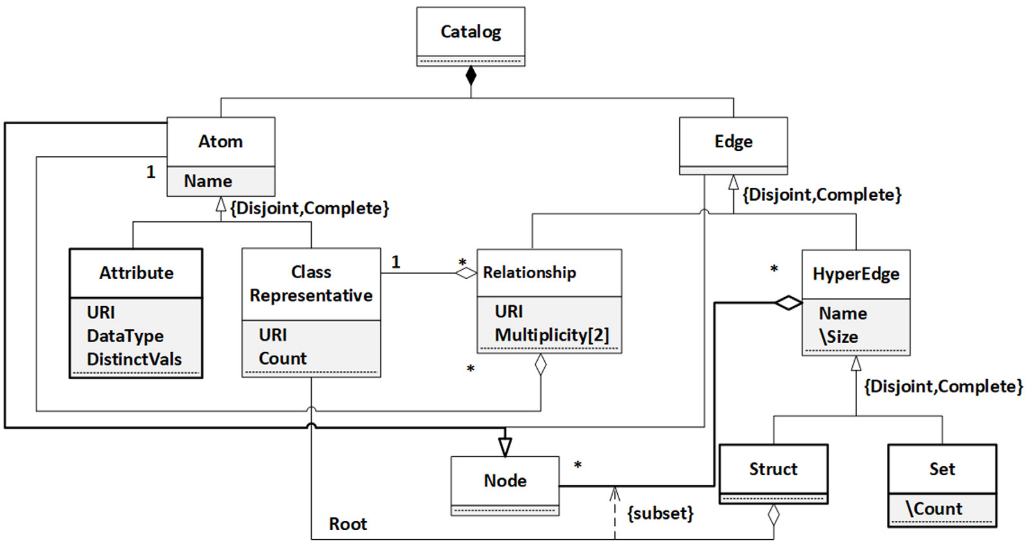

Fig. 2. Class diagram for the overall catalog.

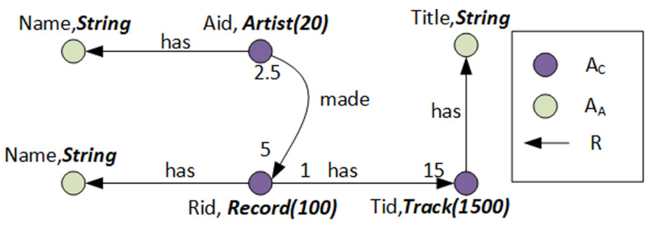

Fig. 3. Translated graph built from the RDF.

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. Additionally, the information on the *Atoms* can be enhanced by the distinct values for A_As and the number of instances on A_Cs identified as $Count$.

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: $E_R^{x,y} = \langle A^x, A^y, u \rangle | A^x, A^y \in \mathbb{A} \wedge (A^x \in \mathbb{A}_C \vee A^y \in \mathbb{A}_C)$

The E_Rs that connect two A_Cs can include the multiplicities between the two classes. Since the relationships are bidirectional multiplicities are also diploid.

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. G us immutable as it contains the knowledge about the data. Fig. 3 shows the graph G of the original RDF example in Listing 1. Here, we can assume that each artist can have 5 records and each record has 2.5 artists on average. A record has on average 15 tracks and each track is in 1 record. Finally, there are 20 *Artist*, 100 *Record*, and 1500 *Track* instances.

We build our data design on top of G, based on the constructs introduced in SOS model. Thus, we make use of $Hyperedge$s 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 $\mathbb{E}_R$, and/or hyperedges $\mathbb{E}_H$ (a). Every struct has a special predefined root atom that enables to identify the struct noted as $O(E_{Struct})$. The root itself is a subset of the E_{Struct}s composition. All the As must have a unique path of relationships to its root which is also part of the struct (b). All the roots of the nested E_{Struct}s inside a parent E_{Struct} must have a unique path of relationships from the root of the parent, and this path must be inside the parent (c). All the E_{Set}s inside a parent E_{Struct} must contain a set of relationships that connects some A_C of the parent to the root of the child E_{Struct} or the A in the E_{Set} (d). All of the E_Rs inside a E_{Struct}, must be involved in a path that connects either the child As or the root of the child E_{Struct}s (e).

(a) $E_{Struct} \subseteq \mathbb{A} \cup \mathbb{E}_R \cup \mathbb{E}_H$
(b) $\forall a \in (self \cap \mathbb{A}) : \exists! \{E_R^{O(self),x_1}, \dots, E_R^{x_n,a}\} \subseteq self$
(c) $\forall s \in (self \cap \mathbb{E}_{Struct}) - O(self) : \exists! \{E_R^{O(self),x_1}, \dots ., E_R^{x_n,O(s)}\} \subseteq (self \cup \mathbb{A}_C)$
(d) $\forall s \in (self \cap \mathbb{E}_{Set}), \forall t \in (s \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : \exists! \{E_R^{y,x_1}, \dots ., E_R^{x_n,z}\} \subseteq s \wedge y \in (self \cap \mathbb{A}) \wedge (t \in \mathbb{A} ? z = t : z = O(t))$
(e) $\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : (\exists y \in (self \cap \mathbb{A}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \dots, E_R^{x_n,y}\} \subseteq self) \vee (\exists y \in (self \cap \mathbb{E}_{Struct}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \dots, E_R^{x_n,O(y)}\} \subseteq self)$

Definition 8. A set E_{Set} is a hyperedge that contains a set of arbitrary E_{Struct}s, As, and specific relationships $\mathbb{E}_{\mathbb{R}}$ (a). All the relationships in the set must originate from a class atom of the parent of the set and the destination should be the root of a child struct or A of the set (b).

(a) $E_{Set} \subseteq \mathbb{E}_{Struct} \cup \mathbb{A} \cup \mathbb{E}_R$
(b) $\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : \exists A_C^x \in self.parent, \exists y \in (self \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : E_R^{a,b} \in \{E_R^{x,x_1}, \dots, E_R^{x_n,z}\} \subseteq self \wedge (y \in \mathbb{A} ? z = y : z = O(y))$

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. Fig. 4 extends our original diagram of constructs to support those.

The E_{Set} are specialized into two types: $Data\ Store$ $\mathbb{E}_D$ and $First\ Level$ $\mathbb{E}_F$. E_D represents a concrete data store of the polyglot system. E_F denotes a set of instances in the particular data store. All the allowed kinds of data stores are a subclass of $\mathbb{E}_D$. Moreover, $E_D.incidenceSet = \emptyset$. Thus, we define three kinds of E_D (namely $Relational \mathbb{E}_D^{Rel}$, $Document\ Store$ $\mathbb{E}_D^{Doc}$, and $Wide - Column$ $\mathbb{E}_D^{Col}$ in Fig. 4), 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.

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}. The specialized E_D, E_F, and E_S identify specific E_Hs in the data store constraints.

Each E_H carries a name which is interpreted depending on the context. In E_D, it represents the physical location of the underlying data store. In E_F, it is 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. 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.

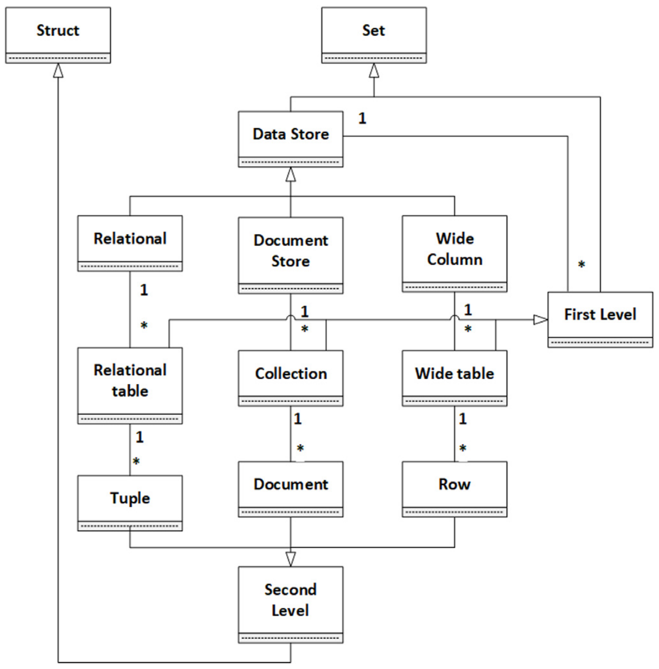

Fig. 4. Class diagram for $Hyperedge$ hierarchy.

Algorithm 1 Query over polyglot system algorithm

```
Input: A query $q$
Output: A set of multi language queries $\mathbb{Q}$ corresponding to data store queries
 1: $\mathbb{Q} \leftarrow \emptyset$
 2: $M \leftarrow newHashmap() < E_H, Set >$
 3: $Q \leftarrow newQueue()$
 4: for each  $Atom\ a \in q$ do
 5:    for each  $E_H\ i \in a.incidenceSet$ do      // hyperedges containing an Atom
 6:       $Q.enqueue(< i, a >)$
 7:    end for
 8: end for
 9: while $Q \neq \emptyset$ do
10:    $temp \leftarrow Q.dequeue$
11:    $current \leftarrow temp.first$
12:    $M.addToSet(current, temp.second)$    // adds the second parameter to the set
13:    for each  $E_H\ j \in current.incidenceSet$ do
14:       $Q.enqueue(< j, current >)$
15:    end for
16: end while
17: for each  $E_F\ f \in M.keys$ do
18:    $\mathbb{Q}.add(CreateQuery(f,""))$
19: end for
20: return $\mathbb{Q}$
```

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 (hash map M) whose closure contains $Atom$s of the query, aided by the incidence sets. First, all the E_Hs that contains $Atom$s in the input query q is added into a queue Q as a pair of $\langle E_H, A\rangle$ (lines 4–8). Then, a pair $temp$ is dequeued from Q and M is updated with the E_H in $temp.first$ as the key and $temp.second$ (can be an $Atom$ orE_H) added to the corresponding value set with the help of $addToSet$ in line 12. All the E_Hs in the $temp.first$'s incidence set is added to the queue Q (lines 13–15). This process is carried out until the queue Q has no more elements (lines 9–16). The generated adjacency list can be used to identify the E_Ds that corresponds to the relevant underlying data stores for the query. Once this adjacency list M is generated and corresponding E_D identified, a simple projection query can be composed

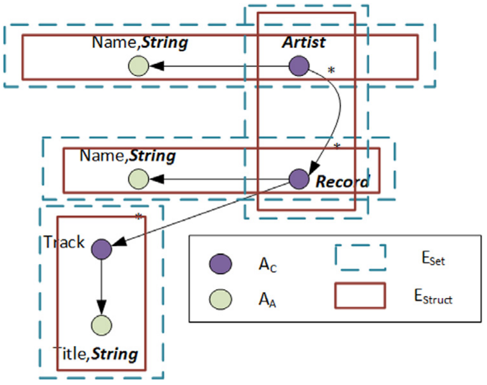

Fig. 5. An example data design for an RDBMS.

recursively with Algorithm 2 for each of the E_F according to different rules depending on the kind of data store (lines 17–19). Algorithm 2 uses a prefix, a suffix and the path relevant for each of the constructs of the data stores.

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 $q$
1: $q \leftarrow prefixOf(source, path)$
2: for each $child \in M.get(source)$ do
3:     $q \leftarrow q + CreateQuery(child, pathOf(source))$
4: end for
5: $q \leftarrow q + suffixOf(source)$
6: return $q$
```

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.

Definition 9. A query q over the hypergraph G is a connected graph consisting of a selection atom sel, a set of projection atoms $proj$, a set of relationships rel and a frequency $freq$.
Formally : $q = \langle sel, proj, rel, freq\rangle | sel \in A, \forall a \in proj : a \in A, \forall A^x, A^y \in sel \cup proj : \exists\{E_R^{x,x_1}, E_R^{x_1,x_2}, \ldots, E_R^{x_n,y}\} \in rel, 0 < freq \leq 1$

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.

4.2.1. Relational database management systems

A typical example of the type of data design in RDBMS is shown in Fig. 5. 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. The E_H containing an E_R that crosses two E_F^{Rel} corresponds to the relation that has the foreign key together with the relevant A_C.

The RDBMS design of Fig. 5 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 prefix and suffix in Table 1 are used in Algorithm 2 to generate the corresponding queries. Note that $deleteComma()$ is an operation that deletes the trailing comma of a string.

Table 1
Symbols for Algorithm 2 in RDBMS.

Symbol	Prefix	Suffix	Path
E_F^{Rel}		$"FROM" + E_F^{Rel}$.name	
E_S^{Rel}	$"SELECT"$	$deleteComma()$	
A	A.name	","	

Table 2
Symbols for Algorithm 2 in wide-column stores.

Symbol	Prefix	Suffix	Path
E_F^{Col}	$"scan'" + E_F^{Col}.name + "',\{COLUMNS => ["$		
E_S^{Col}		$deleteComma() + "]\}"$	
E_{Struct}^{Col}			$path + E_{Struct}^{Col}.name + "."$
A	$"'" + path + A.name + "'"$	","	

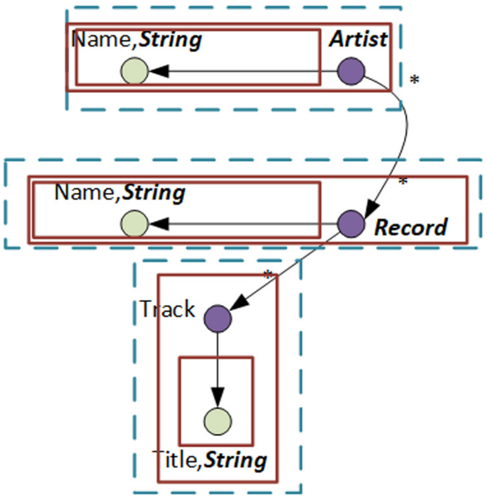

Fig. 6. An example data design for wide column store.

4.2.2. *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} contains several E_{Struct}^{col}, which represent the column families. The A inside the E_{Struct}^{col} represents the different qualifiers. The relationships between A_C can be represented as reverse lookups. In our example scenario, the hypergraph in Fig. 6 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.

4.2.3. *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. Fig. 7 shows a document data store design of our example scenario. The constraints and mappings in a

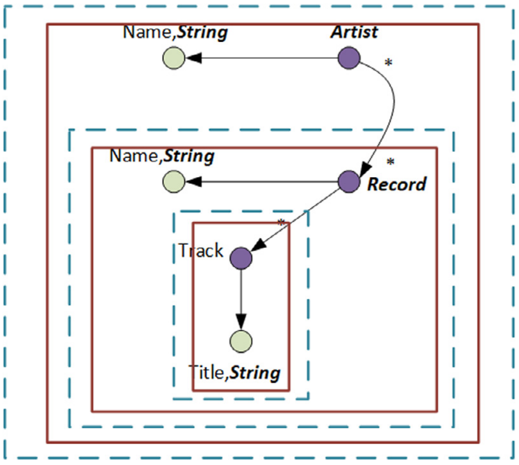

Fig. 7. An example data design for document store.

Table 3
Symbols for Algorithm 2 in document stores.

Symbol	Prefix	Suffix	Path
E_F^{Doc}	$``db." + E_F^{Doc}.name + ``.find(\{\}",\{$	$``\})"$	
E_S^{Doc}		$deleteComma()$	
$E_{Struct/Set}^{Doc}$			$path + E_{Struct/Set}^{Doc}.name + "."$
$A(path \neq \varnothing)$	$"""" + path + A.name + "":1"$	$","$	
$A(path = \varnothing)$	$A.name + ":1"$	$","$	

document store design are as follows:

$$
\begin{aligned}
&E_D^{Doc} \implies E_F^{Doc}*,\ E_F^{Doc} \implies E_S^{Doc}+,\\
&E_S^{Doc} \implies A_C(A|E_{Set}^{Doc}|E_{Struct}^{Doc})^*,\ E_{Set}^{Doc} \implies (A|E_{Struct}^{Doc})^+,\\
&E_{Struct}^{Doc} \implies (A|E_{Set}^{Doc}|E_{Struct}^{Doc})^+
\end{aligned}
$$

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

The design in Fig. 7 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.

4.2.4. 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, we can state that our model covers key–value stores as well.

5. Calculating statistical and storage metadata

We extend our metadata representation by including statistics on the $Hyperedge$s. For this, we consider the data type and the number of distinct values on the A_As, count on A_Cs, and the multiplicity on each end of the E_Rs. E_{Set} will also have the cardinality as a calculated value (see Fig. 2). Using the count and the multiplicity, we can calculate the total storage size for each of the storage structures (relational table, collection, or wide table). In document stores, if there are nested attributes within a collection (i.e., Records or Tracks inside an Artist in Fig. 7) it is important to calculate the multiplicity between the $O(E_{S/Struct})$ and the $O(E_{Struct})$ of the nested E_{Struct}. This information can be used to estimate the size of secondary indexes, especially with multiple nesting levels. Moreover, the same information can be used to determine the physical access patterns of a particular query over a certain datastore. Even though these calculations can be carried out on any datastore, it is particularly interesting in document stores, because they are more flexible on storing data compared to RDBMS and column stores, mainly due to nesting. Thus, we focus on document stores to explain the algorithms.

5.1. Storage size estimation

We take our running example in Fig. 3 to illustrate how Algorithms 3 and 4 can be used to calculate the storage size and the multipliers between the *Root* atom and the rest of the Atoms in an E_{Struct}. The multiplicity between *Artist* and *Record* is 5 and between *Record* and *Track* is 15, and there are 20, 100, 1500 instances of *Artist*s, *Record*s, and *Track*s, respectively. Let us assume all A_Cs are integers of 4 bytes in size, and all the A_A Strings are 10 bytes.

Algorithm 3 works in a recursive manner going through each of the E_Hs, calculating the size of each of the *nodes* inside, and multiplying them by the number of instances of the E_H. In the case of an *Atom*, this will be the size of the stored data (i.e., 4 bytes for an integer). In case of a named E_{Struct} or E_{Set}, it will add the name length to the total size, and its content. E_{Set} represents a collection of elements that could be a list or an array. In this case, we need to calculate the size of this list/array. Algorithm 4 finds the number of instances of such complex objects by referring to the multiplicity of the relationship between parent and the child E_Hs.

If we want to calculate the storage size and the multipliers of the atoms of the schema in Fig. 7, the collection hyperedge will be the input to Algorithm 3. In the first iteration, the size will be set to 0 (Line 6). Then, at line 16, the top level document will be found, and the size will be increased by the length of the name of the collection (6 assuming it is “Artist”). When the embedded *Records* are reached, the E^{Doc}_{Set} will have size 7 (with the name “Records”). Then the *Records* will look into the embedded *Tracks*. The *Tracks* will have a size of 7 with the name, and then the T_ID and the T_NAME will be 4 and 10 bytes respectively, making an individual track 14 bytes overall. Moreover, the E^{Doc}_{Set} that contains the relationship between *Record* and *Track* will return the multiplier being 15. Therefore, the record size will be calculated as the sum of 15 tracks, the length of the text “Tracks”, R_ID and the R_NAME adding up to 230 ($15 * 14 + 6 + 4 + 10$) bytes. The E^{Doc}_{Set} that contains the relationship between *Artist* and *Record* will return the multiplier 5. Hence, the *Records* inside the Artists will have a size of 1150 ($230 * 5$) bytes. The A_ID, A_NAME, and *Records* inside the *Artist* document will get a multiplier of 20. Since there are 20 *Artists*, the total size of the collection will increase to 23,426 ($6 + 20 * (4 + 10 + 7 + 1150)$) bytes.

Algorithm 3 CalculateSize algorithm

```
Input: $source \in A \cup E$
Output: Size $s$, Hashmap <A,multiplier> $map$
1: $map \leftarrow newHashmap() < A, multiplier >$
2: if $source \in A$ then
3:     $s \leftarrow source.size$
4:     $map.put(source, 1)$
5: else if $source \in E_F^{Doc} || source \in E_S^{Doc}$ then
6:     $s \leftarrow 0$
7: else if $source \in E_{Set}$ then      // Embeded list
8:     $s \leftarrow source.name.length()$
9: else if $source \in E_{Struct}$ then
10:     if $source.name = \emptyset$ then      // Struct inside a set
11:         $s \leftarrow 0$
12:     else      // Embedded struct
13:         $s \leftarrow source.name.length()$
14:     end if
15: end if
16: for each $child \in source.getChildren()$ do
17:     $multiplier \leftarrow CalculateMultiplier(source, child)$
18:     $result \leftarrow CalculateSize(child)$
19:     $s \leftarrow s + result.s * multiplier$
20:     for each $key \in result.map.keys()$ do
21:         $map.add(key, result.map.get(key) * multiplier)$
22:     end for
23: end for
24: return $< s, map >$
```

Apart from the total size, Algorithm 3 also returns the number of instances of each of the *Atom*s within that collection. From the previous example, we get 20, 100, 1500 as the number of instances for A_ID, R_ID, and T_ID respectively. By dividing the number of instances by the number of instances of *Root*, we can obtain the multiplying factor between the root and the nested *Atom*s. This value together with the distinct values of an A_A can be used to identify the potential secondary index sizes and their effectiveness. For example, in an alternative document store design where the tracks are the first level documents that embed the records and the record embed the artists, with the same calculations we would get 1500 as the number of *Author*s stored. However, since we know that there are only 20 different *Author*s, if we build a secondary index on A_ID, we can estimate that each A_ID index entry would point to $\frac{1500}{20} = 75$ top level documents (*Tracks*).

Algorithm 4 CalculateMultiplier algorithm

```
Input: $source \in E_H^{doc}$, $child \in M$
Output: $Multiplier\ m$
1: if $source \in E_S^{doc}$ then    // top level collection
2:     $m \leftarrow source.root.count$
3: else if $source \in E_{Set}^{doc}$ then
4:     $relationship \leftarrow source.findRelationship(*, child.root)$    // Set has one parent
5:     $m \leftarrow relationship.Multiplicity$
6: else
7:     $m \leftarrow 1$
8: end if
9: return $m$
```

5.2. Physical access patterns for workloads

A query workload is usually provided as the frequencies of the respective queries. However, the access patterns and the runtimes of the queries depend on the underlying schema design, as shown in works such as [4] and [12]. Therefore, we believe that it is of interest to determine how the underlying physical storage structures are used for a particular workload on different data stores and schema designs. We assume that the workload is given as a set of queries $\mathbb{Q}$ as stated in Definition 9, together with their access frequency. Thus, we use Algorithm 5 to calculate the access frequencies of the collections and their secondary indexes depending on the design choices.

Algorithm 5 CalculateFrequency algorithm

```
Input: $\mathbb{Q}$, $\mathbb{E}_S^{Doc}$
Output: Hashmap<$E_S^{Doc} \cup A_C, freq$> $m$
1: $m \leftarrow newHashmap() < E_S^{Doc} \cup A_C, freq >$
2: $winners \leftarrow newQueue()$
3: for each $query \in \mathbb{Q}$ do
4:     $remaining \leftarrow query.proj$
5:     $cands \leftarrow E_S^{Doc}.find(contains(query.sel))$
6:     if candidates.size = 1 then
7:         $winner \leftarrow cands[0]$
8:     else    // winner is the smallest collection with the selection as root
9:         $winner \leftarrow cands.find(candidate.root = query.sel \wedge candidate.size = findMinSize(cands))$
10:    end if
11:    if $winner = \emptyset$ then    // no selection as the root
12:        $remaining.add(query.sel)$
13:        $winner \leftarrow cands.find(candidate.size =$
           $findMinSize(cands.size))$
14:    end if
15:    $m.Aupdate(< winner,$
       $query.freq \cdot winner.getMultiplier(query.sel) >)$
16:    $m.Aupdate(< winner.get(query.sel), query.freq >)$
17:    $winners.enqueue(winner)$
18:    while $winners \neq \emptyset \wedge remaining = \emptyset$ do
19:        $main \leftarrow winners.dequeue()$
20:        $covered \leftarrow main^+.Atoms \cup remaining$
21:        $remaining \leftarrow remaining - main^+.Atoms$
22:        for each $A_c \in covered$ do
23:            for each $A \in remaining$ do
24:                $rel \leftarrow query.rel.findRelationship(A_c, A)$
25:                if $rel \neq \emptyset$ then    // found a join
26:                    $join \leftarrow G.findCollection(contains(A, A_c))$
27:                    $m.Aupdate(< join, m.get(main)\cdot$
                       $rel.Multiplicity \cdot join.getMultiplier(A_c) >)$
28:                    $m.Aupdate(< join.get(A_c),$
                       $m.get(main) \cdot rel.Multiplicity >)$
29:                    $winners.enqueue(join)$
30:                end if
31:            end for
32:        end for
33:        $SumToUnity(m)$
34:    end while
35: end for
36: return $m$
```

We use a simple query structure with a single selection predicate and multiple projections. Nevertheless, notice that multiple tables/collections within the data store may still be capable of answering the same query. In such cases, we use a greedy approach for selecting the winner as the smallest sized collection out of the candidates. Thus, the base case scenario is a single collection

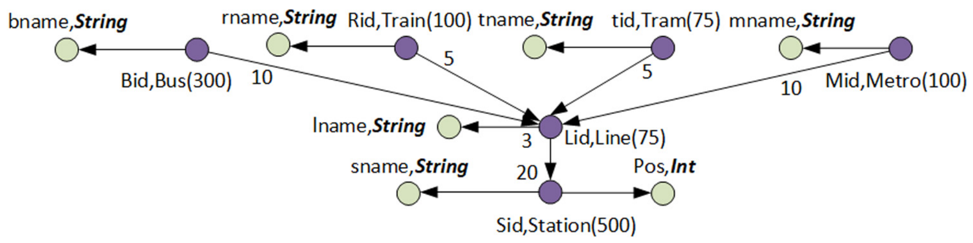

Fig. 8. Graph representation of ESTOCADA.

containing the selection predicate (Line 6), followed by the smallest collection containing the selection *as the root* if many contain it (Line 8), and finally, the smallest collection that contains the selection predicate if none of them have it has the *root* (Line 11).

For example, let us assume that we want to retrieve R_IDs and T_IDs by A_ID with a frequency of 1, the records embed the artists, and the *Tracks* have a separate collection with R_ID as the reference. We will have the *Record* as the winning collection. If there is a secondary index on a particular A, each query will use that index once, and the top level collection will be accessed $multiplier$ times. Therefore, the frequency of the collection is higher than the actual query (Lines 15 and 16). Thus, we multiply the query frequency by the multiplier of the selection predicate as the current frequency of the collection ($1 * 5$ for *Record*) and the selection predicate A_ID, is accessed with the frequency of the query (1). The $getMultiplier(A)$ method will return the number of instances calculated in Algorithm 3 divided by the number of root instances.

All the As transitively contained in the document that belongs to the query are already retrieved by accessing it (R_ID). However, if there are other As in different collections, they need to be joined outside of the data store (T_ID). Assuming that we use a row-nested loop join, the index of the joined collection needs to be accessed $multiplicity$ times with regard to the original collection, as shown in line 28, and the collection is accessed according to the referred index as in line 27. Hence, the index on R_ID is accessed 5 times, and the collection is accessed 75 ($5 * 15$) times. The algorithm continues until all the As in the query are covered by the data store. Finally, in line 33, we use a normalizing factor to sum to unity, giving us the relative access frequencies of different storage structures in the overall data store for this particular workload. In our example, the four physical structures of *Record* and *Track* collections and the indexes on A_ID in *Record* and R_ID on *Track* will be accessed with overall frequency of 0.059 ($\frac{5}{5+1+5+75}$), 0.87 ($\frac{375}{5+1+5+75}$), 0.012 ($\frac{1}{5+1+5+75}$), and 0.059 ($\frac{5}{5+1+5+75}$), respectively.

The Algorithms 3, 4, and 5 enable us to identify the storage requirements of data and physical access patterns of the queries. Thus, we can determine the storage sizes and the individual access frequencies of the collections and their indexes (both primary and secondary). Applying these frequencies and the storage sizes will allow us to estimate the query performance for different datastore designs. We have implemented the approach in [13], to evaluate alternative schema designs on document stores using the cost model in [14] on storage space and query performance.

6. 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 [15]. This involves a typical transportation data storage for a digital city open data warehousing. It uses RDBMS, document stores, and key–value stores. Fig. 8 shows the corresponding graph G. The multiplicities are shown with the corresponding arrows (e.g. each line goes through 20 stations). We have omitted the multiplicity of each bus, train, tram, and metro being associated with one line for clarity of the figure.

The ESTOCADA system is used to store train, tram, and metro information in an 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 [15] for more details). This information can be represented in our polyglot catalog[3] as follows (shown as containment sets):

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

[3] The implementation of the catalog is available in https://git.io/vxyHO.

Table 4
Access frequencies of the document store storage structures.

Query	Usage	
	sid	$MetrosTrams$
Q1 (p = 0.5)	0.5	0.5 * 3 = 1.5
Q2 (p = 0.5)	0.5	0.5 * 3 = 1.5
Total	1	3
Frequency	$\frac{1}{21}$ = 0.25	$\frac{20}{21}$ = 0.75

$$
\begin{aligned}
&E^{Doc}_{S\_Metros.Trams} = \{A_{C\_lid}, A_{A\_lname}, E^{Doc}_{Set\_route}\},\\
&E^{Doc}_{Set\_route} = \{E^{Doc}_{Struct\_Station}\}, E^{Doc}_{Struct\_Station} = \{A_{C\_sid}, A_{A\_sname}\},\\
&E^{Kv}_{D} = \{E^{Kv}_{F\_Station}, E^{Kv}_{F\_Bus}\},\ E^{Kv}_{F\_Station} = \{E^{Kv}_{S\_Route}\},\\
&E^{Kv}_{S\_Route} = \{A_{A\_lname}, E^{Kv}_{Set\_loc}\},\\
&E^{Kv}_{Set\_loc} = \{A_{A\_sname}\},\ E^{Kv}_{F\_Bus} = \{E^{Kv}_{S\_Bus}\},\\
&E^{Kv}_{S\_Bus} = \{A_{C\_bid}, A_{A\_lname}\}
\end{aligned}
$$

Our goal was to store the metadata of the ESTOCADA polyglot system with a hypergraph. Thus, we used HyperGraphDB[4] to save the entire catalog information including the *Atoms*, *Relationships*, and *Hyperedges* 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.

Let us assume that the following queries are issued on the catalog with equal probability.

- **Q1:** Find information about trains on a given station (`sid, sname, rid, rname`).
- **Q2:** Find the metro lines on a given station (`sid, sname, lid, lname`)

By utilizing Algorithms 1 and 2, we can generate the following data store specific queries (selections are added a priori).

-
  ```
  SELECT rid, rname FROM Train
  SELECT sid, sname FROM Station WHERE sid = <>
  SELECT sid, rid FROM Tstat
  db.MetrosTrams.find({route.sid:<>},{route.sid:1, route.sname:1})
  ```

-
  ```
  db.MetrosTrams.find({route.sid:<>},{lid:1, lname:1, route.sid:1, route.sname:1})
  SELECT sid, sname FROM Station WHERE sid = <>
  ```

For the ease of demonstration, let us assume that all the identifiers are 4 bytes and the names are 20 bytes. Then, using the multiplicities provided in G (Fig. 8) we can calculate the storage sizes and the multipliers using Algorithms 3 and 4. Thus, we get a storage size of 54,150 bytes for a document store. Each nested station document is 34 bytes (20+4+4+6). Then, the route having list of 20 stations becomes 688 bytes (20*34+8). Next, the line contains the route and the names with 722 bytes (688+20+4+4+6). Finally, the total size is obtained by multiplying the line size by 75 (number of lines). This algorithm can be reused to calculate sizes of other data stores giving us 58,200 bytes in total on a RDBMS (75*(4+20) for Train, 100*(4+20) for Metro, 75*1*20*(4+4+4) for Tstat, 100*1*20*(4+4+4) for Mstat, and 500*(4+20) for Station) and 19,000 bytes on the Key–value store 30*(20+(20*20)) for Route and 300*(4+20) for Bus). When it comes to multipliers, only the document store and the RDBMS tables with foreign keys should be considered. Thus, sid has a multiplier of 15 on $Tstat$ (3*5) and 30 on $Mstat$ (3*10) and 3 on document store collection due to each station belonging to 3 lines and each line having 5 trains and 10 metros. Finally, using Algorithm 5, we can calculate the access frequencies of sid and $MetrosTrams$ collection (assuming both Q1 and Q2) as 0.25 and 0.75 as shown in Table 4. Similarly, the RDBMS tables Station, Tstat, and Train will be accessed with frequencies of 0.063, 0.4687, and 0.4687 respectively.

7. Related work

There are few polyglot systems already available to support heterogeneous NoSQL systems. In BigDAWG [2], different data models, including relation, array, graph, stream, and text, 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 [1,15] 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 [16] 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.

[4] http://www.hypergraphdb.org.

Myria [17] is a federated data analytics system that allows expressing complex data analytic processes using its own hybrid language MyriaL. The Relational Algebra Compiler (RACO) is used as the federated query executor, which uses relational algebra extended with imperative constructs to capture the semantics of non-relational concepts such as arrays. It generates query plans for a specific array, graph, and key–value engines. Apache Drill [18] is a distributed query engine for ad-hoc analysis. It supports file-based, document, relational, and columnar based storage systems. It used an in-memory columnar data representation based on JSON and Parquet, which allows flexible schema management. CloudMdsQL [19] is a scalable SQL query engine with extended capabilities to query non-relational data stores. It uses a SQL based query language with embedded subqueries native to the underlying data stores. A comprehensive analysis of BigDawg, Myria, Apache Drill, and CloudMdsQL is carried out in [20] comparing the different systems in terms of heterogeneity, transparency, optimally, flexibility, and autonomy. The work concludes by stating that none of the systems nor approaches are better than the other. Different systems are performant on different aspects according to the design trade-offs that they have made.

SQL++ [21] 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. [22] uses a similar approach to SQL++. Katpathikotakis et al. [23] 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 adaptors or drivers for heterogeneous data stores is a common approach used in polyglot systems. The systems, as mentioned earlier, use the same principle. Liao et al. [24] use an adaptor-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 a three-mode query approach that provides different policies on how applications access the data. The Spring framework [25] is one of the most popular software used to access multiple data stores, as it supports different types by using specific drivers and a common access interface. Apache Gremlin [26] and Tinkerpop[5] follow a similar approach but particularly for graph data stores. The main drawback of 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[6] (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 by default but instead handles the documents themselves but, there is a built in schema and data type validation.[7] Some work also has been carried out in managing document store schema externally [27].

Several works have been carried out on data design methodologies for NoSQL systems. NoSQL abstract model (NoAM) [12,28] 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. Our metamodel is capable of representing the same data on different data stores having different schemas and their overlaps (essential for the catalog) which NoAM cannot directly represent. Moreover, NoAM does not discuss multiple levels of nesting which is essential in representing JSON. Similarly, a general approach for designing a NoSQL system for analytical workloads has been presented in [29]. 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 hyper nodes, it is replicated in each of them. Mortadelo [30] introduces a model-driven database design process to automatically generate a concrete NoSQL database system from a high-level conceptual model. The platform-independent Generic Data Metamodel (GDM) is used to represent not only structural data but also the data access patterns. Then, applying a set of transformation rules, the logical NoSQL specification is generated for specific data store models (column family and document stores). Next, a set of implementation scripts are generated for the target technology of the data store. Mortadelo shows improvements over state of the art by comparison on different use cases in document and column stores.

The Concept and Object Modeling Notation (COMN) introduced in [31] covers the full spectrum of not only the datastore 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 ER diagrams fail in representing certain situations, such as nesting.

8. 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

5 http://tinkerpop.apache.org.

6 https://cwiki.apache.org/confluence/display/Hive/HCatalog.

7 https://docs.mongodb.com/manual/core/schema-validation.

store models. Next, we implemented a simple use case to showcase our approach on an existing polyglot system. Next, we enhanced the metadata storage by including statistical information enabling storage size estimation and identifying the physical query access patterns. We showed the expressiveness of hypergraphs, which is the essential feature of a canonical model of a federated system [7]. Then, effectively introduced a metadata management approach for polyglot systems leveraging this expressiveness.

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. Together with a cost estimator for the designs that could incorporate the storage metadata from the metamodel, it will be possible to predict the query performance of alternative designs on NoSQL systems. This will be an initial step towards a cost-based schema design for NoSQL systems, moving away from the current rule-based schema design.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

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

References

[1] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Manolescu, S. Zampetakis, Flexible hybrid stores: Constraint-based rewriting to the rescue, in: IEEE 32nd Int. Conf. on Data Engineering, ICDE, 2016.
[2] J. Duggan, S. Zdonik, A.J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, The BigDAWG polystore system, ACM SIGMOD Rec. 44 (2) (2015) arXiv:1609.07548.
[3] H. Garcia-Molina, J. Ullman, J. Widom, Database Systems: The Complete Book, Pearson Education India, 2008.
[4] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to NoSQL systems, Inf. Syst. 43 (2014).
[5] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to non-relational database systems: The SOS platform, in: International Conference on Advanced Information Systems Engineering, CAiSE, 2012.
[6] M. Hewasinghage, J. Varga, A. Abelló, E. Zimányi, Managing polyglot systems metadata with hypergraphs, in: Int. Conf. on Conceptual Modeling, ER, 2018, pp. 463–478.
[7] F. Saltor, M. Castellanos, M. García-Solaco, Suitability of data models as canonical models for federated databases, ACM Sigmod Rec. 20 (4) (1991).
[8] G. Klyne, J.J. Carroll, Resource Description Framework (RDF): Concepts and abstract syntax, Accessed: 2018-02-16, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.
[9] P. Čontoš, M. Svoboda, JSON schema inference approaches, in: Advances in Conceptual Modeling, 2020, pp. 173–183.
[10] J. Euzenat, P. Shvaiko, et al., Ontology Matching, 18, Springer, 2007.
[11] P. Shvaiko, J. Euzenat, Ontology matching: state of the art and future challenges, IEEE Trans. Knowl. Data Eng. 25 (1) (2013).
[12] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, Data modeling in the NoSQL world, Comput. Stand. Interfaces (2016).
[13] M. Hewasinghage, A. Abelló, J. Varga, E. Zimányi, Docdesign: Cost-based database design for document stores, in: SSDBM, 2020.
[14] M. Hewasinghage, A. Abelló, J. Varga, E. Zimányi, A cost model for random access queries in document stores, IEEE Trans. Knowl. Data Eng (Under Review).
[15] F. Bugiotti, D. Bursztyn, U.C.S. Diego, I. Ileana, Invisible glue : Scalable self-tuning multi-stores, in: CIDR, 2015.
[16] R. Sellami, S. Bhiri, B. Defude, Supporting multi data stores applications in cloud environments, IEEE Trans. on Serv. Comput. 9 (1) (2016).
[17] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, et al., The myria big data management and analytics system and cloud services, in: CIDR, 2017.
[18] M. Hausenblas, J. Nadeau, Apache drill: interactive ad-hoc analysis at scale, Big data 1 (2) (2013) 100–104.
[19] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, J. Pereira, The CloudMdsQL multistore system, in: SIGMOD, 2016, pp. 2113–2116.
[20] R. Tan, R. Chirkova, V. Gadepally, T.G. Mattson, Enabling query processing across heterogeneous data models: A survey, in: IEEE Int. Conf. on Big Data, 2017, pp. 3211–3220.
[21] K.W. Ong, Y. Papakonstantinou, R. Vernoux, The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-hadoop, NoSQL and NewSQL databases, 2014, CoRR, abs/1405.3631, arXiv:1405.3631.
[22] Á. Vathy-Fogarassy, T. Hugyák, Uniform data access platform for SQL and NoSQL database systems, Inf. Syst. 69 (2017).
[23] M. Karpathiotakis, I. Alagiannis, A. Ailamaki, Fast queries over heterogeneous data through engine customization, Proc. of the VLDB Endow. 9 (12) (2016).
[24] Y.T. Liao, J. Zhou, C.H. Lu, S.C. Chen, C.H. Hsu, W. Chen, M.F. Jiang, Y.C. Chung, Data adapter for querying and transformation between SQL and NoSQL database, Future Gener. Comput. Syst. 65 (2016).
[25] R. Johnson, J. Hoeller, K. Donald, M. Pollack, et al., The spring framework–reference documentation, Interface 21 (2004).
[26] M.A. Rodriguez, The Gremlin graph traversal machine and language, in: 15th Symposium on Database Programming Languages, ACM, 2015.
[27] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, C. Wang, Schema management for document stores, PVLDB 8 (9) (2015).
[28] F. Bugiotti, L. Cabibbo, P. Atzeni, R. Torlone, Database design for NoSQL systems, in: Int. Conf. on Conceptual Modeling, ER, 2014.
[29] V. Herrero, A. Abelló, O. Romero, NoSQL design for analytical workloads: Variability matters, in: 35th Int. Conf. Conceptual Modeling, ER, 2016.
[30] A. de la Vega, D. García-Saiz, C. Blanco, M.E. Zorrilla, P. Sánchez, Mortadelo: Automatic generation of NoSQL stores from platform-independent data models, Future Gener. Comput. Syst. 105 (2020) 455–474.
[31] T. Hills, NoSQL and SQL Data Modeling, Technics Publications, 2016.

Moditha Hewasinghage is currently working towards the Ph.D. degree at Politècnica de Catalunya, BarcelonaTech, inside the IT4BI-DC Erasmum Mundus Joint Doctorate. He received a joint M.Sc. degree from Université libre de Bruxelles, Université François Rabelais and CentraleSupélec (2017). His current research is focused on data modeling, query optimization, cost models, and document stores.

Alberto Abelló is an Associate professor. Ph.D. in Informatics, UPC. Local coordinator of the Erasmus Mundus Ph.D. program IT4BI-DC. Active researcher with more than 100 peer-reviewed publications and H-factor of 29, his interests include Data Warehousing and OLAP, Ontologies, NOSQL systems and BigData management. He has served as Program Chair of DOLAP and MEDI, being member also of the PC of other database conferences like DaWaK, CIKM, VLDB, etc.

Jovan Varga is a data engineer and researcher focusing on Big Data and Data Science areas. He holds a joint Ph.D. degree from Universitat Politècnica de Catalunya and Aalborg University. His research interests include semantic web, next generation business intelligence and document stores, and he has several publications related to these fields.

Esteban Zimányi is a professor and the director of the Department of Computer and Decision Engineering (CoDE) at Université libre de Bruxelles. He is Editor-in-Chief of the Journal on Data Semantics published by Springer. His current research interests include data warehouses, spatio-temporal databases, geographic information systems, and semantic web.