RESEARCH

Open Access

A universal approach for multi-model schema inference

Pavel Koupil*, Sebastián Hricko and Irena Holubová

*Correspondence:
pavel.koupil@matfyz.cuni.cz

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

The variety feature of Big Data, represented by *multi-model data*, has brought a new dimension of complexity to all aspects of data management. The need to process a set of distinct but interlinked data models is a challenging task. In this paper, we focus on the problem of inference of a schema, i.e., the description of the structure of data. While several verified approaches exist in the single-model world, their application for multi-model data is not straightforward. We introduce an approach that ensures inference of a common schema of multi-model data capturing their specifics. It can infer local integrity constraints as well as intra- and inter-model references. Following the standard features of Big Data, it can cope with overlapping models, i.e., data redundancy, and it is designed to process efficiently significant amounts of data.To the best of our knowledge, ours is the first approach addressing schema inference in the world of multi-model databases.

Keywords: Multi-model data, Schema inference, Cross-model references, Data redundancy

Introduction

The knowledge of a schema, i.e., the structure of the data, is critical for its efficient processing. We can distinguish *schema-full*, *schema-less*, and *schema-mixed* database management systems (DBMSs), where the schema definition is required, ignored, or can be only partial. However, despite the specification of a schema when storing the data (i.e., the so-called *schema-on-write* approach) is not required in some systems, the knowledge of the structure of the data is needed when the data is to be processed, i.e., the so-called *schema-on-read* approach is still essential. Hence, when the user does not define the schema, it needs to be extracted from the data.

The problem of *inference of a schema* for a given data has been studied for several years, mainly for XML [1] and JSON [2], i.e., the document model, which has the richest structure among the current common models. For XML documents, where the order of elements is significant, the respective schemas involve regular expressions that describe the structure of the data. According to the Gold's theorem [3] regular languages are not identifiable only from positive examples (i.e., sample XML documents), so either heuristics [4, 5] or a restriction to an *identifiable* subclass of regular languages [6] is applied.

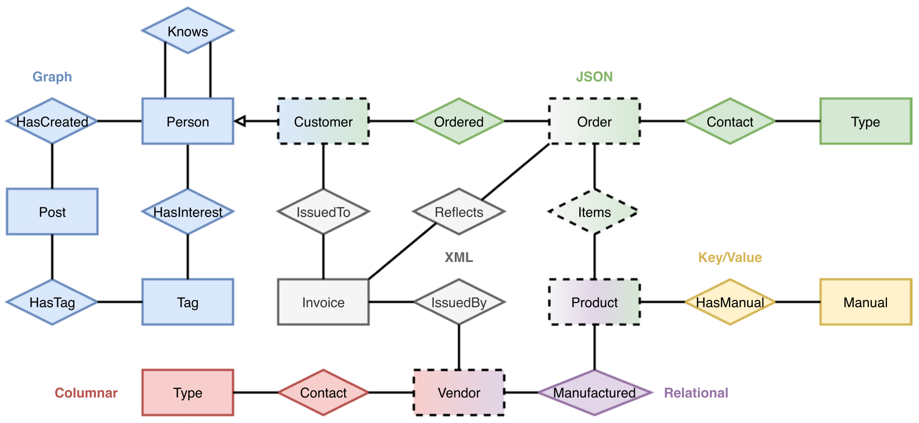


Fig. 1 Extended *UniBench* multi-model scenario

Newer approaches for currently popular JSON format, where the order is not captured in the schemas and, thus, the inference process is in this manner less complex, focus mainly on schema inference for Big Data [7, 8]. However, the ***volume*** of Big Data is not its only challenge. The ***variety*** feature represented by the *multi-model data* adds a new dimension of complexity—the need to process a set of distinct but interlinked data models.

Example 1.1 Figure 1 provides an example of a scenario inspired by the multi-model benchmark *UniBench*[1]. It depicts an ER model where we omit attributes, identifiers, and cardinalities for the sake of simplicity[2]. The colors denote the particular logical models in which the respective part of the ER model is represented—blue graph, violet relational, yellow key/value, and two document models, green JSON and gray XML. The example represents an e-shop where customers, members of a social network capturing mutual acquaintance, order products from various vendors.

At the logical level, the transition between two models can be expressed either via (1) *inter-model references* or by (2) *embedding* one model into another (such as, e.g., columns of type *JSONB* in relational tables of *PostgreSQL*[3]). Another possible combination of models is via (3) *cross-model redundancy*, i.e., storing the same data fragment in multiple models.

In the case of multi-model data, the problem of schema inference is further complicated by contradictory features of the combined models (such as structured vs semi-structured, aggregate-oriented vs aggregate-ignorant, order-preserving vs order ignoring etc.), inter-model references and cross-model integrity constraints (ICs) in general, the existence of a (partial) schema in schema-full/schema-mixed systems preserving the data, or cross-modal redundancy. Besides, there are verified single-model approaches that, however, naturally target only specifics of the particular data model. And, last

[1] http://udbms.cs.helsinki.fi/?projects/ubench.

[2] The full model will be provided in the following examples.

[3] https://www.postgresql.org/.

but not least, the question is how to represent the resulting multi-model schema, i.e., whether to choose one of the models (and which one) or whether a more abstract representation, such as UML [9], is a better choice.

To address the key indicated problems, we extend our previous research results both in the area of inference of an XML schema [4, 5] and unified management of multi-model data [10, 11]. We propose a novel approach capable of inference of a schema for a given set of multi-model data. The main contributions are as follows:

- In the proposed approach, we support all popular data models (relational, array, key/value, document, column, graph, and RDF) and all three types of their combination (embedding, references, and redundancy).
- We can cover schema-less, schema-mixed, and schema-full systems, i.e., if needed, we can re-use an existing schema both user-defined or inferred using a verified single-model approach.
- We support both local integrity constraints (e.g., unique or primary key) and global integrity constraints, i.e., intra-model and inter-model references.
- We introduce two versions of the approach–record-based and property-based–and experimentally verify their appropriateness for structurally different data.
- Following the current trends, the approach is designed to be parallelizable and, thus, scalable for Big Data.
- The proposed approach was implemented as a tool called *MM-infer*[4] [12], i.e., the proof of the proposed concept.

Outline The rest of the paper is structured as follows: In "Related work" section, we overview related work and motivate the proposed approach. In "Data models and their unification" section, we discuss the currently popular data models, their specifics, and the respective influence on schema inference. In "Multi-model schema inference" section, we describe in detail the proposed approach. "Architecture and implementation" section, describes the architecture and implementation details of *MM-infer* and "Experiments" section, introduces results of experiments. In "Conclusion and future work" section, we conclude and outline future work.

Related work

Several papers currently deal with the inference of a schema for a given set of sample data. We can divide them into approaches inferring (1) structural and (2) semantic schema. The approaches focus mainly on the document model expressed using XML or JSON in the former case. The critical difference is whether the order of child properties is significant or not. And in addition, since the JSON documents are closely related to NoSQL databases and Big Data, the approaches often support scalable processing, i.e., they can be parallelized. In the latter case of inference of a semantic schema, the aim is different. The approaches focus on the inference of a schema describing the semantics of the information stored in the data, usually expressed in RDF [13], but not its logical structure within a selected data model. Since this is not our current main target, we refer an interested reader to a recent extensive survey in Ref. [14].

[4] https://www.ksi.mff.cuni.cz/~koupil/mm-infer/.

XML schema inference An extensive comparison of XML schema inference approaches can be found in [15]. The approaches are older, reflecting the decreasing popularity of XML with the arrival of Big Data and JSON. They can be classified according to various criteria such as, e.g., the type of the result (i.e., the language used), the way it is constructed, the inputs used, etc.

Heuristic approaches [16–21] are based on experience with manual construction of schemas. Their results do not belong to any specific class of XML grammars, and they are based on the generalization of a trivial schema using a set of predefined heuristic rules, such as, e.g., "if there are more than three occurrences of an element, it is probable that it can occur arbitrary times". These techniques can be further divided into methods that generalize the trivial schema until a satisfactory solution is reached [17, 18, 20] and methods that generate a considerable number of candidates and then choose the best one [19]. While in the first case, the methods are threatened by a wrong step which can cause the generation of a suboptimal schema. In the latter case, they have to cope with space overhead and specify a proper function for the evaluation quality of the candidates. A special type of heuristic methods are so-called *merging state algorithms* [18, 20]. They are based on the idea of searching a space of all possible generalizations, i.e., XML schemas, of the given XML documents represented using a prefix tree automaton. By merging its states and thus generalizing the automaton, they construct the sub-optimal solution. Since the space is theoretically infinite, only a proper subspace of possible solutions is searched using various heuristics.

On the other hand, methods based on *inferring of a grammar* [22–28] exploit the theory of languages and grammars and thus ensure a certain degree of quality of the result. We can view an XML schema as grammar and an XML document valid against the schema as a word generated by the grammar. Although grammars accepting XML documents are, in general, context-free [29], the problem can be reduced to inferring a set of regular expressions, each for a single element (and its subelements). But, since, according to Gold's theorem [3] regular languages are not identifiable only from positive examples (i.e., sample XML documents which should conform to the resulting schema), the existing methods exploit various other information such as, e.g., the predefined maximum number of nodes of the target automaton, restriction to an identifiable subclass of regular languages, etc.

JSON schema inference The current popular JSON schema inference approaches are described and compared in Ref. [30]. Paper [31] statically compares several schema extraction algorithms over multiple NoSQL stores.

Paper [32] presents an approach for inferring versioned schemas from document NoSQL databases based on the *Model-Driven Engineering* (MDE) along with sample applications created from such inferred schemas. This research is furthered by dissertation thesis [31] and by paper [33] who tackle the issues of visualization of schemas of *aggregate-oriented* NoSQL databases and propose desired features that should be supported in visualization tools. Most recently, Fernandez et al. expand upon the meta-model from paper [32] by introducing a unified meta-model capable of modeling both NoSQL and relational data [34].

Authors of Ref. [35] propose an approach to extract a schema from JSON data stores, measuring the degree of heterogeneity in the data and detecting structural outliers. They also introduce an approach for reconstructing schema evolution history of *data lakes* [36]. Additionally, *jHound* [37], a JSON data profiling tool is presented, which can be used to report key characteristics of a dataset, find structural outliers, or detect documents violating best practices of data modelling. Finally, *Josch* [38] is a tool that enables NoSQL database maintainers to extract a schema from JSON data more efficiently, refactor it, and then validate it against the original dataset.

Authors of Ref. [39] propose a distributed approach for parameterized schema inference of massive JSON datasets and introduce a simple but expressive JSON type language to represent the schema.

Paper [40] provides an MDE-based approach for discovering schema of multiple JSON web-based services. Later its authors put it into practice as a web-based application along with a visualization tool [41].

Last but not least, Frozza et al. introduce a graph-based approach for schema extraction of JSON and BSON[5] document collections [42] and another inference process for columnar NoSQL databases [43], specifically HBase[6].

Summary As we can see, the amount of related work is significant, and there exist approaches focusing on many specifics of schema inference. However, to the best of our knowledge, currently, there exists no approach that deals with the inference of a multi-model schema. At first sight, the single-model approaches can be reused. However, this idea is not that straightforward. As we will show in the following sections, the particular models have distinct, even contradictory features, so first, a way to unify them must be found. Another complication is the mutual references and redundancy that need to be considered.

Data models and their unification

In the rest of our work, we consider the following currently popular data models: relational, array, key/value, document, column, graph, and RDF, i.e., we support all currently popular structured and semi-structured data to cover all combinations of models used in the existing popular multi-model systems[7]. First, we provide a brief overview of their features. Next, we discuss their unification to simplify and clarify the further explanation of the proposal.

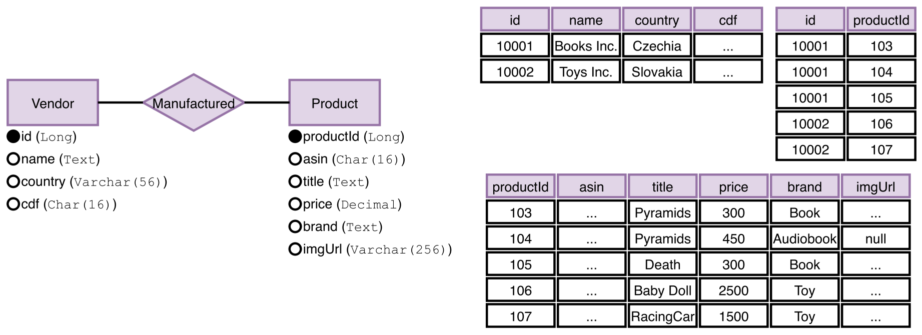
Overview of models

From the structural point of view, which is our main target, the core classification is based on the complexity of the supported data structures. *Aggregate-oriented models* (key/value, document, and column) primarily support the data structure of an *aggregate*, i.e., a collection of closely related (semi-)structured objects we want to treat as a unit. In the traditional relational world, we would speak about data de-normalization. On the contrary, *aggregate-ignorant models* (relational, array, graph, and RDF) are not primarily oriented to the support of aggregates. The relational world strongly emphasizes the normalization of structured data, whereas the graph model is, in principle, a set of flat objects mutually linked by any number of edges.

[5] https://bsonspec.org.

[6] http://hbase.apache.org.

[7] https://db-engines.com/en/ranking.

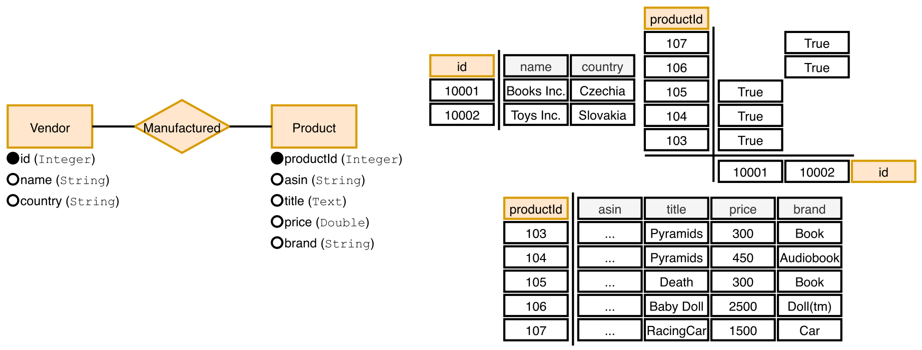


id	name	country	cdf
10001	Books Inc.	Czechia	...
10002	Toys Inc.	Slovakia	...

id	productId
10001	103
10001	104
10001	105
10002	106
10002	107

productId	asin	title	price	brand	imgUrl
103	...	Pyramids	300	Book	...
104	...	Pyramids	450	Audiobook	null
105	...	Death	300	Book	...
106	...	Baby Doll	2500	Toy	...
107	...	RacingCar	1500	Toy	...

Fig. 2 An example of relational data



id	name	country
10001	Books Inc.	Czechia
10002	Toys Inc.	Slovakia

productId	10001	10002
107		True
106		True
105	True	
104	True	
103	True	

productId	asin	title	price	brand
103	...	Pyramids	300	Book
104	...	Pyramids	450	Audiobook
105	...	Death	300	Book
106	...	Baby Doll	2500	Doll(tm)
107	...	RacingCar	1500	Car

Fig. 3 An example of array data

Relational model Relational model is based on the mathematical term *relation*, i.e. a subset of Cartesian product. The data are logically represented as tuples forming relations. Each tuple in a relation is uniquely identified by a *key*. A part of the *Structured Query Language* (SQL) [44], called *Data Definition Language* (DDL), is denoted for the definition of a relational schema, i.e., the names of relation, names of attributes, their domains (simple data types), and integrity constraints (i.e., keys, foreign keys etc.).

Example 3.1 In Fig. 2 we can see an example of data from the relational model, namely the one implemented in *PostgreSQL* as reflected by the particular data types. On the left we can the respective part of ER model from Fig. 1 and its transformation to three respective tables (relations) *Vendor*, *Product*, and *Manufactured* with the respective columns.

Array model The array model works with the notion of *multi-dimensional array* being represented as a mapping from a set of dimensions to a set of attributes. In this sense, the relational model represents the case of one dimension, i.e., the identifier (index) of a particular tuple of a relation, or two dimensions corresponding to an identifier of a tuple and a particular attribute. Also, in this case, the DDL specifies the structure of the arrays, i.e., their names, the domains, ranges, and steps of dimensions, the names and domains of attributes, and respective integrity constraints.

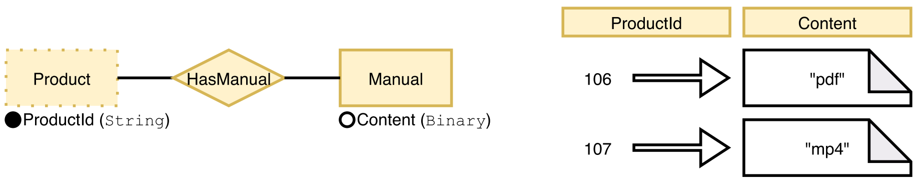


Fig. 4 An example of key/value data

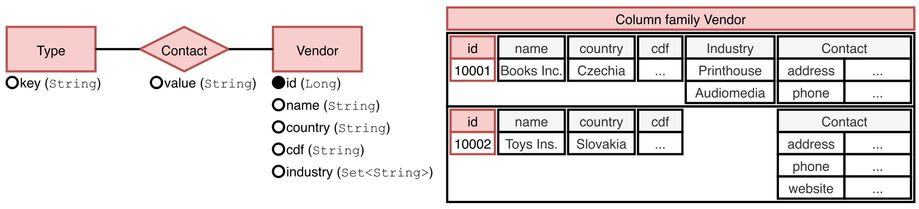


Fig. 5 An example of column data

Example 3.2 In Fig. 3 we can see how the the same part of ER schema used in Fig. 2 would be transformed to the array model. While one-dimensional arrays *Vendor* and *Product* have the same structure, two-dimensional array *Manufactures* occupies much more space.

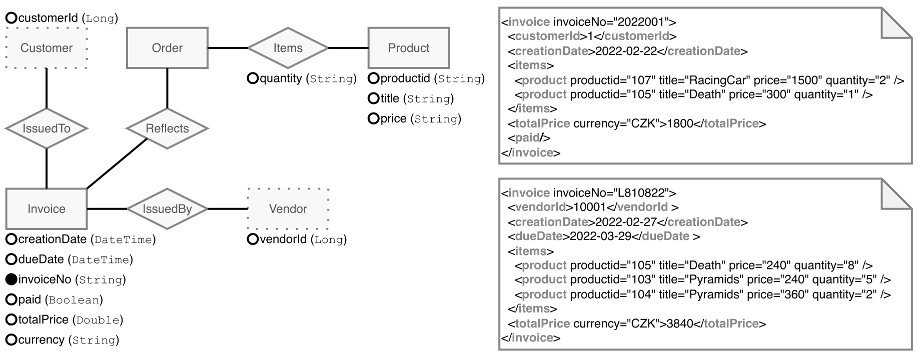
Key/value model The key/value model is the simplest aggregate-oriented data model. It corresponds to associative arrays, dictionaries, or hashes. Each record in the key/value model consists of an arbitrary schema-less value and its unique key, enabling storing, retrieving, or modifying the value.

Example 3.3 In Fig. 4 we can see an example of key/value data. Both the ER model on the left and visualization of the data on the right depict the simplicity of the model—the identifier *ProductID* and respective unstructured (binary) data content denoted as *Content*.

Column model The (wide) column model can be interpreted as a two-dimensional key/value model. It consists of the notions of a column family (table), a row, and a column. However, unlike the relational model, each row of a column family (table) can have different columns (having different names and/or data types). In other words, each row is a set of key/value pairs independent of other rows of the same column family. In some wide column systems, such as *Cassandra*[8], it is possible to specify (a part of) a schema of column families. Usually, a set of optional/compulsory columns is common for rows of the column family, whereas others can be arbitrary. If only a part of the schema can be specified, we speak about schema-mixed systems.

Example 3.4 In Fig. 5, we can see sample column data, namely the approach used in *Cassandra*, corresponding to the respective part of ER model on the left. Each row in the column family on the right is identified using column *id* and further contains three

[8] https://cassandra.apache.org/_/index.html.



```xml
<invoice invoiceNo="2022001">
  <customerId>1</customerId>
  <creationDate>2022-02-22</creationDate>
  <items>
    <product productid="107" title="RacingCar" price="1500" quantity="2" />
    <product productid="105" title="Death" price="300" quantity="1" />
  </items>
  <totalPrice currency="CZK">1800</totalPrice>
  <paid/>
</invoice>
```

```xml
<invoice invoiceNo="L810822">
  <vendorId>10001</vendorId >
  <creationDate>2022-02-27</creationDate>
  <dueDate>2022-03-29</dueDate >
  <items>
    <product productid="105" title="Death" price="240" quantity="8" />
    <product productid="103" title="Pyramids" price="240" quantity="5" />
    <product productid="104" title="Pyramids" price="360" quantity="2" />
  </items>
  <totalPrice currency="CZK">3840</totalPrice>
</invoice>
```

Fig. 6 An example of document data expressed in the XML format

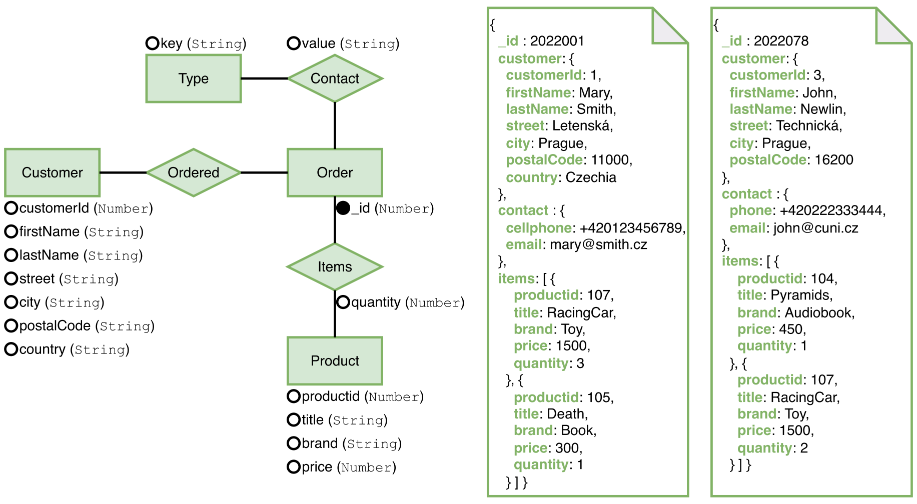
columns *name*, *country*, and *cdf* of type *String*. Next, it contains column *Industry* of complex type *Set<String>* (i.e., a set of strings) and column *Contact* of type *Map* which is supported by *Cassandra*. Since the column *Industry* is not compulsory, the respective value is missing in some rows.

Document model The document (semi-structured) model is based on the idea of representing the data without an explicit and separate definition of its schema. Instead, the particular pieces of information are interleaved with structural/semantic tags that define their structure, nesting etc.

The XML is a human-readable and machine-readable markup language. The data are expressed using elements delimited by tags containing simple text, subelements, or their combination. Additional information can be stored in the attributes of an element. Standard languages like *Document Type Definition* (DTD) [1] or *XML Schema* [45, 46] enable to specify the structure of XML documents using regular expressions.

Example 3.5 Figure 6 represents an example of document data expressed in XML. As we can see, the structure of invoices can differ depending on whether the invoice is for a customer or for a vendor. The XML document with root element *invoice* identified by attribute *invoiceNo* contains identification of the respective customer (element *customerId*) or vendor (element *vendorID*), date of creation (element *creationDate*), due date (optional element *dueDate* present if the invoice is not paid yet), list of ordered items (subelements *product* of element *items*), total price of the order (element *totalPrice*), and an indication whether it was already paid (optional element *paid*). So, there can exist several structurally different version of an invoice in the document collection depending on their status.

The JSON is a human-readable open-standard format. It is based on the idea of an arbitrary combination of three basic data types used in most programming languages – key/value pairs, arrays, and objects. Contrary to XML, the specific order of items in a JSON document is not essential. *JSON Schema* [47] language enables to specify the structure of JSON documents.



```
{
  _id : 2022001
  customer: {
    customerId: 1,
    firstName: Mary,
    lastName: Smith,
    street: Letenská,
    city: Prague,
    postalCode: 11000,
    country: Czechia
  },
  contact : {
    cellphone: +420123456789,
    email: mary@smith.cz
  },
  items: [ {
      productid: 107,
      title: RacingCar,
      brand: Toy,
      price: 1500,
      quantity: 3
    }, {
      productid: 105,
      title: Death,
      brand: Book,
      price: 300,
      quantity: 1
  } ] }
```

```
{
  _id : 2022078
  customer: {
    customerId: 3,
    firstName: John,
    lastName: Newlin,
    street: Technická,
    city: Prague,
    postalCode: 16200
  },
  contact : {
    phone: +420222333444,
    email: john@cuni.cz
  },
  items: [ {
      productid: 104,
      title: Pyramids,
      brand: Audiobook,
      price: 450,
      quantity: 1
    }, {
      productid: 107,
      title: RacingCar,
      brand: Toy,
      price: 1500,
      quantity: 2
  } ] }
```

Fig. 7 An example of document data expressed in the JSON format

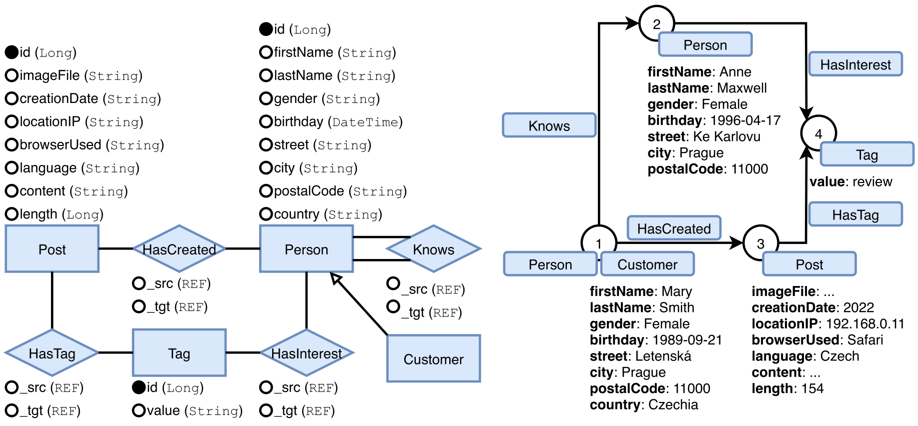
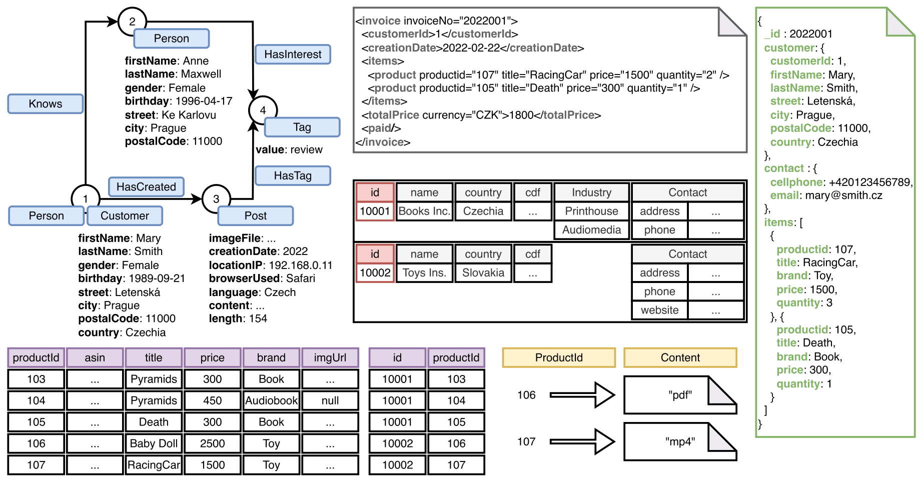


Fig. 8 An example of graph data

Example 3.6 In Fig. 7 we can see an example of document data expressed in JSON. Again, we can see two structurally different documents belonging to the same collection, this time describing orders. The order is identified using a simple property *_id* and further contains embedded documents *customer*, *contact*, and *Items*. Note that some information about customers and products may be stored redundantly in each order where the customer or product appears. In addition, since the property *contact* represents a map which is not supported in JSON, it corresponds to a set of optional properties.

Graph model The graph data model is based on the mathematical definition of a *graph*, i.e., a set of vertices (nodes) V and edges E corresponding to pairs of vertices from V. Nodes and edges are assigned with attributes, each having a name and domain (simple type). In addition, both nodes and edges have their type, enabling to group nodes/edges that represent the same piece of reality. However, the schema of nodes/edges does not (or even is not expected to) be defined.



```
<invoice invoiceNo="2022001">
  <customerId>1</customerId>
  <creationDate>2022-02-22</creationDate>
  <items>
    <product productid="107" title="RacingCar" price="1500" quantity="2" />
    <product productid="105" title="Death" price="300" quantity="1" />
  </items>
  <totalPrice currency="CZK">1800</totalPrice>
  <paid/>
</invoice>
```

id	name	country	cdf	Industry	Contact	
10001	Books Inc.	Czechia	...	Printhouse	address	...
				Audiomedia	phone	...

id	name	country	cdf	Contact	
10002	Toys Ins.	Slovakia	...	address	...
				phone	...
				website	...

```
{
  _id : 2022001
  customer: {
    customerId: 1,
    firstName: Mary,
    lastName: Smith,
    street: Letenská,
    city: Prague,
    postalCode: 11000,
    country: Czechia
  },
  contact : {
    cellphone: +420123456789,
    email: mary@smith.cz
  },
  items: [
    {
      productid: 107,
      title: RacingCar,
      brand: Toy,
      price: 1500,
      quantity: 3
    }, {
      productid: 105,
      title: Death,
      brand: Book,
      price: 300,
      quantity: 1
    }
  ]
}
```

productId	asin	title	price	brand	imgUrl
103	...	Pyramids	300	Book	...
104	...	Pyramids	450	Audiobook	null
105	...	Death	300	Book	...
106	...	Baby Doll	2500	Toy	...
107	...	RacingCar	1500	Toy	...

id	productId
10001	103
10001	104
10001	105
10002	106
10002	107

ProductId	Content
106	"pdf"
107	"mp4"

Fig. 9 An example of multi-model data

Example 3.7 Figure 8 provides an example of graph data. As we can see, from the structural aspect, the model contains only nodes, edges, and attributes with simple types. In this example, the nodes correspond to ER entities (i.e., *Post*, *Tag*, *Person*, and *Customer*) and edges to respective ER relationships.

RDF model The RDF model corresponds to a directed graph composed of triple statements. The statement is represented by a node for the subject, a node for the object, and an edge that goes from the subject to the object representing their mutual relationship. Each of the three parts of the statement can be identified by a URI.

The *RDF schema* (RDFS) [48] or the *Web Ontology Language* (OWL) [49] enable to define a schema of RDF data. However, it does not express the structure of the data, but the semantics, i.e., classes to which the represented entities of the real word belong, their features and mutual relationships etc.

Multi-model data Multi-model data are in general data which are logically represented in more than one model. The currently existing multi-model DBMSs [50] differ in the strategy used to extend the original model to other models or to combine multiple models. The new models can be supported by adopting an entirely new storage strategy, an extension of the original storage strategy, a new interface, or even no change in the original storage strategy (used for trivial cases). From the logical level, the transition between two models can be expressed either via:

1. *Inter-model references*,
2. *Embedding* one model into another (e.g., columns of type JSON in tables of the relational model of *PostgreSQL*[9]), or
3. *Multi-model redundancy*, i.e., storing the same data fragment in two or more distinct models, usually for efficient query evaluation.

[9] https://www.postgresql.org/.

Table 1 Unification of terms in popular models

Unifying term	Relational	Array	Graph	RDF	Key/Value	Document	Column
Kind	Table	Matrix	Label	Set of triples	Bucket	Collection	Column family
Record	Tuple	Cell	Node/edge	Triple	Pair (key, value)	Document	Row
Property	Attribute	Attribute	Property	Predicate	Value	JSON Field/XML element or attribute	Column
Domain	Data type	Data type	Data type	IRI/iteral/blank node	–	Data type	Data type
Value	Value	Value	Value	Object	Value	Value	Value
Identifier	Key	Coordinates/dimensions	Identifier	Subject	Key	JSON identifier/XML ID or key	Row key
Reference	Foreign key	–	–	–	–	JSON reference/XML keyref	–
Array	–	–	Array	–	Array	JSON array/repeating XML elements	Array
Structure	–	–	–	–	Set/ZSet / Hash	Nested document	Super column

Example 3.8 For the sake of clarity, the complete set of sample multi-model data corresponding to the ER model in Fig. 1 is provided in Fig. 9. The Fig. illustrates the need for a multi-model schema inference approach. As we can see, even this simple example depicts how hard it is to manage multi-model data and discover the overall structure, references, redundancy etc., within distinct models and their specifics.

Regarding the definition of a schema, currently there exists no standard language for expressing the structure of multi-model data. More abstract approaches, such as the *Unified Modeling Language* (UML) [9] or the *Entity-Relationship (ER) model* [51, 52], or our proposal [11] based on category theory [53] can be used.

Unification of models

Since the terminology within the considered models differs, we provide a unification used throughout the text in Table 1.

As we can see, the terminology is apparent in most cases, but some comments might be needed in specific cases. A *kind* represents a single logical class of items represented in each of the models. For instance, in the relational model, a kind corresponds to the notion of a table, whereas in the graph model, a kind corresponds to a class of nodes/labels specified by a label. A *record* then represents one particular item of a kind, e.g., a row of a table or a graph node/edge with a particular label.

A record consists of *properties* which can be either simple, i.e., having a simple scalar value, or complex, i.e., containing other properties. The complex properties enable hierarchically nested structure in the case of the document model and also the column model in some cases (e.g., in *Cassandra*, where super-columns, i.e., a two-level hierarchy of the columns, are supported). With this view, the whole kind can be treated as a single complex *top-level property*, whose name is the name of the kind and its child properties correspond to the properties of the kind. For example, in the case of the document model, there is only one such child property—the root element of the XML document or an anonymous root of the JSON document. Or, in the case of the relational model, the child properties of the top-level property correspond to the particular columns. We will use this view to simplify the inference algorithms.

Domains and *values* correspond to data types and selected values in all the models. *Identifiers* correspond to the notion unambiguously identifying particular records. *References* from one kind to another are allowed only in the relational and document model.

Last but not least, considering more complex data types, some models support *arrays* or *structures*. We distinguish between homogeneous and heterogeneous arrays. In the former case, an array should contain fields of the same type. In the latter case, which is allowed only in the document model, an array can contain fields of multiple types. Only in the case of the document model the type of an array item can be complex (i.e., represent nested documents); in all other cases, only arrays of simple (scalar) types are allowed.

Multi-model schema inference

We have to assume that the input data may be stored either in one multi-model DBMS or in a *polystore*, i.e., a set of single-model or multi-model systems. To infer a multi-model schema, we first need to process all types of input data. In addition, for some systems, a (partial) user-defined schema may exist. Or, for some models, a verified single-model inference approach may exist. These results can be integrated into the multi-model result.

First, in "Basic building blocks" section, we introduce two basic building blocks of the proposed approach:

1 A *type hierarchy* used for inference of data types and
2 A unifying representation of a single record or a possibly existing schema called *Record Schema Description*.

Next, we provide an overview of the general workflow of the algorithm in "Schema inference workflow" section. Then, we introduce in detail the inference approach, namely its record-based ("Record-based local inference algorithm" section) and property-based ("Property-based local inference algorithm" section) version. Finally, we discuss the process of inference of integrity constraints ("Gathering candidates for ICs and redundancy" section).

Basic building blocks

To be able to process all the considered data models using a single unified approach, we propose two auxiliary structures – the type hierarchy (see Type hierarchy" section) and the record schema description (see "Record Schema Description" section). The *type hierarchy* enables us to cope with the problem of distinct sets of simple types used in the models (or their system-specific implementations). The *record schema*

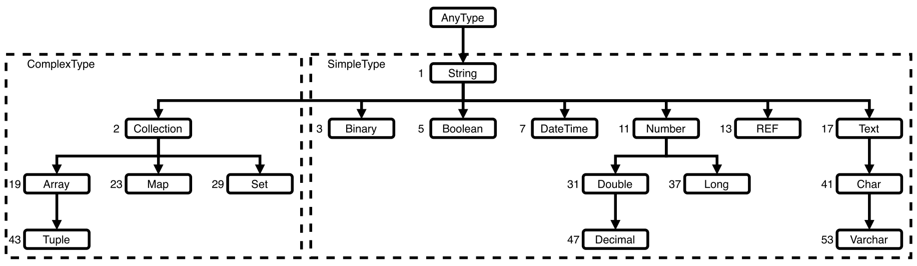


Fig. 10 An example of Type Hierarchy

description enables one to describe a schema of one kind (including the case of a schema of a single record) regardless of its model(s).

Type hierarchy

The *type hierarchy* is a simple tree-based representation of basic data types that can occur in the input models and their mutual relationships. It enables us to quickly find a common supertype that covers a set of data types even though not all of them are supported in each model.

The data types supported across the data models, forming the set $\mathbb{T}'$, are represented as vertices v_i and the natural hierarchy between the types is represented as edges $e : v_j \to v_i$, when values of v_j involve also values of v_i. For example, the fact that `String` is a generalization of `Integer` is represented as `String` $\to$ `Integer`. Additionally, we assign a unique prime p_i or number 1 to each vertex v_i. The numbers are assigned using the BFS algorithm, starting from 1 assigned to the root of the hierarchy and ensuring that $p_i < p_j$ if $e : v_i \to v_j$. The integer representation of node v_i is then computed as $T_i = \prod_{j=0}^{i} p_j$, where $p_0, \ldots, p_i$ are prime numbers (or 1 assigned to the root) assigned to vertices $v_0, \ldots, v_i$ on the path from root node v_0 to v_i. The concept of *union type UT*, is recursively defined as a union (denoted using operator $\oplus$) of types $T_i, i = 0, ..., n$, i.e., $UT := \oplus_{i=0}^{n} T_i$, where T_i is a simple type or a union type. We will denote the set of all types, i.e., both basic and union types as $\mathbb{T}$.

Next, we can introduce an associative and distributive operator *bestGeneralType*, defined as $T_i \sqcup T_j := gcd(T_i, T_j)$, where *gcd* is the greatest common divisor of prime products T_i and T_j representing the best general type of T_i and T_j.

In addition, we introduce additional associative and distributive operator *typeMerge*, denoted as $\diamond$, defined as:

- $T_i \diamond T_j := T_i \oplus T_j$ if $T_i \neq T_j$
- $UT \diamond T := UT \oplus T$ if $T \nsubseteq UT$
- $UT \diamond T := UT$ if $T \subseteq UT$
- $UT_1 \diamond UT_2 := UT_1 \oplus UT_2$ if $UT_1 \nsubseteq UT_2$
- $UT_1 \diamond UT_2 := UT_1$ if $UT_2 \subseteq UT_1$

Example 4.1 Let us illustrate an example of the type hierarchy of data types used in Fig. 1. The hierarchy illustrated in Fig. 10 is represented as a tree having an artificial root *AnyType*[10] and then the natural type hierarchy. As we can see, we do not consider the hierarchy typical for programming languages, i.e., an object being a supertype. Instead, we consider representation hierarchy, i.e., natural conversions between data types. For example, since anything can be represented as a *String*, it is the root of the tree. Additionally, *String* is assigned with 1, i.e., *String* is a supertype of all types.

As we can also see, each node is assigned with prime (in the BFS order), whereas the data type itself is represented as a product (corresponding to the path from *String* to the data type). For example, $String := 1, Tuple := 1 \times 2 \times 19 \times 43$, and $Double := 1 \times 11 \times 31$.

Additionally, in the example we also represent any *ComplexType* as an even number, because $Collection := 1 \times 2$ is a supertype of all complex types. This allows us to quickly distinguish simple and complex types using binary operations (lowest bit value test).

Example 4.2 Having the data type hierarchy from Fig. 10, we can represent a union type as a union of data types represented as products. For example:

- A union type $UT_s := Double \oplus Long$ of two simple types is computed as a simple union of products, i.e., $UT_s := (1 \times 11 \times 31) \oplus (1 \times 11 \times 37)$[11].
- A union of two complex types $UT_c := Tuple \oplus Map$ is represented as $UT_c := 2 \times 19 \times 43 \oplus 2 \times 23$.
- Finally, a union of two union types $UT_u := UT_s \oplus UT_c$ is represented as a union $UT_u := 11 \times 31 \oplus 11 \times 37 \oplus 2 \times 19 \times 43 \oplus 2 \times 23$.

Example 4.3 Computing the best general type can be done as follows:

- Having a union type of simple types $UT_s := 11 \times 31 \oplus 11 \times 37$, the best best general type is computed as $gcd(11 \times 31, 11 \times 37) = 11 \cong Number$.
- Having $UT_c := 2 \times 19 \times 43 \oplus 2 \times 23$, the best general type is computed as $gcd(2 \times 19 \times 43, 2 \times 23) = 2 \cong Collection$.
- Finally, the best general type of $UT_u := 11 \times 31 \oplus 11 \times 37 \oplus 2 \times 19 \times 43 \oplus 2 \times 23$ is $gcd(11 \times 31, 11 \times 37, 2 \times 19 \times 43, 2 \times 23) = 1 \cong String$.

Note that the implementation of finding the best general type may follow the hierarchical structure of the tree. Therefore it is not needed to compute $gcd()$ in an explicit way (with the exponential complexity of the algorithm). Instead, we traverse the tree from its root, and we try to divide all the type representations by a prime assigned to the node. If it returns an integer, we traverse deeper. Otherwise, we try the sibling nodes. Having all the siblings processed and no subtree to traverse, the *gcd* is found.

[10] In usual implementations, we consider *String* as a supertype of all data types, therefore the numbering of nodes starts from *String* instead of *AnyType* (i.e., *AnyType* is ignored).

[11] Note that parentheses can be omitted with regards to the order of operations $\times$ and $\oplus$. Also $String := 1$ can be omitted in the product.

Record schema description

The *Record Schema Description* (RSD) enables us to describe a schema of one kind regardless of its model(s). It naturally covers also the case of a *trivial schema* of a single record. So, in the proposed multi-model inference process it serves for representation of:

1. All types of *input schemas*, i.e.,
   (a) an existing user-defined schema,
   (b) a single-model schema inferred using a single-model approach, and
   (c) a *basic schema* inferred using the *local schema inferrer* in the remaining cases, i.e., for kinds without a schema,
2. *Intermediate schemas* created during the inference process by the *global schema inferrer*, and
3. The *resulting multi-model schema* that is transformed to a required output form.

The RSD has a tree structure, and it describes the structure of a property, a record, or a kind (i.e., a set of records) because all the three cases can be treated in the same way, as we have discussed above. The root of an RSD corresponds to a root property of the respective data model (e.g., the root XML element or the anonymous root of a JSON hierarchy), or it is an artificial root property with trivial settings encapsulating the properties (e.g., in the relational or graph model). An RSD describing a property (or a record or a kind) p is recursively defined as a tuple $rsd = (name, unique, share, id, types, models, children, regexp, ref)$, where:

- *name* is the name of property p extracted from the data (e.g., `_id`, `person`) or it can be anonymous (e.g., in case of items of JSON arrays or an artificial root property).
- *unique* is the IC specifying uniqueness of values of p. Its values can be `T` (true), `F` (false), or `U` (unknown) for intermediate steps of the inference process.
- $share = (share_p, share_x)$ is a tuple, where $share_p$ is the number of all occurrences of property p and $share_x$ is the number of parent properties containing p at least once. Note that $share_p > share_x$ reflects so-called *repeating* property, i.e., property forming an element of an array. Also note that if we combine $p.share_x$ with $pp.share_p$ (of any type), where pp is the parent property of p, we get the optionality of p, i.e., $p.share_x = pp.share_p$ reflects a required property, while $p.share_x < pp.share_p$ reflects an optional property.
- *id* is the IC specifying that the property is an identifier. Its values can also be `T`, `F`, or `U` with the same meaning.[12]
- *types* is a set of data types that cover the property. For a simple property it involves simple data types (i.e., `String`, `Integer`, ...). For a complex property it involves the following values:

[12] For the sake of simplicity we currently do not support composite identifiers or respective composite references.

  - `Array`, i.e., ordered (un)named (not) unique child properties (e.g., child elements in XML or items of arrays in JSON),
  - `Set`, i.e., unordered unnamed unique child properties (e.g., items of `Set` in column store *Cassandra*), and
  - `Map`, i.e., unordered named unique child properties (e.g., attributes of a relational table).

  As we will see later, in the final phase of the inference process, the set is reduced to a single resulting datatype.

- *models* is a (possibly empty) set of models (`JSON` = JSON document, `XML` = XML document, `REL` = relational, `GRAPH` = graph, `COL` = column, `KV` = key/value) that involve the property. If the set contains more than one item, it represents cross-model redundancy. If the value of *models* within a child property changes, it corresponds to embedding one model to another.
- *children* is a (possibly empty) set of recursively defined child properties.
- (Optional) *regexp* specifies a regular expression over the set *children*, or its subset (e.g., in case of schema-mixed systems or case of XML elements and attributes, forming together child properties).
- (Optional) *ref* specifies that the property references another property. Since we do not specify any restriction on the referencing and referenced property models, we also cover self-references and the third possible combination of multiple models, i.e., inter-model references.

Example 4.4 Figure 11 provides sample RSDs of kinds from Fig. 1 (having the respective colors) in their textual form. Each node is described as a tuple of values of its above-listed components (in the given order) in curly brackets. If the set *children* is not empty, in curly brackets, there occur the child properties described in the same way.

Having the unifying representation of all possible types of input data having any of the supported models, we can propose a much more straightforward multi-model inference approach. It is based on an essential feature of RSDs – the fact that two RSDs can be merged to describe a common schema of the respective kinds. The merging strategy is a part of the proposed approach (see "Merging of RSDs—function merge()" section).

Schema inference workflow

The proposed inference process takes into account the following features and specifics of the multi-model environment:

- Various aspects of the combined models and their specifics known for popular multi-model DBMSs [50] (such as sets/maps/arrays/tuples, (un)ordered properties, various treatments of missing values etc.), relational table Product

```
( _, U, (5,5), U, Map, REL, {
  ( asin, F, (5,5), F, Char(16), REL, ε, ε, ε ),
  ( brand, F, (5,5), F, Text, REL, ε, ε, ε ),
  ( imgUrl, F, (4,4), F, Varchar(256), REL, ε, ε, ε ),
  ( price, F, (5,5), F, Decimal, REL, ε, ε, ε ),
  ( productId, T, (5,5), T, Long, REL, ε, ε, ε ),
  ( title, F, (5,5), F, Text, REL, ε, ε, ε )
  }, ε, ε )
```

relational table Vendor

```
( _, U, (2,2), U, Map, REL, {
  ( cdf, F, (2,2), F, Char(16), REL, ε, ε, ε ),
  ( country, F, (2,2), F, Varchar(64), REL, ε, ε, ε ),
  ( id, T, (2,2), T, Long, REL, ε, ε, ε ),
  ( name, F, (2,2), F, Text, REL, ε, ε, ε )
  }, ε, ε )
```

graph labels Person, Customer

```
( _, U, (2,2), U, Map, GRAPH, {
  ( birthday, F, (2,2), F, DateTime, GRAPH, ε, ε, ε ),
  ( city, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( country, F, (1,1), F, String, GRAPH, ε, ε, ε ),
  ( firstName, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( gender, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( id, T, (2,2), T, Long, GRAPH, ε, ε, ε ),
  ( lastName, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( postalCode, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( street, F, (2,2), F, String, GRAPH, ε, ε, ε )
  }, ε, ε )
```

graph label Knows

```
( _, U, (1,1), U, Map, GRAPH, {
  ( _src, F, (1,1), F, REF, GRAPH, ε, ε, Person(id) ),
  ( _tgt, F, (1,1), F, REF, GRAPH, ε, ε, Person(id) )
  }, ε, ε )
```

column family Vendor

```
( _, U, (2,2), U, Map, COL, {
  ( cdf, F, (2,2), F, String, COL, ε, ε, ε ),
  ( contact, F, (2,2), F, Map, COL, {
    ( address, F, (2,2), F, String, COL, ε, ε, ε ),
    ( phone, F, (2,2), F, String, COL, ε, ε, ε ),
    ( website, F, (1,1), F, String, COL, ε, ε, ε )
    }, ε, ε ),
  ( country, F, (2,2), F, String, COL, ε, ε, ε ),
  ( id, T, (2,2), T, Long, COL, ε, ε, ε ),
  ( industry, F, (1,1), F, Set, COL, {
    ( _, F, (2,1), F, String, COL, ε, ε, ε )
    }, (_+), ε ),
  ( name, F, (2,2), F, String, COL, ε, ε, ε )
  }, ε, ε )
```

key/value pair Feedback

```
( _, U, (2,2), U, Map, KV, {
  ( content, F, (2,2), T, Binary, KV, ε, ε, ε ),
  ( productId, T, (2,2), T, String, KV, ε, ε, ε )
  }, ε, ε )
```

JSON document Order

```
( _, U, (2,2), U, Map, JSON, {
  ( _id, T, (2,2), T, Number, JSON, ε, ε, ε ),
  ( contact, F, (2,2), F, Map, JSON, {
    ( cellphone, F, (1,1), F, String, JSON, ε, ε, ε ),
    ( email, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( phone, F, (1,1), F, String, JSON, ε, ε, ε )
    }, ε, ε ),
  ( customer, F, (2,2), F, Map, JSON, {
    ( city, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( country, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( customerId, F, (2,2), F, Number, JSON, ε, ε, ε ),
    ( firstName, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( lastName, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( postalCode, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( Street, F, (2,2), F, String, JSON, ε, ε, ε ),
  }, ε, ε ),
  ( items, F, (2,2), F, Array, JSON, {
    ( _, F, (4,2), F, Map, JSON, {
      ( brand, F, (4,4), F, String, JSON, ε, ε, ε ),
      ( price, F, (4,4), F, Number, JSON, ε, ε, ε ),
      ( productId, F, (4,4), F, Number, JSON, ε, ε, ε ),
      ( quantity, F, (4,4), F, Number, JSON, ε, ε, ε ),
      ( title, F, (4,4), F, String, JSON, ε, ε, ε )
      }, ε, ε ),
    }, (_+), ε ),
  }, ε, ε )
```

XML document Invoice

```
( invoice, U, (2,2), U, Map+Array, XML, {
  ( creationDate, F, (2,2), F, DateTime, XML, ε, ε, ε ),
  ( customerId, F, (1,1), F, Long, XML, ε, ε, ε ),
  ( dueDate, F, (1,1), F, DateTime, XML, ε, ε, ε ),
  ( @invoiceNo, T, (2,2), T, String, XML, ε, ε, ε ),
  ( items, U, (2,2), F, Array, XML, {
    ( product, F, (5,2), F, Map, XML, {
      ( @price, F, (5,5), F, String, XML, ε, ε, ε ),
      ( @productId, F, (5,5), F, String, XML, ε, ε, ε ),
      ( @quantity, F, (5,5), F, String, XML, ε, ε, ε ),
      ( @title, F, (5,5), F, String, XML, ε, ε, ε )
    }, ε, ε ),
  }, (product+), ε ),
  ( paid, F, (1,1), F, Boolean, XML, ε, ε, ε ),
  ( totalPrice, F, (2,2), F, Map+Array, XML, {
    ( ___TEXT___, F, (2,2), F, Double, XML, ε, ε, ε )
    ( @currency, F, (2,2), F, String, XML, ε, ε, ε )
    }, (___TEXT___), ε ),
  ( vendorId, F, (1,1), F, Long, XML, ε, ε, ε ),
  }, ((customerId|vendorId)+, creationDate,
dueDate?, items, totalPrice, paid?), ε )
```

Fig. 11 An example of RSDs

- Local ICs, various types of redundancy (both intra-model and inter-model), and intra/inter-model references,
- (Partial) schemas required/allowed in selected models or inferred single-model schemas,
- Possible, but not compulsory user interaction involving modification of suggested candidates (for ICs, redundancy etc.) and specification of non-detected cases, and
- Processing of Big Data, i.e., maximum possible reduction of unnecessary information and parallel processing.

The input of the inference process is formed of the following:

1. A non-empty set of single/multi-model DBMSs $\mathcal{D}_1, \mathcal{D}_2, ...$ which together contain a set of kinds $\kappa_1, \kappa_2, ..., \kappa_N$. Each kind is associated with its model(s). For each model

supported in a particular DBMS $\mathcal{D}_i$ we also know whether it is schema-less/full/mixed and whether the order of sibling properties of a kind must be preserved.
2. A (possibly empty) set of predefined schemas $\sigma'_1, \sigma'_2, ..., \sigma'_n, n \leq N$, (partially) describing selected kinds.
3. A (possibly empty) set of user-specified input information which can be of the following types:
    (a) A *redundancy set of kinds* $RK = \{\kappa_1, \kappa_2, ..., \kappa_r\}, r \leq N$ which describe the same part of reality, i.e., they will have a common schema σ. (Note that there is no restriction on the models the kinds in R can have. On the other hand, we do not know the schema of all kinds at this stage, so the redundancy cannot be specified at a higher level of granularity.)
    (b) A *simple data type* assigned to a selected property.
    (c) A *local IC* assigned to a selected property. The possible constraints involve identifier, unique, or (not) null.
    (d) A *reference* represented by an ordered pair of properties where the first one is the *referencing property* and the second one is the *referenced property*.

In other words, the user specifies the data on which the inference process should be applied. Depending on the type of the database system where it is stored, i.e., its specific features, the inference process can re-use an existing user-defined schema, it knows whether the order of siblings should be preserved, etc. Eventually, at the beginning or during the inference process (see "Architecture and implementation" section), the user can provide a partial inferred schema, user-specified simple data types, ICs, and references for selected properties, as well as redundantly stored kinds[13].

The general workflow of the inference process has two main phases—local and global. In the local phase, the process assumes as the input a large set of data, and the task is to reduce the information in parallel efficiently, i.e., we infer basic local schemas for each kind. The aim of the global phase is to merge the local schemas and enrich them with additional information gathered from the input data (i.e., ICs and references). Since we can assume that the number of all kinds in the whole multi-model schema is several orders smaller than the amount of input data, this phase does not need to be parallelised.

The workflow consists of the following stages:

1. *Local schema inferrer* For each kind κ we generate its local RSD as follows:
    (a) If κ has a predefined schema σ'_κ, we transform it into RSD representation.
    (b) Otherwise, we generate for κ a *basic RSD* using a parallel approach as follows:
        (i) We generate a *trivial RSD* for each record (or property, depending on the selected type of the algorithm—see "Record-based local inference algorithm" section and "Property-based local inference algorithm" section) of κ.

[13] Note that for the sake of simplicity we currently consider redundancy only for whole kinds.

(ii) We merge (see "Merging of RSDs—function merge()" and "Forest appender—function addToForest()" section) trivial RSDs of κ and eventually all kinds in its respective redundancy set to RK_κ a basic RSD.

2. *Gathering of footprints* Parallel to the local schema inference, for each kind κ we gather its auxiliary statistics, called *footprints* (see "Gathering candidates for ICs and redundancy" section), as follows:

   (a) *Phase map* We gather footprints for each value of each property p_i^κ of κ.
   (b) *Phase reduce* We merge all footprints of values of each property p_i^κ, resulting in an aggregated footprint of property p_i^κ.
   (c) *Candidate Set* We apply a set of rules on the merged footprints of each property to produce a set of candidates for redundancy, local ICs, and references. Note that when the structure of kinds is inferred, the redundancy can be specified at the level of (complex) properties. A *redundancy set of properties* $RP = \{\pi_1, \pi_2, ..., \pi_s\}, s \in \mathbb{N}$, where each property is a part of some kind regardless its model, describes the same part of reality.
   (d) The user can confirm/refute the candidates at this stage or add new ones.

3. *Global Schema Inferrer* Having a unified RSD representation and footprints for each input kind κ, we generate the final multi-model schema as follows:

   (a) (Optionally), we perform the full check of candidates. It is not done if the user confirms the candidates.
   (b) We merge all redundancy sets of properties, i.e., for each property $\pi_i \in RP_j$, $i, j \in \mathbb{N}$ we extend its schema by joining RSDs of all properties in RP_j.
   (c) We extend the RSDs with discovered references.
   (d) We create the final multi-model schema formed by all inferred RSDs.

4. We transform the resulting set of RSDs and respective ICs to the user-required output.

Next we introduce two versions of the local inference algorithm—*record-based* and *property-based*. The former follows the usual strategy in the existing works, i.e., "horizontal" processing; the latter introduces a "vertical" optimisation for complex data.

Record-based local inference algorithm

The more intuitive approach, so-called *Record-Based Algorithm* (RBA), considers a record, i.e., the root property including all its child properties, as a working unit. The input of Algorithm 1 consists of the particular database wrapper w_D (implementing specific behaviour and features of the system D) and set N_D of names of kinds whose schemas are to be inferred. Having initiated an empty schema S (i.e., a forest of RSDs), the algorithm infers the schema of each kind κ during three logically different steps:

- *Preparation phase* The data is first loaded using a particular database wrapper w_D, and then each record is *in parallel* mapped into an RSD describing its trivial schema. The result of the preparation phase is a collection R (possibly containing duplicities) of RSDs describing schemas of individual records within kind κ.

- *Reduce phase* Next, the collection R is merged using function *merge()* (see "Merging of RSDs—function merge()" section) *in parallel* into single r_κ describing the basic RSD of kind κ.
- *Collection phase* The inferred schema r_κ is added to set S.

Having all the kinds processed, the resulting schema S is returned.

Algorithm 1: Record-Based Local Inference Algorithm

```
1  Input: $w_D$ – a wrapper for database system (or model) $D$
          $N_D$ – a set of names of kinds whose schema is to be inferred
   Output: $S$ – a minimal set of structurally different RSDs
2  Schema $S := \emptyset$
3  foreach $name_\kappa$ in $N_D$ do
       // preparation phase:
4      $R := w_D$.mapRecordsToRSDs($name_\kappa$)
       // reduce phase:
5      $r_\kappa := R$.merge()
       // schema collection phase:
6      $S$.add($r_\kappa$)
7  return $S$
```

Merging of RSDs—function merge()

During the merging process we merge the information, and we modify respectively the regular expression describing the data. In particular, having two RSDs $rsd_1 = (name_1, unique_1, share_1, id_1, types_1, models_1, children_1, regexp_1, ref_1)$ and $rsd_2 = (name_2, unique_2, share_2, id_2, types_2, models_2, children_2, regexp_2, ref_2)$, the merging process creates the resulting RSD $rsd = (name, unique, share, id, types, models, children, regexp, ref)$ as follows:

- Within local stage, names $name_1$, $name_2$ are always equal, i.e., only properties having the same name are merged, therefore $name := name_1$.
- *unique* is set to minimum value of $unique_1$, $unique_2$, where `F` < `U` < `T`. In other words, the fact that a property is not unique (i.e., either $unique_1$ or $unique_2$ is set to `F`) cannot be changed. If neither $unique_1$ nor $unique_2$ is set to `F` but at least one is set to `U`, we have to wait to finish parallel checking of the data. Otherwise, having $unique_1$ and $unique_2$ set to `T`, the resulting *unique* is set to `T`.
- $share := (share_p, share_x)$, where $share_p := share_{p_1} + share_{p_2}$ and $share_x := share_{x_1} + share_{x_2}$.
- Similarly to *unique*, the same principle applies for *id*.
- $types := types_1 \diamond types_2$ (see "Type hierarchy" section).
- $models := models_1 \cup models_2$.
- $children := children_1 \cup children_2$, whereas the child properties with the same name are recursively merged too.

- *regexp* is the result of merging of regular expressions $regexp_1$ and $regexp_2$ using existing verified schema inference approaches [4, 5].[14] If $regexp_1 = \epsilon$, then $regexp := regexp_2$. Else $regexp := regexp_1$.
- If $ref_1 = ref_2$, then $ref := ref_1$. Otherwise, it has to be resolved by the user/default settings.

Algorithm 2: Function *merge()*

```
Input: $r_1$ , $r_2$ – RSDs to be merged
Output: $r$ – the merged RSD
// the names are always equal
1  $r.name := r_1.name$
   // select minimum, i.e., F < U < T
2  $r.unique :=$ min($r_1.unique$, $r_2.unique$)
   // sum shares
3  $r.share :=$ sum($r_1.share$, $r_2.share$)
   // select minimum, i.e., F < U < T
4  $r.id :=$ min($r_1.id$,$r_2.id$)
   // type merge operator
5  $r.types := r_1.types \diamond r_2.types$
   // union of models
6  $r.models := r_1.models \cup r_2.models$
   // recursively merge children
7  $r.children :=$ mergeChildren($r_1.children$,$r_2.children$)
   // regular expression merge
8  $r.regexp :=$ mergeRegexp($r_1.regexp$, $r_2.regexp$)
   // references are the same or missing, therefore arbitrary $ref$ is selected
9  $r.ref := r_1.ref$
10 return $r$
```

Property-based local inference algorithm

Alternatively, instead of a whole record, the working unit of the local inference process can be a single property, whose eventual child properties are processed separately too. This strategy is not common in the existing approaches; however, it may lead to a performance boost (as we show in "Experiments" section) when, e.g., the record is highly structured and contains a large number of nested properties. Moreover, this version of the algorithm can be merged with the mining of footprints (see "Gathering candidates for ICs and redundancy" section) into a single algorithm.

To work with individual properties instead of records, we need to be able to distinguish their RSDs. Therefore, we introduce the notion of a *hierarchical name* that uniquely identifies each property p by concatenating the names of properties on the path from the root property to property p, where each step is separated by a delimiter '/' (slash). As the 0th and 1st steps, we use the name of the system and the kind where the property occurs. Additionally, if the property is an anonymously named element of an array, the name of the property consists of '_' (underscore) and its data type.

Example 4.5 For example, property *productId* from the JSON document model in Fig. 9 has hierarchical name: `/mongoDB/Order/items/_Object/productId`

The input of the *Property-Based Algorithm (PBA)* (see Algorithm 3) also consists of the particular database wrapper w_D and set N_D of names of kinds whose schemas are to

[14] Due to rich related work (see "Related work" section) we omit technical details.

be inferred. Having initiated an empty schema S, the algorithm processes each kind κ as follows:

- *Preparation phase* For each property p of each record, the hierarchical name of the property and the trivial RSD of the property is extracted from the data *in parallel*, forming a collection NP of pairs $(name_p, rsd_p)$. Next, the grouping according to $name_p$ is performed, resulting in the set GP of pairs $(name_p, P)$, where P is a set of RSDs of properties with the same $name_p$.
- *Reduce phase* Next, each collection P is *in parallel* aggregated using function *aggregateByHierarchicalName()* (see "Aggregating of RSDs—function aggregateByHierarchicalName()" section) into rsd_p describing the basic RSD of property p with hierarchical name $name_p$. The resulting set of pairs $(name_p, rsd_p)$ is denoted as AP.
- *Collection phase*: Set AP is iterated and each rsd_p is added into schema S, continuously enriching and building the schema of kind κ using function *addToForest()* (see "Forest appender—function addToForest()" section).

Finally, the resulting schema S is returned as the result.

Algorithm 3: Property-Based Local Inference Algorithm

```
  Input: $w_D$ – wrapper for database system (or model) $D$
1        $N_D$ – set of names of kinds whose schema is to be inferred
  Output: $S$ – a set of RSDs describing the resulting schemas of kinds in $N_D$
2 Schema $S := \emptyset$
3 foreach $name_\kappa$ in $N_D$ do
      // preparation phase:
4     $NP := W_D$.flatMapRecordsToPairs($name_\kappa$)
      // group properties with the same $name_p$ together
5     $GP := NP$.groupByKey()
      // reduce phase:
6     $AP := GP$.aggregateByHierarchicalName()
      // schema collection phase:
7     foreach $(name_p, rsd_p)$ in $AP$ do
8         addToForest($rsd_p$, $S$)
9 return $S$
```

Aggregating of RSDs—function aggregatebyhierarchicalname()

Generation of a basic (local) RSD consists of generating an RSD for each property and their aggregation into a common property schema. During the process, we aggregate the information, and we modify respectively the regular expression describing the order of the nested properties.

As we can see in Algorithm 4, having a collection of RDSs $rsd_i = (name_i, unique_i, share_i, id_i, types_i, models_i, children_i, regexp_i, ref_i)$, $i = 1, ..., n$, the aggregation process creates the resulting RSD $rsd_p = (name, unique, share, id, types, models, children, regexp, ref)$ corresponding to a schema of property p as follows:

- The names $name_i$ are always equal, i.e., only the properties having equal hierarchical name are aggregated, therefore $name := name_1$.
- *Unique* is set to minimum value of $unique_i, i = 1, ..., n$, where `F < U < T`.
- *Share* is set to the sum of shares, i.e., $share := (share_p, share_x) = (\sum_{i=1}^{n} share_{p_i}, \sum_{i=1}^{n} share_{x_i})$
- Similarly to *unique*, the same principle applies for *id*.
- $types := types_1 \cup ... \cup types_n$, whereas if there appear two types $t_i, t_j \in types$, s.t. $t_i \subset t_j$, then t_i is removed from *types*.
- $models := \bigcup_{i=1}^{n} models_i$.
- *regexp* is the result of merging regular expressions $regexp_1, \ldots, regexp_n$ using an existing verified approach.
- Within the aggregate function, *children* is always an empty set as individual children have their own hierarchical name and thus are processed separately. Its content is resolved in later stage of the algorithm within function *addToForest()* (see "Forest appender—function addToForest()" section).
- For all $ref_1, \ldots, ref_n$ either $ref_i = \epsilon$ or all the values of ref_i are equal, therefore $ref := ref_1$ is selected. If $ref = \epsilon$, the references are resolved after applying the candidates stage (see "Global phase" section).

```
Algorithm 4: Function aggregateByHierarchicalName()
Input: $P$ – Non-empty list of Property RSDs to be aggregated
Output: $rsd_p$ – resulting aggregated rsd
1  $rsd_p$ := (null, T, (0,0), T, ∅, ∅, ∅, null, null)
   // the names are always equal
2  $rsd_p.name$ := $P$.first().$name$
3  foreach property $p$ in $P$ do
       // select minimum, i.e., F < U < T
4      $rsd_p.unique$ := min($rsd_p.unique$, $p.unique$)
       // sum shares
5      $rsd_p.share$ := sum($rsd_p.share$, $p.share$)
       // select minimum, i.e., F < U < T
6      $rsd_p.id$ := min($rsd_p.id$,$p.id$)
       // type merge operator
7      $rsd_p.types$ := $rsd_p.types \diamond p.types$
       // union of models
8      $rsd_p.models$ := $rsd_p.models \cup p.models$
       // regular expression merge
9      $rsd_p.regexp$ := regexpMerge($rsd_p.regexp$, $p.regexp$)
       // $children$ are resolved later (using forest appender), the processed
          property contains $children = \emptyset$
   // references are the same or missing, therefore arbitrary $ref$ is selected
10 $rsd_p.ref$ := $P$.first().$ref$
11 return $rsd_p$
```

Forest appender—function addToForest()

The purpose of this function (see Algorithm 5) is to join RSDs describing the schema of particular properties to form an RSD corresponding to a schema of the whole kind.

Moreover, the RSDs describing a schema of a single kind are grouped into a forest. Having pairs ($name_p, rsd_p$) which are alphabetically ordered according to $name_p$ (in ascending order), the parent property is always included in the schema S before its children (note that locally the schema is a tree). If the properties are not ordered, then if any parent property is missing, we can insert an empty placeholder of the not-yet-processed parent allowing it to include its children. As soon as the parent is being processed, it replaces the placeholder.

Algorithm 5: Function *addToForest()*

```
Input: $name_p$ – hierarchical name of a property $p$
       $o_p$ – object that holds an information, e.g., an RSD corresponding to property $p$
       $O$ – resulting set of possibly interlinked hierarchical objects, e.g., a forest of RSDs
$name_\kappa := name_p.kind()$
$node := O.getOrCreateRoot(name_\kappa)$
// the first step of the hierarchical name
$name := name_p$.head()
repeat
    if $node$.hasChildren($name$) then
        $node := node$.getChildren($name$);
    else
        $node := node$.getOrCreateChildren($name$)
    if $name$ is $name_p$.tail() then
        // if $node$ represents an inner node, its content is merged
        // otherwise (leaf) its placed as is
        $node$.placeContent($o_p$)
    // the next step of the hierarchical name
    $name := name$.next()
until $name$ is $name_p$.tail()
```

Gathering candidates for ICs and redundancy

To detect integrity constraints and redundancy efficiently, we utilise a two-stage approach:

1. We efficiently detect a set of candidates.
2. The user can confirm/refute them or request a full check.

For this purpose, we introduce a set of lightweight and easy to compute *footprints* and we apply them to compute candidates for ICs (i.e., primary keys, intra- and inter-model references, and interval-based value constraints) and redundancy in data. A naive approach would compare active domains of all pairs of properties. Instead, when walking through all the data during the schema inference process, the same access can be exploited to mine statistical (and other) information about the active domains, i.e., the *footprints*. They can be then used to compare active domains and determine the desired integrity constraints more efficiently.

Property domain footprint (PDF)

For each property *p*, we define an active domain descriptor utilizing basic statistics and the Bloom filter [54], so-called *Property Domain Footprint* (*PDF*). It is represented as a tuple $PDF = (count, first, unique, required, repeated, sequential, min, minHash, max, maxHash, totalHash, averageHash, bloomFilter)$.

- *Count* is the number of items of the active domain.
- *First* is the number of parent properties in which *p* occurs.
- *unique* $\in \{\mathtt{T}, \mathtt{F}\}$ represents the uniqueness of values within a particular active domain. It is computed by counting the occurrence of each item of the active domain.
- *required* $\in \{\mathtt{T}, \mathtt{F}\}$ represents the nullability of the value of a particular property. It is computed by comparing *p.first* of property *p* with *pp.count* of its parent property *pp*, i.e., $required := (p.first = pp.count)$.
- *repeated* $\in \{\mathtt{T}, \mathtt{F}\}$ represents whether the property is a direct element of an array (`T`) or a single value (`F`). It is computed using auxiliary features *count* and *first*, i.e., $repeated := (count \div first > 1)$.
- *sequential* $\in \{\mathtt{T}, \mathtt{F}, \mathtt{U}\}$ represents the possibility of an active domain of a simple property to represent a sequence of integers. The default value is `U` (if the property is not of data type `Integer`). The sequential feature is computed using auxiliary features *min*, *max*, and *count*, i.e., $sequential := (max - min = count - 1)$.
- *min* is the minimum value of the active domain.
- *minHash* is the minimum hashed value of the active domain. It allows to compare values of distinct data types efficiently and prevents the comparison of possibly extensive data, e.g., BLOBs.
- *max* is the maximum value of the active domain.
- *maxHash* is the maximum hashed value of the active domain.
- *totalHash* is the sum of hashed values of the active domain.
- $averageHash := (totalHash \div count)$ represents the average of hash of unique values within the active domain.
- *bloomFilter* is an array of "small" size σ describing a much larger active domain K of property p at the cost of false positives (i.e., equal hashes of two different values). Using multiple hash functions $H_1(), \ldots, H_n()$ returning values from $\{1, \ldots, \sigma\}$, each distinct value $k \in K$ is hashed and each value of $bloomFilter[H_i(k)]$ is incremented.

PDF miner algorithm

The purpose of the footprint miner (see Algorithm 6) is to create a PDF for each property. First, the data are loaded from the database store in the form of records using a particular database wrapper. For each property *p* of each individual record a footprint *f* is created describing a single value of active domain of a certain property. The hierarchical name $name_p$ is attached to each footprint instance. Next, the instances are merged to create distinct unique sets of each active domain using function *mergeValueDuplicates()*. Then, the distinct values are first grouped by function *groupByKey()*, resulting in set *GP* of pairs $(name_{p_i}, F_i)$, where $F_i = \{f_{i_0}, \ldots, f_{i_n}\}$ is the set of footprints. Second, they are grouped to determine the footprint f_p describing the whole active domain. To do so, merge function *aggregateByHierarchicalName()* (see Algorithm 7) is applied.

Algorithm 6: PDF Miner Algorithm

```
   Input: $W_D$ – wrapper for database system (or model or data source in general) $D$
1         $N_D$ – set of names of kinds whose schema is to be inferred
   Output: $S$ – a set of hierarchically ordered footprints
2  SchemaForest $S = \emptyset$
3  foreach $name_\kappa$ in $N_D$ do
       // preparation phase:
4      $NP := W_D$.flatMapRecordToNameFootprintPairs($name_\kappa$)
       // aggregate footprints to create one footprint for each property:
5      $DP := NP$.mergeValueDuplicates()
6      $GP := DP$.groupByKey()
7      $AP := GP$.aggregateByHierarchicalName()
       // footprint finalisation and collection phase:
8      foreach $(name_p, pdf_p)$ in $AP$ do
9          addToForestAndFinalize($pdf_p$, $S$)
10 return $S$
```

Algorithm 7: Function *aggregateByHierarchicalName()*

```
   Input: $F$ – Non-empty list of footprints to be aggregated
1  $r := (0, 0, T, T, F, F, \infty, \infty, 0, 0, 0, 0, [0, \ldots, 0])$
2  foreach footprint $f$ in $F$ do
3      $r.count$ := sum($r.count$, $f.count$)
4      $r.first$ := sum($r.first$, $f.first$)
5      $r.unique$ := $r.unique$ AND $f.unique$
6      $r.required$ := $r.required$ AND $f.required$
       // $repeated$ and $sequential$ are resolved in later stages
7      $r.min$ := min($r.min$, $f.min$)
8      $r.minHash$ := min($r.minHash$, $f.minHash$)
9      $r.max$ := max($r.max$, $f.max$)
10     $r.maxHash$ := max($r.maxHash$, $f.maxHash$)
11     $r.totalHash$ := sum($r.totalHash$, $f.totalHash$)
       // $averageHash$ is resolved in later stages
12     $r.bloomFilter$ := merge($r.bloomFilter$, $f.bloomFilter$)
13 return $r$
```

Finally, the aggregated property features are appended to the tree structure representing the data and the missing features (i.e., *repeated*, *sequential*, and *averageHash*) are resolved.

Candidate builder algorithm

Having computed footprint f_p for each property p, we can determine candidates for identifiers, references, and data redundancy. We propose Algorithm 8 that consists of three phases:

1. *Identifier candidate* The candidates for identifiers C_{ident} are inferred from the footprints in set F. An identifier must be unique within the active domain of the respective property p and required. Also, the property can not be a direct element of an array. Therefore, the algorithm tests whether $f_p.unique = f_p.required = \texttt{T}$ and $f_p.repeated = \texttt{F}$.

2. *Reference candidate* Candidates for references C_{ref} are inferred on the basis of several observations. A reference is a property that refers to the identifier of another kind. Therefore, we search the Cartesian square $F \times C_{ident}$ excluding pairs (c, c), $c \in C_{ident}$ in order to find pairs (f, c), s.t. f is the footprint of the referencing property p_f and c is the footprint of the referenced property p_c. Additionally, the active domain of referencing property p_f must form a subset of active domain of the referenced property p_c. Therefore, we compare active domains of both the properties using function *formsSubset()* (see Algorithm 9), i.e., we analyse footprints f and c using the following rules:

   - The referencing property does not have to be strictly unique, as the one-to-many (as well as many-to-one or many-to-many) relationship may occur.
   - The referencing property does not have to be required as the lower bound of relationship may be zero-to-one/many.
   - It must hold that $f.minHash \geq c.minHash$, i.e., the referencing property p_f does not contain a smaller value than the referenced property p_c.
   - Similarly, it must hold that $f.maxHash \leq c.maxHash$.
   - Finally, only if all the above conditions are satisfied, Bloom filters are compared. To denote that property f is a reference to property c for each pair of elements $f.bloomFilter[i]$, $c.bloomFilter[i]$ it must hold that $f.bloomFilter[i] \leq c.bloomFilter[i]$. In other words, active domain of p_f must be a subset of active domain of p_c.

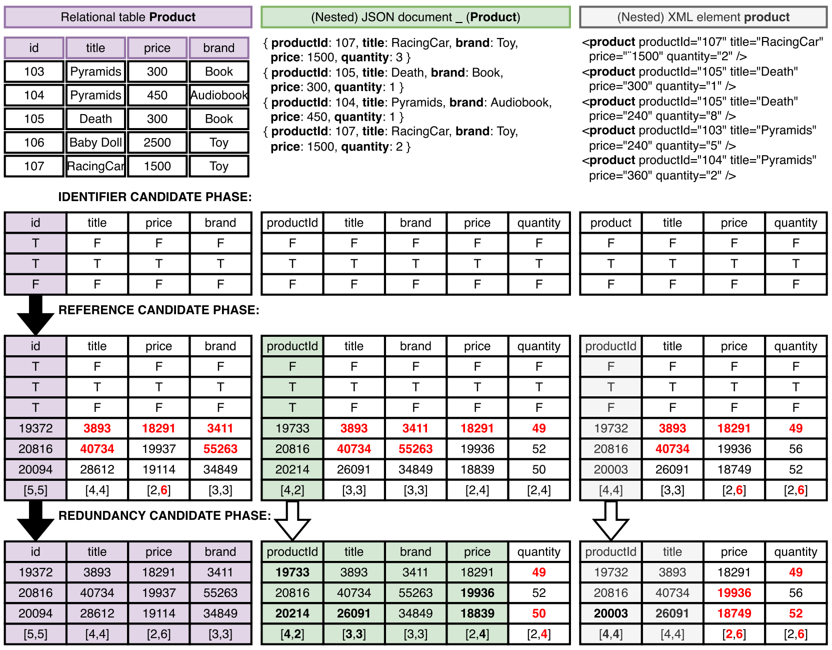
   Additionally, we distinguish between *strong* and *weak* reference candidates. Having *sequential*, *unique*, and *required* set to `T` for both the referencing and referenced properties may imply that there is no relationship between the properties (i.e., both properties form a technical (auto-incremented) identifier of their kind). Therefore, such a combination may lead to a weak candidate for a reference[15].

3 *Redundancy candidate* Finally, reference candidates may be extended into data redundancy candidates C_{red}. Naturally, each pair of referencing and referenced properties having footprints f and c store redundant information. However, we assume that redundantly stored data should cover a more significant part. Hence, we check their descendants and siblings to find more pairs of properties whose active domains form a mutual subset. If there is at least k pairs of neighbouring properties forming such subsets, the reference candidate (f, c) is marked as a *weak reference candidate* and, together with its neighbourhood, extended into a *redundancy candidate*. If for all the pairs of properties in the redundancy candidate the active domains are equal, we speak about *full redundancy*. Otherwise, i.e., when one kind contains only a subset of records of another kind, it is a *partial redundancy*. Also, note that only redundant properties are considered as a part of redundancy, even though both kinds may contain properties having the same name. If multiple properties can form a redundancy pair with the same property, it is up to the user to decide.

[15] Note that such a candidate is not discarded, yet it is not marked as recommended when the inference process applies candidates for RSDs inferred within the local stage.

Algorithm 8: Candidate Builder Algorithm

```
Input: $F$ – list of (pre)computed footprints
       $k$ – minimum number of siblings/descendants to form a redundancy
$C_{ident} := \emptyset$; $C_{ref} := \emptyset$; $C_{red} := \emptyset$;
// Identifier candidate phase:
foreach footprint $f$ in $F$ do
    if $f.unique$ AND $f.required$ AND not $f.repeated$ then
        $C_{ident}$.add($f$)
// Reference candidate phase:
foreach identCandidate $c$ in $C_{ident}$ do
    foreach footprint $f$ in $F \setminus c$ do
        $r$ := formsSubset($f$, $c$)
        if $r$ is not "EMPTY" then
            if isAutoincrement($f$, $c$) then
                $C_{ref}$.add(($f$,$c$,"WEAK",$r$))
            else
                $C_{ref}$.add(($f$,$c$,"STRONG",$r$))
// Redundancy candidate phase:
foreach refCandidate $(f, c, t, r)$ in $C_{ref}$ do
    $D_f$ := descendantOrSibling($f$);
    $D_c$ := descendantOrSibling($c$);
    $R := \emptyset$
    $red$ := "FULL"
    foreach $d_1$ in $D_f$ do
        foreach $d_2$ in $D_c$ do
            $type$ := formsSubset($d_1$, $d_2$)
            if $type$ is not "EMPTY" then
                $R$.add($d_1$,$d_2$)
                $red$ := min($red$, $type$)
    if $R.size() \geq k$ then
        $R$.add(($f$,$c$))
        $C_{red}$.add(($R$,min($red$, $r$)))
        $t$ := "WEAK"
return $(C_{ident}, C_{ref}, C_{red})$
```



Relational table **Product**

id	title	price	brand
103	Pyramids	300	Book
104	Pyramids	450	Audiobook
105	Death	300	Book
106	Baby Doll	2500	Toy
107	RacingCar	1500	Toy

(Nested) JSON document _ (**Product**)

```
{ productId: 107, title: RacingCar, brand: Toy,
  price: 1500, quantity: 3 }
{ productId: 105, title: Death, brand: Book,
  price: 300, quantity: 1 }
{ productId: 104, title: Pyramids, brand: Audiobook,
  price: 450, quantity: 1 }
{ productId: 107, title: RacingCar, brand: Toy,
  price: 1500, quantity: 2 }
```

(Nested) XML element **product**

```
<product productId="107" title="RacingCar"
  price="¨1500" quantity="2" />
<product productId="105" title="Death"
  price="300" quantity="1" />
<product productId="105" title="Death"
  price="240" quantity="8" />
<product productId="103" title="Pyramids"
  price="240" quantity="5" />
<product productId="104" title="Pyramids"
  price="360" quantity="2" />
```

IDENTIFIER CANDIDATE PHASE:

property	id	title	price	brand	productId	title	brand	price	quantity	product	title	price	quantity
unique	T	F	F	F	F	F	F	F	F	F	F	F	F
required	T	T	T	T	T	T	T	T	T	T	T	T	T
repeated	F	F	F	F	F	F	F	F	F	F	F	F	F

REFERENCE CANDIDATE PHASE:

property	id	title	price	brand	productId	title	brand	price	quantity	productId	title	price	quantity
unique	T	F	F	F	F	F	F	F	F	F	F	F	F
required	T	T	T	T	T	T	T	T	T	T	T	T	T
sequential	T	F	F	F	T	F	F	F	F	F	F	F	F
minHash	19372	**3893**	**18291**	**3411**	19733	**3893**	**3411**	**18291**	**49**	19732	**3893**	**18291**	**49**
maxHash	20816	**40734**	19937	**55263**	20816	**40734**	**55263**	19936	52	20816	**40734**	19936	56
averageHash	20094	28612	19114	34849	20214	26091	34849	18839	50	20003	26091	18749	52
bloomFilter	[5,5]	[4,4]	[2,**6**]	[3,3]	[4,2]	[3,3]	[3,3]	[2,4]	[2,4]	[4,4]	[3,3]	[2,**6**]	[2,**6**]

REDUNDANCY CANDIDATE PHASE:

property	id	title	price	brand	productId	title	brand	price	quantity	productId	title	price	quantity
minHash	19372	3893	18291	3411	**19733**	3893	3411	18291	**49**	19732	3893	18291	**49**
maxHash	20816	40734	19937	55263	20816	40734	55263	**19936**	52	20816	40734	**19936**	56
averageHash	20094	28612	19114	34849	**20214**	**26091**	34849	**18839**	**50**	**20003**	**26091**	**18749**	**52**
bloomFilter	[5,5]	[4,4]	[2,6]	[3,3]	[**4**,**2**]	[**3**,**3**]	[3,3]	[2,**4**]	[2,**4**]	[4,4]	[4,4]	[**2**,**6**]	[2,**6**]

Fig. 12 An example of selected footprints and building of candidates

Algorithm 9: Function *formsSubset()*

```
Input: $f_1$, $f_2$ – compared footprints
// $f_1.minHash = f_2.minHash \to$ "FULL"
// $f_1.minHash > f_2.minHash \to$ "PARTIAL"
// $f_1.minHash < f_2.minHash \to$ "EMPTY"
1  $minType$ := determine($f_1.minHash$, $f_2.minHash$)
// $f_1.maxHash = f_2.maxHash \to$ "FULL"
// $f_1.maxHash < f_2.maxHash \to$ "PARTIAL"
// $f_1.maxHash > f_2.maxHash \to$ "EMPTY"
2  $maxType$ := determine($f_1.maxHash$, $f_2.maxHash$)
// $f_1.averageHash = f_2.averageHash \to$ "FULL"
// $f_1.averageHash <> f_2.averageHash \to$ "EMPTY"
3  $avgType$ := determine($f_1.averageHash$, $f_2.averageHash$)
// $\forall i : f_1.bloomFilter[i] = f_2.bloomFilter[i] \to$ "FULL"
// $\forall i : f_1.bloomFilter[i] <= f_2.bloomFilter[i] \to$ "PARTIAL"
// $\exists i : f_1.bloomFilter[i] > f_2.bloomFilter[i] \to$ "EMPTY"
4  $bfType$ := determine($f_1.bloomFilter$, $f_2.bloomFilter$)
5
6  if min($minType$,$maxType$,$avgType$,$bfType$) is "FULL" then
7      return "FULL"
8  else if min($minType$,$maxType$,$bfType$) is "EMPTY" then
9      return "EMPTY"
10 else
11     return "PARTIAL"
```

Example 4.6 Figure 12 introduces an example of footprints of selected properties of the multi-model data from Fig. 9 and their application for detection of candidates. The upper part of the Fig. contains the data, i.e., a subset of properties from relational table *Product* (violet), nested documents _ from JSON collection *Order* (green) and nested elements *Product* from XML collection *Invoice* (grey). The bottom three parts correspond to the three phases, where we can see the values of respective necessary features of footprints.

In order to determine all the footprint features, the duplicate values are removed, the unique values are hashed (using, e.g., rolling hash $rh(v) := v_{rh}$ for each $v \in \mathbb{V}$) and then *minHash*, *maxHash*, *averageHash*, and *bloomfilter* are computed as was described. *averageHash* is rounded to *floor*(*averageHash*) and to compute the Bloom filter (of size = 2), hash functions $h_1(v_{rh}) := v_{rh} \mod 2$ and $h_2(v_{rh}) := floor(sqrt(v_{rh})) \mod 2$ are used.

Regarding the detection of candidates, first all footprints are iterated and their *unique*, *required*, and *repeated* features are checked. In this particular case, only the footprint of property *id* satisfies that *unique* and *required* are both set to `T` and *repeated* is set to `F`, therefore only a single identifier candidate is created and propagated to the next phase of the algorithm. Also note that *id* is probably a technical identifier based on auto-increment (i.e., *sequential* is also set to `T`).

Next, all relevant distinct pairs of footprints are compared to find candidates for references. If any footprint feature does not satisfy the requirements for a reference, it is denoted by the red colour. As we can see, only properties *productId* (JSON) and *productId* (XML) satisfy the requirements and therefore form the set of reference candidates C_{ref}.

Finally, reference candidates are checked to form redundancy candidates, whereas $k = 2$. In this case, we compare siblings (as there are no further nested properties) of pairs (*id*, *productId*) (JSON) and (*id*, *productId*) (XML). In the former case, there is a redundancy between properties *title* (REL, JSON), *brand* (REL, JSON), and *price* (REL, JSON). In the latter case, there is a redundancy only between properties *title* (REL, XML). In the former case, the number of pairs $3 \geq k$, therefore, the reference candidate is extended into the redundancy candidate, and the former reference candidate is marked as weak. In the latter case, the reference candidate remains unchanged as $1 < k$, and it does not form the candidate for redundancy.

Also note that (*id*, *title*, *price*, *brand*) (REL), (*productId*, *title*, *price*, *brand*) (JSON) form a partial redundancy, since multiple requirements are violated, e.g., features *average* are not equal (see the bold font).

Global phase

The local phase consists of inference of local (single-system, single-kind) schemas described as tree-based RSDs and building a set of candidates for identifiers, references, and redundancy. The global phase applies the knowledge gained in the previous steps and joins RSDs using candidates for references and redundancy into the resulting global multi-model schema. It can also begin with an optional full check of candidates, i.e., removing false positives.

Checking of candidates

Depending on the implementation, either the user may confirm/refute the suggested candidates (or denote user-specified candidates) using an interactive interface (see

"Architecture and implementation" section). Or, (s)he may decide which subset of candidates for references and redundancy[16] will be thoroughly checked. The checking itself is implemented as a distributed MapReduce job:

- Checking of references involves mapping of each value of a referencing property *ref* into tuple (*value*, `REF`) while each value of referenced property *id* is mapped to tuple (*value*, `ID`) as well. Reduction and further mapping by key takes place as follows:

  – If the list assigned to *value* contains both `REF` and `ID`, the result is mapped to 0.
  – If the list assigned to *value* contains only `ID`, the result is mapped to 1.
  – Otherwise, the result is mapped to -1.

  Finally, the minimum value is selected. If the result is -1, the candidate for reference is not valid and removed. If the result is 0, it is a full reference (i.e., the active domains of *ref* and *id* are equal). The result of 1 denotes that property *ref* refers to a subset of the active domain of property *id*.

- Checking of redundancy is similar; in addition we have to check values of all neighbouring redundant properties of the referencing and referenced properties *ref* and *id*. First, for each record its redundant properties are mapped to tuple (*value*, {*red_subrecord*, *source*}), where *value* is the value of referencing/referenced property, *red_subrecord* are values of ordered redundant properties in the record, and *source* $\in$ {`REF`, `ID`}. Next, the tuples are reduced by key and then mapped as follows:

  – If the list assigned to *value* contains two equal sub-records from distinct sources, the result is mapped to 0. If the sub-records are not equal, the result is mapped to -1.
  – If the list assigned to *value* contains only sub-record from the *source* `REF`, the result is mapped to -1.
  – Otherwise, the result is mapped to 1.

  Finally, the minimum value is selected. If the result is -1, the candidate for redundancy is not valid, i.e., either there is a sub-record in the kind containing the referencing property *ref* that is not a part of the kind containing referenced property *id*, or the sub-records with the same *value* do not share the same values in all redundant neighbouring properties. If the result is 0, the redundancy is full. Otherwise, the redundancy is partial.

Joining of RSDs

Joining of RSDs may be implemented variously, depending on the selected output of the inference process. Either the RSDs, together with the confirmed/thoroughly checked candidates for identifiers, references, and redundancy, are transformed to an output

[16] Candidates for identifiers do not have to be checked once approved by the user as long as the identifier requires only features ***unique*** and ***requires*** being set to `T`.

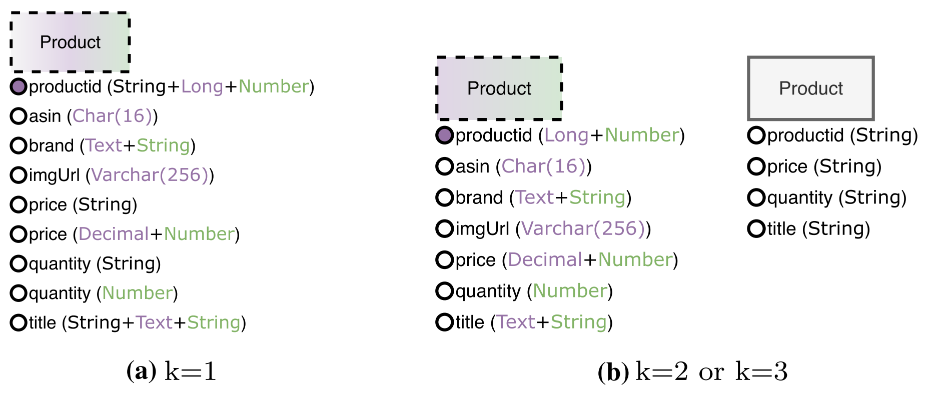


Fig. 13 Example of redundancy, k=1, k=2, k=3

format, such as XML Schema [55], JSON Schema [47] etc. Or, a more abstract representation using ER [51] or UML [9] can be used. How the information is captured depends on the selected format. For example, for representation of redundancy, the globally defined XML elements and respective XML references can be used. In the implementation of the inference approach (see "Architecture and implementation" section), we also use a simple visualisation of the forest of RSDs, where the identifiers, as well as redundant parts of the trees, are respectively graphically denoted, and the trees are interlinked with a new type of edges representing the references.

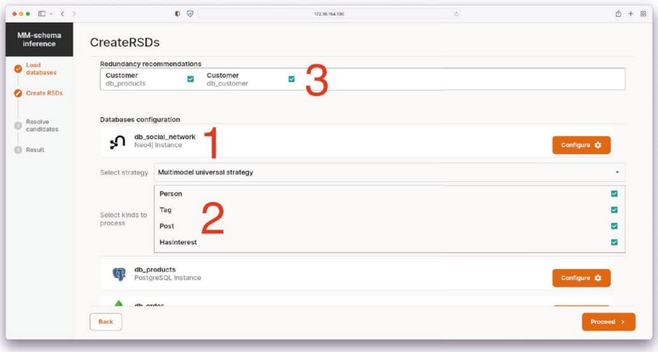
Example 4.7 Depending on the selected parameters of the algorithm, we can get as a result, e.g., the ER model depicted in Fig. 1. If we look closely, e.g., at entity *Product*, in Fig. 13 we can see an example of its alternative result depending on parameter $k = 1$ or $k \in \{2, 3\}$. The colours represent the respective "overlapping" models and properties (relational, JSON document, or XML document). If $k = 1$, we would get one common schema for kind *Product* represented in all the three models. If $k \in \{2, 3\}$, we would get a common schema for the relational and JSON document model and a different schema for the XML document model.

Note that property *productId* is an identifier only in the relational table *Product* (illustrated by the purple colour). Also, note that non-redundant property *price* has probably a different meaning in distinct models. In the case of the XML document model, it could be a purchase price, whereas, in the relational and JSON document model, it could be a selling price.

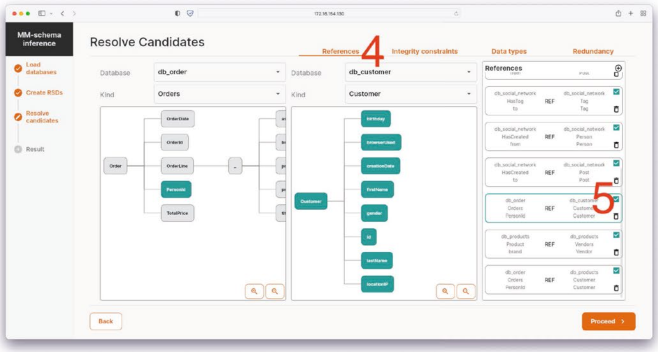
Architecture and implementation

The proposed approach was implemented as a modular framework called *MM-infer*[17]. Its graphical interface and general functionality have been introduced in demo paper [12], but without technical details, algorithms, and experiments provided in this paper. It currently supports the following models and DBMSs: *PostgreSQL* (relational and

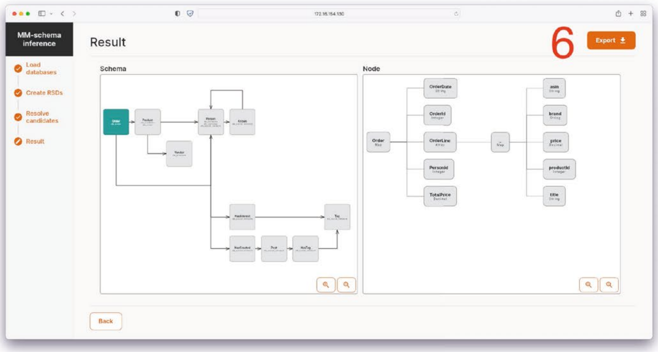
[17] https://www.ksi.mff.cuni.cz/~koupil/mm-infer/index.html.



(a)



(b)



(c)

Fig. 14 Screenshots of *MM-infer*

document, i.e., multi-model), *Neo4j*[18] (graph), and *MongoDB*[19] (document) which represents both schema-full and schema-less DBMS.

The frontend of *MM-infer* was implemented in *Dart* using framework *Flutter*[20]. Sample screenshots are provided in Fig. 14. The expected work with the tool is as follows:

[18] https://neo4j.com/.

[19] https://www.mongodb.com/.

[20] https://flutter.dev/.

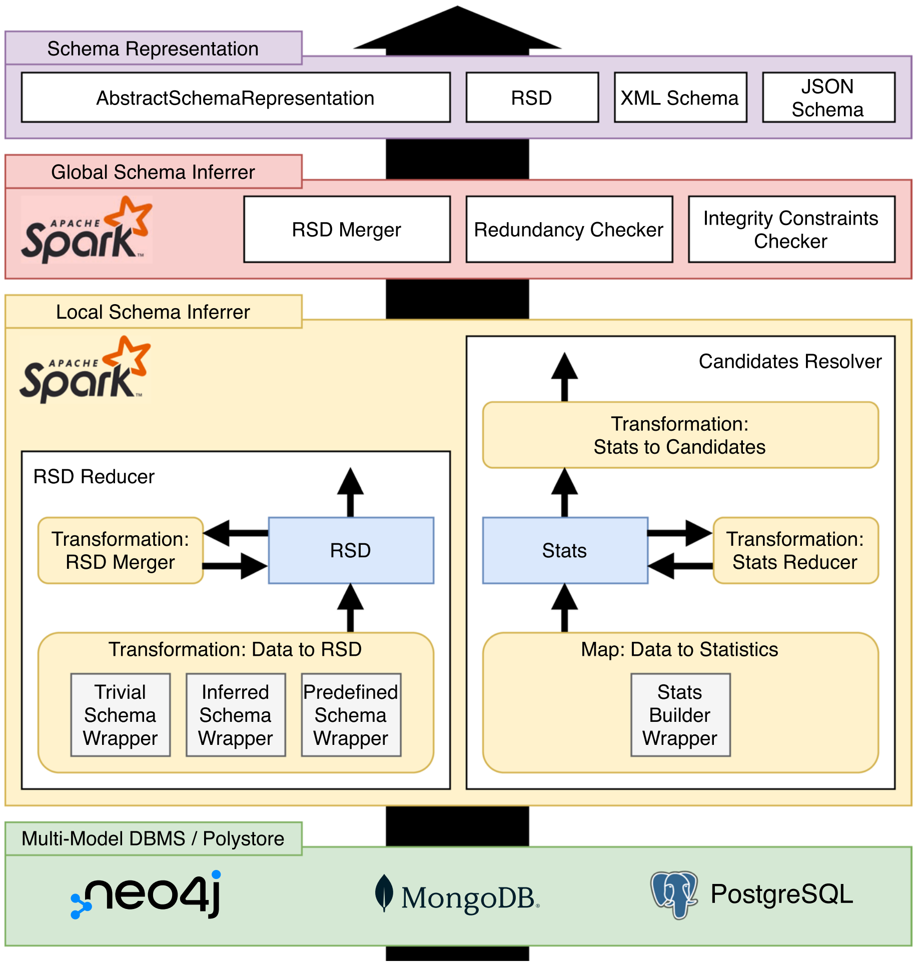


Fig. 15 Architecture of *MM-infer*

The user selects particular DBMSs (1) and kinds (2) to be involved in the inference process. (S)he can also confirm/refute initial redundancy (3) based on kind names. Then the local schema inferrer infers the local RSDs and generates the candidates. In the next screen, the user selects particular types of candidates (4) and confirms/refutes the suggestions (5). After the global schema inferrer performs the full check of candidates (if required) and merges the RSDs into the global schema to be visualised to the user or transformed to a requested form (6).

The backend of *MM-infer* was implemented in *Java* and using *Apache Spark*[21]. The architecture of *MM-infer*, depicted in Fig. 15, reflects the steps of the above described inference process:

- At the bottom we can see data sources (green box) – a multi-model DBMS or a set of single/multi-model DBMS (i.e., a polystore-like storage).
- The local schema inferrer (yellow box) uses three types of wrappers that transform the input data/schemas into RSDs:

[21] https://spark.apache.org/.

Table 2 An example of *PostgreSQL* (relational model) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Ref.
Tuple	_	U	(1,1)	F	Map	REL	{...}	ϵ	ϵ
Attribute (simple)	Name	T/F/U	(0/1,0/1)	T/F/U	Simple	REL	ϵ	ϵ	$\epsilon/\kappa.p$
Attribute (array)	Name	T/F/U	(0/1,0/1)	F	Array	REL	{...}	Trivial	ϵ
Element of an array	_	T/F/U	(1,0/1)	F	Simple	REL	ϵ	ϵ	ϵ
Simple identifier	Name	T	(1,1)	T	Simple	REL	ϵ	ϵ	$\epsilon/\kappa.p$
Reference	Name	T/F/**U**	(0/1,1)	T/F/U	Simple	REL	ϵ	ϵ	$\kappa.p$

- Having schema-full database, a *pre-defined schema* already exists. Therefore, we only fetch the schema and translate it to the unifying representation (i.e., only a unifying wrapper translator must be implemented to translate the local schema into a unified representation).
- Having a schema-free (or mixed) database approach, a robust schema inference approach may already exist for a particular model (e.g., the XML or JSON document model). If so, we infer the schema by exploiting such an approach, and then we translate the resulting *inferred schema* using a unifying translator layer.
- Finally, having schema-free or schema-mixed DBMS and no existing schema inference approach, the *basic schema* is inferred for each kind.

  Local schema inferrer then merges the RSDs locally (i.e., within one DBMS) using *Apache Spark*. In parallel, it gathers and merges the data statistics and produces the respective candidates to be eventually modified by the user.

- The global schema inferrer (red box) checks candidates for references and redundancy and merges the RSDs globally (i.e., in the context of all inputs).
- The resulting multi-model schema is provided to the user in the chosen representation (violet box).

Database wrappers

MM-infer is based on custom-tailored wrappers, each of which reads individual records from a particular DBMS and returns its RSD. Note that not only the whole data set may be represented in different data formats and data models, but a collection of data models may represent even a single record. The wrapper also includes user settings that determine the level of detail of the described schema. For example, the user may request the inference of data structures which implicitly may not be supported by the data model (e.g., `Set` and `Map` in the case of JSON documents) but can be specified in RSDs.

Naturally, we assume the implementation of a wrapper for each DBMS. However, separate wrappers may exist for distinct settings of a particular DBMS—e.g., schema-less vs schema-mixed—or a wrapper that involves a particular single-model schema inference approach. A separate wrapper module is also devoted to reading data statistics and their transformation into PDFs.

The following examples show the core of sample implementation of wrappers for particular considered models of selected popular DBMSs.

Table 3 An example of *SciDB* (array model) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Refs.
Cell	_	U	(1,1)	F	Map	ARRAY	{...}	ϵ	ϵ
Attribute (simple type)	Name	T/F/U	(0/1,0/1)	T/F/U	Simple	ARRAY	ϵ	ϵ	ϵ
Simple identifier (dimension)	Name	T	(1,1)	T	Simple	ARRAY	ϵ	ϵ	ϵ

Table 4 An example of *Neo4j* (graph model) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Refs.
Node/edge	_	U	(1,1)	F	Map	GRAPH	{...}	ϵ	ϵ
Property (simple type)	name	T/F/U	(1,1)	T/F/U	Simple	GRAPH	ϵ	ϵ	$\epsilon/\kappa.p$
Property (array type)	name	T/F/U	(1,1)	F	Array	GRAPH	{...}	Trivial	ϵ
Element of an array	_	T/F/U	(1,0/1)	F	Simple	GRAPH	ϵ	ϵ	ϵ
Identifier (simple)	name	T	(1,1)	T	Simple	GRAPH	ϵ	ϵ	ϵ
Reference	from/to	T/F/U	(1,1)	F	REF	GRAPH	ϵ	ϵ	$\kappa.p$

Example 5.1 Table 2 illustrates the mapping of PostgreSQL record schema (representing relational tuple) or property (i.e., an attribute of a tuple or an identifier of a tuple) to an RSD. Note that PostgreSQL record (tuple) has an anonymous name _ and its type corresponds to `Map` as the properties are named values stored in an arbitrary order. In this case, we distinguish between `Simple` (representing any simple type) and `Array` type attribute, both possibly nullable (i.e., *share* can be either 0 or 1), and array being only homogeneous (i.e., described by a set of children $\{\ldots\}$ and a trivial automaton representing an arbitrary number of anonymously named elements of an array (i.e., _+). Also, note that a structured property is not allowed as the purely relational model is naturally aggregate-ignorant[22]. Finally, we may consider an identifier and a reference as a particular attribute type. The identifier must be unique, and *share* cannot be 0 as the identifier cannot be nullable. In the case of references, the *ref* field is set to the referenced kind κ and respective referenced property p.

Example 5.2 Table 3 illustrates mapping of SciDB[23] array schema to the unified RSD representation. A multi-dimensional array consists of anonymously named cells which contain a `Map` of named simple attributes. Note that SciDB does not allow complex attributes, therefore neither `Array` type nor `Structure` type is allowed in the mapping. Finally, each cell is uniquely identified by dimension coordination (i.e., an identifier), whereas references between tables are not supported.

[22] PostgreSQL also supports the document model (i.e., JSON and XML). In this case, the mapping of respective embedded properties is the same way as in the document model (see Example 5.6 and 5.5).

[23] https://www.paradigm4.com/.

Table 5 An example of *Redis* (key/value model) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Refs.
Pair	_	U	(1,1)	F	Tuple	KV	{...}	ϵ	ϵ
Value (simple)	_	T/F/U	(1,1)	F	Simple	KV	ϵ	ϵ	ϵ
Value (array)	_	T/F/U	(1,1)	F	Array	KV	{...}	Automaton	ϵ
Value (set)	_	T/F/U	(1,1)	F	Set	KV	{...}	ϵ	ϵ
Element of an array/set	_	T/F/U	(1,0/1)	F	AnyType	KV	{...}/ϵ	ϵ	ϵ
Key (simple)	_	T	(1,1)	T	Simple	KV	ϵ	ϵ	ϵ

Table 6 An example of *MarkLogic* (XML document model) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Refs.
Root	Name	U	(1,1)	F	Array/ Map/ Array+ Map	XML	{...}	Automaton	ϵ
@attribute	@name	T/F/U	(1,1)	T/F/U	ϵ/Simple	XML	ϵ	ϵ	$\epsilon/\kappa.p$
Element (simple w/o atts)	Name	T/F/U	(1,0/1)	F	Simple	XML	ϵ	ϵ	ϵ
Element (simple + atts)	Name	T/F/U	(1,0/1)	F	Array+ Map	XML	{...}	(__TEXT__?)	ϵ
Element (array w/o atts)	Name	T/F/U	(1,0/1)	F	Array	XML	{...}	automaton	ϵ
Element (array + atts)	Name	T/F/U	(1,0/1)	F	Array+ Map	XML	{...}	automaton	ϵ
TEXT_NODE	__TEXT__	T/F/U	(1,0/1)	F	ϵ/Simple	XML	ϵ	ϵ	ϵ
Identifier (simple)	Name	T	(1,1)	T	Simple	XML	ϵ	ϵ	ϵ
Reference	Name	T/F/U	(0/1,1)	T/F/U	Simple	XML	ϵ	ϵ	$\kappa.p$

Example 5.3 Table 4 illustrates mapping of Neo4j node/edge schema to RSD. Both a node and an edge have an anonymous name _ and its type corresponds to `Map` (i.e., both contain an unordered map of uniquely named properties). Similarly to PostgreSQL (its relational model), only simple attributes and homogeneous arrays of simple types are allowed. Also, only simple identifiers are allowed, and only two special kinds of references are allowed, i.e., special properties *from* and *to* representing the source and the target of an edge. Also, note that *share* = 0 is not allowed since *null* meta values are represented as a missing property in Neo4j. Therefore no property can be mapped to an RSD having *share* = 0.

Example 5.4 Table 5 illustrates mapping of Redis[24] (key/value model) key/value pair schema to RSD. A pair is an anonymously named `Tuple` of ordered anonymous properties, i.e., a key and a value. A key is a simple identifier while the value can be `Simple`, `Array`, `Set` or structured in general. However, we do not support mapping of structural

[24] https://redis.io/.

Table 7 An example of *MongoDB* (JSON document) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Refs.
Document	_	F	(1,1)	F	Map	DOC	{...}	ϵ	ϵ
Field (simple)	Name	T/F/U	(0/1,1)	T/F/U	ϵ/Simple	DOC	ϵ	ϵ	$\epsilon/\kappa.p$
Field (hom. array)	Name	T/F/U	(0/1,0/1)	T/F/U	ϵ/Array	DOC	{...}	Trivial	ϵ
Field (het. array)	Name	T/F/U	(0/1,0/1)	T/F/U	ϵ/Array	DOC	{...}	Automaton	ϵ
Field (structure)	Name	T/F/U	(0/1,0/1)	T/F/U	ϵ/Map	DOC	{...}	ϵ	$\epsilon/\kappa.p$
Element of an array	_	T/F/U	(1,0/1)	T/F/U	AnyType	DOC	ϵ/{...}	ϵ/ trivial/ automaton	ϵ
Identifier (simple)	Name	T	(1,1)	T	Simple/ Map	DOC	ϵ/{...}	ϵ	$\epsilon/\kappa.p$
Reference	Name	T/F/U	(0/1,0/1)	T/F/U	Simple/ Map	DOC	ϵ/{...}	ϵ	$\kappa.p$

Table 8 An example of *Cassandra* (columnar model) schema mapping

Construct	Name	Unique	Share	Id	Types	Models	Children	regexp	Refs.
Row	_	F	(1,1)	F	Map	COL	{...}	ϵ	ϵ
Column (simple)	Name	T/F/U	(0/1,0/1)	T/F/U	Simple	COL	ϵ	ϵ	ϵ
Column (homogeneous array)	Name	T/F/U	(1,1)	F/U	Array	COL	{...}	Trivial	ϵ
Tuple	Name	T/F/U	(1,1)	F/U	Tuple	COL	{...}	Automaton	ϵ
Column family (complex structure)	Name	T/F/U	(1,1)	F/U	Map	COL	{...}	Trivial	ϵ
List (= array; column)	Name	T/F/U	(1,1)	F/U	Array	COL	{...}	Automaton	ϵ
Column (map)	Name	T/F/U	(1,1)	F/U	Map	COL	{...}	ϵ	ϵ
Column (set)	Name	T/F/U	(1,1)	F/U	Set	COL	{...}	ϵ	ϵ
Element of an array	_	T/F/U	(1,0/1)	F/U	AnyType	COL	ϵ	ϵ/trivial/ automaton	ϵ
Element of a tuple	_	T/F/U	(1,1)	F/U	AnyType	COL	ϵ/{...}	ϵ/trivial/ automaton	ϵ
Element of a set	_	T/F/U	(1,1)	F/U	AnyType	COL	ϵ/{...}	ϵ/trivial/ automaton	ϵ
Identifier (simple)	Name	T	(1,1)	T	Simple	COL	ϵ	ϵ	ϵ

values – otherwise it would be document model. Redis also does not support references, therefore no mapping exists for them[25].

Example 5.5 Table 6 illustrates the mapping of an XML schema to a unifying RSD. The root of an XML document is a named root element possibly having attributes (reflected as type `Map`) and nested subelements (reflected as type `Array`), or both (`Array+Map`).

[25] Yet it is not a rule in general, e.g., key/value store *RiakKV* (https://riak.com/products/riak-kv/index.html) supports links between so-called *buckets* allowing link walking

Any element can be `Simple` (i.e., without nested elements) or `Array`-type (i.e., having at least 1 nested element), with or without attributes. In addition, XML allows text content of an element (being arbitrary nested between other subelements within an array). As for the attributes, only simple types are allowed, and the name of an attribute is prefixed by `@`. Finally, a simple/composite attribute may identify an XML element. A reference may refer to this identifier. Both are mapped as an identifier or a reference as a special kind of attribute. Note that in an XML document, an element of an array is an ordinary named (sub)element. Therefore Table 6 does not contain a special row for an element of an array.

Example 5.6 Table 7 illustrates the mapping of a JSON schema into a unifying RSD. The root of a JSON document is an anonymously named map (reflected as type `Map`) of name/value pairs (fields). A field can be simple (i.e., allowing only simple types of values), array (i.e., a homogeneous array allowing elements of the same type or a heterogeneous array allowing elements of any type), or structural (i.e., a nested document of type `Map`). An element of an array may be of any type, e.g., simple, array, or structural. Finally, each document is identified by an identifier (simple or composite), and references to other documents are supported.

Example 5.7 Finally, Table 8 illustrates mapping of *Cassandra*[26] column family schema to a unifying RSD. A row of column family is an unordered set of uniquely named name/value pairs, i.e., reflected as type `Map`. In addition, *Cassandra* explicitly allows many variations of columns—e.g., a simple column, complex columns (e.g., array, set, map), or simple and complex identifiers. On the other hand, *Cassandra* does not allow references. Therefore no mapping for references is proposed (allowed).

Experiments

MM-infer was implemented not only as a user-friendly tool for interaction with the user during the inference process but also as a tool that enables verification of the correctness and efficiency of the proposed algorithms. In particular, we evaluate RBA against PBA schema inference in terms of execution performance concerning the number of input documents and their structure. The experiments were run over subsets of 6 real-world datasets:

[26] https://cassandra.apache.org/.

Table 9 Statistics of the used data sets

Collection	Size (MB)	Properties	NestedDocs	Arrays	maxDepth	avgDepth
imdb16k	1.19	11.00	1.00	0.00	2	2.00
imdb32k	2.37	11.00	1.00	0.00	2	2.00
imdb64k	4.80	11.00	1.00	0.00	2	2.00
imdb128k	9.63	11.00	1.00	0.00	2	2.00
imdb256k	19.19	11.00	1.00	0.00	2	2.00
imdb512k	39.09	11.00	1.00	0.00	2	2.00
imdb1024k	81.32	11.00	1.00	0.00	2	2.00
imdb2048k	164.17	11.00	1.00	0.00	2	2.00
imdb4096k	331.08	11.00	1.00	0.00	2	2.00
imdb8192k	668.97	11.00	1.00	0.00	2	2.00
lexemes1k	2.09	128.53	39.26	14.73	26	10.92
lexemes2k	3.80	118.33	36.05	13.60	26	10.62
lexemes4k	7.43	115.32	35.06	13.16	26	10.60
lexemes8k	23.99	193.84	56.38	26.29	26	10.10
lexemes16k	109.96	399.25	119.88	50.52	26	13.42
lexemes32k	244.33	439.74	131.86	53.07	26	13.58
lexemes64k	429.01	408.93	120.60	50.95	26	11.93
lexemes128k	863.78	405.33	119.90	48.78	26	12.03
lexemes256k	1560.63	367.57	108.94	45.79	26	11.71
lexemes512k	3202.35	375.31	111.44	46.64	26	11.86
wikidata1k	54.57	3 041.47	825.10	302.71	22	11.96
wikidata2k	99.13	2753.54	746.45	274.60	22	12.32
wikidata4k	158.38	2218.33	603.29	221.99	22	12.36
wikidata8k	280.31	1965.14	533.50	198.65	22	12.55
wikidata16k	492.38	1738.91	470.97	174.48	22	12.89
wikidata32k	914.44	1597.41	433.88	158.36	22	12.86
wikidata64k	1611.56	1398.28	381.05	136.18	22	12.89
wikidata128k	2702.60	1175.67	321.31	114.15	22	12.91
wikidata256k	4276.11	925.00	255.06	86.86	22	12.73
wikidata512k	6755.81	725.59	201.30	66.62	22	12.60
yelpreview1k	0.72	11.00	1.00	0.00	2	2.00
yelpreview2k	1.41	11.00	1.00	0.00	2	2.00
yelpreview4k	2.77	11.00	1.00	0.00	2	2.00
yelpreview8k	5.57	11.00	1.00	0.00	2	2.00
yelpreview16k	11.12	11.00	1.00	0.00	2	2.00
yelpreview32k	22.27	11.00	1.00	0.00	2	2.00
yelpreview64k	44.58	11.00	1.00	0.00	2	2.00
yelpreview128k	89.48	11.00	1.00	0.00	2	2.00
yelpreview256k	179.15	11.00	1.00	0.00	2	2.00
yelpreview512k	361.99	11.00	1.00	0.00	2	2.00
yelpreview1024k	729.29	11.00	1.00	0.00	2	2.00
yelpreview2048k	1461.10	11.00	1.00	0.00	2	2.00
yelpreview4096k	2932.20	11.00	1.00	0.00	2	2.00
yelptip1k	0.18	7.00	1.00	0.00	2	2.00
yelptip2k	0.37	7.00	1.00	0.00	2	2.00
yelptip4k	0.74	7.00	1.00	0.00	2	2.00
yelptip8k	1.48	7.00	1.00	0.00	2	2.00
yelptip16k	2.95	7.00	1.00	0.00	2	2.00
yelptip32k	5.90	7.00	1.00	0.00	2	2.00

Table 9 (continued)

Collection	Size (MB)	Properties	NestedDocs	Arrays	maxDepth	avgDepth
yelptip64k	11.82	7.00	1.00	0.00	2	2.00
yelptip128k	23.68	7.00	1.00	0.00	2	2.00
yelptip256k	47.60	7.00	1.00	0.00	2	2.00
yelptip512k	95.73	7.00	1.00	0.00	2	2.00
yelpuser1k	11.59	24.00	1.00	0.00	2	2.00
yelpuser2k	20.07	24.00	1.00	0.00	2	2.00
yelpuser4k	35.83	24.00	1.00	0.00	2	2.00
yelpuser8k	59.72	24.00	1.00	0.00	2	2.00
yelpuser16k	92.84	24.00	1.00	0.00	2	2.00
yelpuser32k	130.97	24.00	1.00	0.00	2	2.00
yelpuser64k	240.10	24.00	1.00	0.00	2	2.00
yelpuser128k	359.46	24.00	1.00	0.00	2	2.00
yelpuser256k	648.87	24.00	1.00	0.00	2	2.00
yelpuser512k	1055.76	24.00	1.00	0.00	2	2.00
yelpuser1024k	1814.98	24.00	1.00	0.00	2	2.00

- An 8.192 million record sample of IMDB *title.basics* tab-separated-values (TSV) collection [27] imported into Neo4j graph database.
- A 512 thousand record sample of Wikidata Lexeme namespace JSON collection [28] imported into mongoDB document database.
- A subset of 512 thousand records of Wikidata entities in a single JSON collection dump [29] imported into mongoDB.
- A three collections of Yelp Academic Dataset [30] of JSON documents imported into mongoDB, namely 4.096 million record sample of Review collection, 512 thousand records of Tip collection, and 1.024 million records of User collection.

The characteristics of the selected datasets are listed in Table 9 to indicate the growing complexity. Experiments with different data sizes were executed to measure the performance depending on the number and complexity of the input documents.

The experiments were performed on a bare-metal virtual machine running on the VMware[31] infrastructure. The allocated hardware resources were CPU Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz (8 core), 64 GB of memory, and a solid-state drive with a capacity of 1.1 TB. Apache Spark was executed locally with 32 GB RAM set to JVM via -Xmx32768M. A possible bias caused by temporary decreases in system resources was mitigated by 20 runs of each algorithm on each extracted subset of input data. The extremes (minimum and maximum) were removed from the measurements, and the remaining measurements were averaged.

[27] https://www.imdb.com/interfaces/.

[28] https://www.wikidata.org/wiki/Wikidata:Database_download.

[29] https://www.wikidata.org/wiki/Wikidata:Database_download.

[30] https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset.

[31] https://www.vmware.com/.

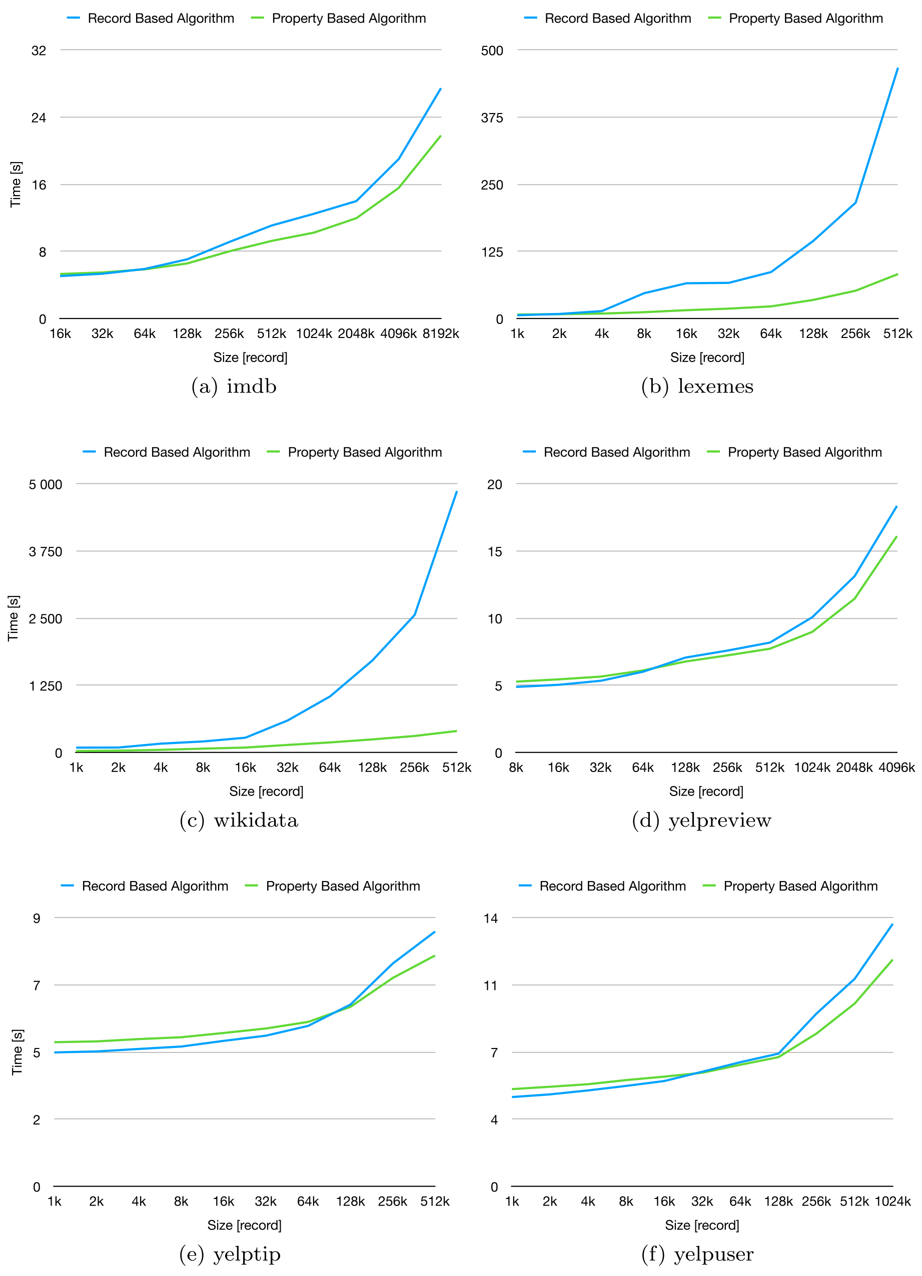


Fig. 16 Results of experiments

Figure 16 confirms our hypothesis that the RBA, which utilised the traditional strategy to work with whole records, is slower when run on more complex data (i.e., lexeme or wikidata dump) because it has to work with whole records instead of separate properties. Instead, PBA first merges separate properties and, in the end, it merges them into the resulting schema of the whole kind. In the case of simple data, this difference is not that significant, but larger, more complex documents depict the difference. In addition, PBA is also scalable more easily because it works with the approximately same data portions regardless of the size of the input. Hence, RBA is more suitable for smaller, less complex data. In general, RBA is more appropriate for aggregate-ignorant models with a smaller amount of properties in kinds, whereas PBA is a better choice for aggregate-oriented models.

Conclusion and future work

This paper introduces a novel proposal of an approach dealing with schema inference for multi-model data. Contrary to existing works, it covers all currently popular data models and possible combinations, including cross-model references and data redundancy. In addition, it can cope with large amounts of data—a standard feature in NoSQL databases but not commonly considered in existing schema inference strategies. Note that the idea of multi-model schema inference, namely the PBA, was introduced in the paper [56] together with initial experiments. In this paper, we have provided a complete description of the proposal, i.e., both the RBA and PBA strategies and their experimental comparison; a discussion of the broad context of the schema inference problem and the variety of the combined models; all the building blocks of the proposed approach, including, e.g., the type hierarchy; illustration of the schema inference process using the tool *MM-infer* as well as its architecture and examples of system-specifics wrappers.

The core idea of the proposal is completed, implemented as a tool *MM-infer*, and experimentally verified. Nevertheless, there are still possible directions for extension and exploitation. In our future work, we will focus on the inference of more complex cross-model integrity constraints which can be expressed, e.g., using the Object Constraint Language [57]. To infer a more precise target schema, we can also incorporate the eventual knowledge of multi-model queries or semantics of the data. In the former case, we can infer an equivalent schema that reflects the expected data access. In the latter case, we can reveal information that cannot be found in the data itself. Last but not least, the inference approach, together with statistical analysis of the source data, can reveal and enable the backwards correction of errors (i.e., occasionally occurring exceptional cases) in the data.

Abbreviations

XML	eXtensible Markup Language
JSON	Javascript Object Notation
DBMS	Database management system
UML	Unified Modeling Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
ER	Entity-Relationship
MDE	Model Driven Engineering
IC	Integrity constraint
DTD	Document Type Definition
MM	Multi-model
SQL	Structured Query Language
DDL	Data Definition Language
OCL	Object Constraint Language
OWL	Web Ontology Language
URI	Uniform Resource Identifier
RSD	Record Schema Description

Author contributions
The share of author's contributions in this paper is equal. All the authors read and approved the final manuscript.

Funding
The work was supported by the GAČR project no. 20-22276S and by the GAUK project no. 16222.

Availability of data and materials
Framework ***MM-infer*** was described in demo paper [12].

Declaration

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 28 May 2022 Accepted: 27 June 2022
Published online: 11 August 2022

References
1. Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C 2008. http://www.w3.org/TR/REC-xml. Accessed 28 May 2022.
2. International E. JavaScript Object Notation (JSON). Ecma International. 2017. http://www.JSON.org/.
3. Gold EM. Language identification in the limit. Inform Control. 1967;10(5):447–74.
4. Vosta O, Mlynkova I, Pokorný J. Even an ant can create an XSD. In: DASFAA 2008. LNCS, vol. 4947, pp. 35–50. Springer, 2008
5. Klempa M, Kozak M, Mikula M, Smetana R, Starka J, Švirec M, Vitásek M, Nečaský M, Mlýnková I. jInfer: a framework for XML schema inference. Comput J. 2013;58(1):134–56.
6. Bex GJ, Gelade W, Neven F, Vansummeren S. Learning deterministic regular expressions for the inference of schemas from XML data. ACM Trans Web. 2010;4(4):1.
7. Baazizi MA, Colazzo D, Ghelli G, Sartiani C. Parametric schema inference for massive JSON datasets. VLDB J. 2019;28(4):497–521.
8. Ruiz DS, Morales SF, Molina JG. Inferring versioned schemas from NoSQL databases and its applications. In: Ruiz DS, editor. ER 2015 LNCS, vol. 9381. Berlin: Springer; 2015. p. 467–80.
9. Group OM. OMG Unified Modeling Language (OMG UML), Version 2.5. 2015. http://www.omg.org/spec/UML/2.5/. Accessed 28 May 2022.
10. Koupil P, Svoboda M, Holubova I. MM-cat: A tool for modeling and transformation of multi-model data using category theory. In: MODELS '21, pp. 635–639. IEEE. 2021. https://doi.org/10.1109/MODELS-C53483.2021.00098. Accessed 28 May 2022.
11. Svoboda M, Contos P, Holubova I. Categorical modeling of multi-model data: One model to rule them all. In: Svoboda M, editor. MEDI 2021, LNCS, vol. 12732. Berlin: Springer; 2021. p. 1–8.
12. Koupil P, Hricko S, Holubová I. MM-infer: a tool for inference of multi-model schemas. In: Stoyanovich J, Teubner J, Guagliardo P, Nikolic M, Pieris A, Mühlig J, Özcan F, Schelter S, Jagadish HV, Zhang M. (eds.) Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29–April 1, 2022, p. 2:566–2:569. OpenProceedings.org, 2022. https://doi.org/10.48786/edbt.2022.52.
13. Beckett D. RDF 1.1 XML Syntax. W3C, 2014. http://www.w3.org/TR/rdf-syntax-grammar/. Accessed 28 May 2022.
14. Kellou-Menouer K, Kardoulakis N, Troullinou G, Kedad Z, Plexousakis D, Kondylakis H. A survey on semantic schema discovery. VLDB J. 2021. https://doi.org/10.1007/s00778-021-00717-x.
15. Mlýnková I, Nečaský M. Heuristic methods for inference of XML schemas: Lessons learned and open issues. Informatica. 2013;24(4):577–602.
16. Shafer KE. Creating DTDs via the GB-Engine and Fred. In: Proceedings of SGML'95. Graphic communications association. 1995. p. 399. http://xml.coverpages.org/shaferGB.html.
17. Moh CH, Lim EP, Ng WK (2000) Re-engineering structures from web documents. In: Proceedings of DL '00. DL '00, pp. 67–76. San Antonio: ACM Press.
18. Wong RK, Sankey J. On structural inference for XML data. Report UNSW-CSE-TR-0313, school of computer science. Sydney: The University of New South Wales; 2003.
19. Garofalakis M, Gionis A, Rastogi R, Seshadri S, Shim K. Xtract: a system for extracting document type descriptors from XML documents. SIGMOD Rec. 2000;29(2):165–76.
20. Vošta O, Mlýnková I, Pokorný J. Even an ant can create an XSD. In: Proceedings of DASFAA'08. Lecture notes in computer science, vol. 4947. 2008. New Delhi: Springer; 2008. p. 35–50. https://doi.org/10.1007/978-3-540-78568-2_6.
21. Chidlovskii B. Schema extraction from XML collections. In: Proceedings of JCDL '02. JCDL '02. Portland: ACM Press; 2002. p. 291–292.

22. Ahonen H. Generating grammars for structured documents using grammatical inference methods. Report A-1996-4, department of computer science. Helsinki: University of Helsinki; 1996.
23. Fernau H. Learning XML grammars. In: Perner, P. (ed.) Proceedings of MLDM '01. lecture notes in computer science, vol. 2123. London: Springer; 2001. p. 73–87. https://doi.org/10.1007/3-540-44596-X_7.
24. Min J-K, Ahn J-Y, Chung C-W. Efficient extraction of schemas for XML documents. Inf Process Lett. 2003;85(1):7–12.
25. Bex GJ, Neven F, Schwentick T, Tuyls K. Inference of concise DTDs from XML data. In: Proceedings of VLDB '06. VLDB '06. Seoul: VLDB Endowment. 2006. p. 115–126.
26. Bex GJ, Gelade W, Neven F, Vansummeren S. Learning deterministic regular expressions for the inference of schemas from XML data. ACM Trans Web. 2010;4(4):14–11432. https://doi.org/10.1145/1841909.1841911.
27. Bex GJ, Neven F, Schwentick T, Vansummeren S. Inference of concise regular expressions and DTDs. ACM Trans Database Syst. 2010;35(2):11–11147.
28. Bex GJ, Neven F, Vansummeren S. Inferring XML schema definitions from XML data. In: Proceedings of VLDB '07. VLDB '07. Vienna: VLDB Endowment; 2007. p. 998–1009.
29. Berstel J, Boasson L. XML grammars. In: Berstel J, editor. Mathematical foundations of computer science. LNCS. Berlin: Springer; 2000. p. 182–91.
30. Contos P, Svoboda M. JSON schema inference approaches. In: Grossmann G, Ram S (eds.) Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings. Lecture notes in computer science, vol. 12584. Berlin: Springer; 2020. p. 173–183. https://doi.org/10.1007/978-3-030-65847-2_16.
31. Morales SF. Inferring NoSQL data schemas with model-driven engineering techniques. PhD thesis, University of Murcia. Murcia, Spain. March 2017.
32. Sevilla Ruiz D, Morales SF, García Molina J. Inferring versioned schemas from NoSQL databases and its applications. In: Sevilla Ruiz D, editor. Conceptual modeling. Berlin: Springer; 2015. p. 467–80.
33. Chillón AH, Morales SF, Sevilla D, Molina JG. Exploring the visualization of schemas for aggregate-oriented NoSQL databases. In: ER Forum/Demos 1979. CEUR workshop proceedings, vol. 1979, CEUR-WS.org, 2017. p. 72–85.
34. Candel CJF, Ruiz DS, García-Molina J. A unified metamodel for NoSQL and relational databases. CoRR. 2021. arXiv:2105.06494
35. Klettke M, Störl U, Scherzinger S. Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: Seidl, T., Ritter, N., Schöning, H., Sattler, K., Härder, T., Friedrich, S., Wingerath, W. (eds.) Datenbanksysteme Für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings. LNI, vol. 241. 2015. p. 425–444. http://subs.emis.de/LNI/Proceedings/Proceedings241/article35.html.
36. Klettke M, Awolin H, Storl U, Muller D, Scherzinger S. Uncovering the evolution history of data lakes. In: 2017 IEEE International conference on big data. New York: IEEE; 2017. p. 2380–2389.
37. Möller ML, Berton N, Klettke M, Scherzinger S, Störl U. jhound: Large-scale profiling of open JSON data. BTW 2019. 2019.
38. Fruth M, Dauberschmidt K, Scherzinger S. Josch: Managing schemas for NoSQL document stores. In: ICDE '21. New york: IEEE; 2021. p. 2693–2696.
39. Baazizi M-A, Colazzo D, Ghelli G, Sartiani C. Parametric schema inference for massive JSON datasets. VLDB J. 2019. https://doi.org/10.1007/s00778-018-0532-7.
40. Izquierdo JLC, Cabot J. Discovering implicit schemas in JSON data. In: ICWE '13. Berlin: Springer; 2013. p. 68–83.
41. Izquierdo JLC, Cabot J. Jsondiscoverer: visualizing the schema lurking behind JSON documents. Knowl Based Syst. 2016;103:52–5.
42. Frozza AA, dos Santos Mello R, da Costa FdS. An approach for schema extraction of JSON and extended JSON document collections. In: IRI 2018. New york: IEEE; 2018. p. 356–363.
43. Frozza AA, Defreyn ED, dos Santos Mello R. A process for inference of columnar NoSQL database schemas. In: Anais do XXXV Simpósio Brasileiro de Bancos de Dados. Nashville: SBC; 2020. p. 175–180.
44. ISO: ISO/IEC 9075-1:2008 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). ISO. 2008. http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498. Accessed 28 May 2022.
45. Thompson HS, Beech D, Maloney M, Mendelsohn N. XML Schema Part 1: Structures Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028. 2004.
46. Biron PV, Malhotra A. XML Schema part 2: Datatypes second edition, w3c recommendation. 2004. http://www.w3.org/TR/xmlschema-2/. Accessed 28 May 2022.
47. JSON Schema – Specification. 2020–12. https://json-schema.org/specification.html. Accessed 28 May 2022.
48. Brickley D, Guha RV. RDF Schema 1.1. W3C 2014. https://www.w3.org/TR/rdf-schema/. Accessed 28 May 2022.
49. Hitzler P, Krötzsch M, Parsia B, Patel-Schneider PF, Rudolph S. OWL 2 Web Ontology Language Primer (Second Edition). Cambridge: W3C; 2012. https://www.w3.org/TR/owl2-primer/. Accessed 28 May 2022.
50. Lu J, Holubová I. Multi-model databases: a new journey to handle the variety of data. ACM Comput Surv. 2019;52(3):1.
51. Chen PP. The entity-relationship model–toward a unified view of data. ACM Transact Database Syst. 1976;1(1):9–36. https://doi.org/10.1145/320434.320440.
52. Thalheim B. Entity-relationship modeling: foundations of database technology. 1st ed. Berlin: Springer; 2000.
53. Barr M, Wells C. Category theory for computing science, vol. 49. New York: Prentice Hall; 1990.
54. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM. 1970;13(7):422–6.

55. Thompson HS, Beech D, Maloney M, Mendelsohn N. XML Schema part 1: structures. 2nd ed. Cambridge: WC3; 2004.
56. Koupil P, Hricko S, Holubova I. Schema inference for multi-model data. In: MODELS '22 (accepted).
57. OMG: object constraint language specification, version 2.4. OMG. 2014. https://www.omg.org/spec/OCL/2.4/PDF. Accessed 28 May 2022.

Publisher's Note