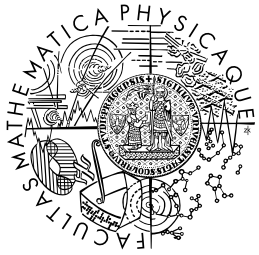

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Sebastián Hricko

Schema Inference for Multi-model Data

Department of Software Engineering

Supervisor of the master thesis: doc. RNDr. Irena Holubová, Ph.D.
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank all those that supported me during the work on this thesis. My thanks goes to my supervisor doc. Irena Holubová and my advisor Pavel Koupil for their patience and exceptional guidance.

Title: Schema Inference for Multi-model Data

Author: Bc. Sebastián Hricko

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Advisor: Ing. Pavel Koupil, Department of Software Engineering

Abstract: In recent years, multi-model databases have become very popular as the individual models better suit the different domains, use cases or scenarios. NoSQL databases are an integral part of the multi-model world of big and variable datasets. While the usage of these databases is relatively simple and functional, in some cases, we lack insight into the structure of the data and the possible interconnection between the data in various databases and different models. This thesis presents a novel approach that generates a schema of the multi-model data concerning the undeclared relationships between the models. Firstly we analyse the existing single-model approaches and point out the main flaws. We then propose the universal multi-model approach and implement it as a proof of concept.

Keywords: Multi-model, Schema inference, NoSQL databases

Název práce: Odvozování schématu multi-model dat

Autor: Bc. Sebastián Hricko

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: doc. RNDr. Irena Holubová, Ph.D., Katedra softwarového inženýrství

Konzultant: Ing. Pavel Koupil, Katedra softwarového inženýrství

Abstrakt: V posledných rokoch nabrali multimodelové databázy na popularite, pretože jednotlivé modely lepšie vyhovujú rôznym doménam alebo prípadom použitia. NoSQL databázy sú neoddeliteľnou súčasťou multimodelového sveta veľkých a variabilných dátových sád. Aj keď je použitie týchto databáz relatívne jednoduché a praktické, v niektorých prípadoch postrádame vhľad do štruktúry dát a informáciu o možnom prepojení medzi dátami v rôznych databázach a rôznych modeloch. Táto práca predstavuje nový prístup, ktorý generuje schému multimodelových dát, pričom sa berú do úvahy vzťahy medzi modelmi. Najprv analyzujeme existujúce prístupy navrhnuté pre samostatné modely a poukážeme na ich hlavné nedostatky. Potom navrhneme univerzálny multimodelový prístup a implementujeme ho ako proof of concept, čiže overenie realizovateľnosti.

Klíčová slova: Multi-model, Odvozování schématu, NoSQL databáze

Contents

1. Introduction

In recent years, thanks to the variability of the data stored nowadays, it is common to use more than a single data model in one solution. Various data models are better suited for various data and use cases. For example, consider an online e-commerce store that sells products to customers as depicted in Figure 1.1. The data about the products may be better suited for the document data models due to the high variability of the information about the products. At the same time, the shopping cart's content is described with a key-value data model to be able to look up the simple relations fast. In addition, the customers can have relationships with each other, and thus we would like to store this information in a graph model and then recommend the products that their friends bought recently.

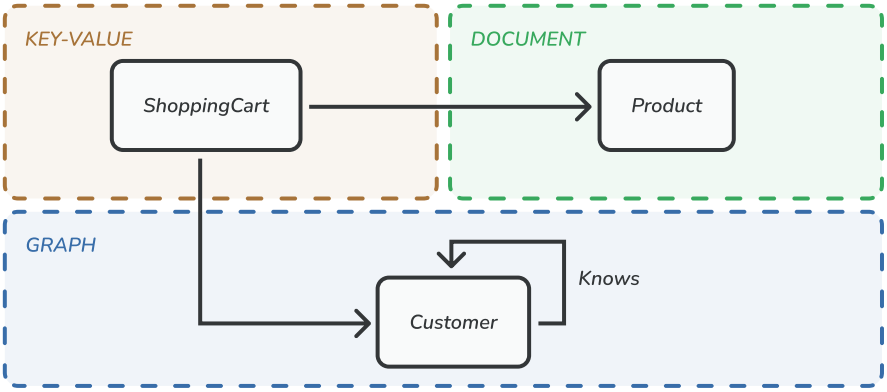

Figure 1.1: Multi-model scenario of an e-commerce store

The described example is considered a multi-model scenario, where the data are stored in different data models. Yet, they might be a part of a single solution and thus be interconnected. Due to the high variability of the data stored in each model, the schema describing the structure of the data or the data itself is often not specified in advance. We call this a *schema-less* variant as the opposite of the *schema-full* variant. In the schema-full variant, the data stored in the database has to follow the predefined schema description. This variant is iconic and fundamental for the relational data model, the most traditional one presented here. With the dawn of the NoSQL databases, the popularity of the schema-less concept highly increased. Combining the two variants results in a *schema-mixed* variant, which allows the possibility to predefine the structure of the part of the data, leaving the other part without additional restrictions.

When we want to analyse the data stored in schema-less databases, it is necessary to have some knowledge about the structure or domain of the data in order to write reasonable queries. Various approaches have been proposed in the past that can analyse the data stored in the specific data model and generate a more or less detailed view of the structure of the data. However, suppose we face the single model approaches against the multi-model scenario. In that case, we get a diverse result that lacks any information about the relationships between the data stored in the models.

Considering these flaws, in this thesis we want to propose a multi-model approach that is able to extract the schema of the data from widely used data models (namely relational, document, key-value, columnar and graph model) and concerning the schema-full, schema-less as well as schema-mixed variants. The approach will be able to unify the data models to produce a consistent result. In addition, it will be able to infer the relationships between the data within a single data model as well as between distinct data models. We will also observe the similarities in the data to discover data redundancies that are common in the world of NoSQL data.

The vision paper [1] can be considered as the basis for this thesis. The authors discussed the open problems and challenges of a multi-model database. As for the case of the schema inference, they noted that the proposed multi-model schema inference approach should deal with references between records as an extension on top of the existing single-model approaches.

This work is organised as follows: In Chapter 2, we will recall the necessary preliminaries used throughout the thesis. In Chapter 3, the existing approaches and other related work will be presented. Chapter 4 describes the proposed multi-model approach. In Chapter 5, we will describe the implementation of the proof of concept of the proposed approach. Chapter 6 will present the results of the experiments of running our approach. In Chapter 7 we conclude and outline the future work.

2. Preliminaries

Throughout this thesis, we will mainly consider the multi-model scenario based on the IMDb dataset [2]. We have slightly modified the dataset to serve our purposes better. The high-level overview can be seen in Figure 2.1. The IMDb dataset contains the information about the movies and the people behind the movies (e.g. actors, writers, directors etc.) and also the total rating of the movies. We placed the individual components of this scenario inside various models considering their intended use cases. In addition, we introduced three components - *User*, *Review* and *FavouritePeople* to describe all the considered data models.

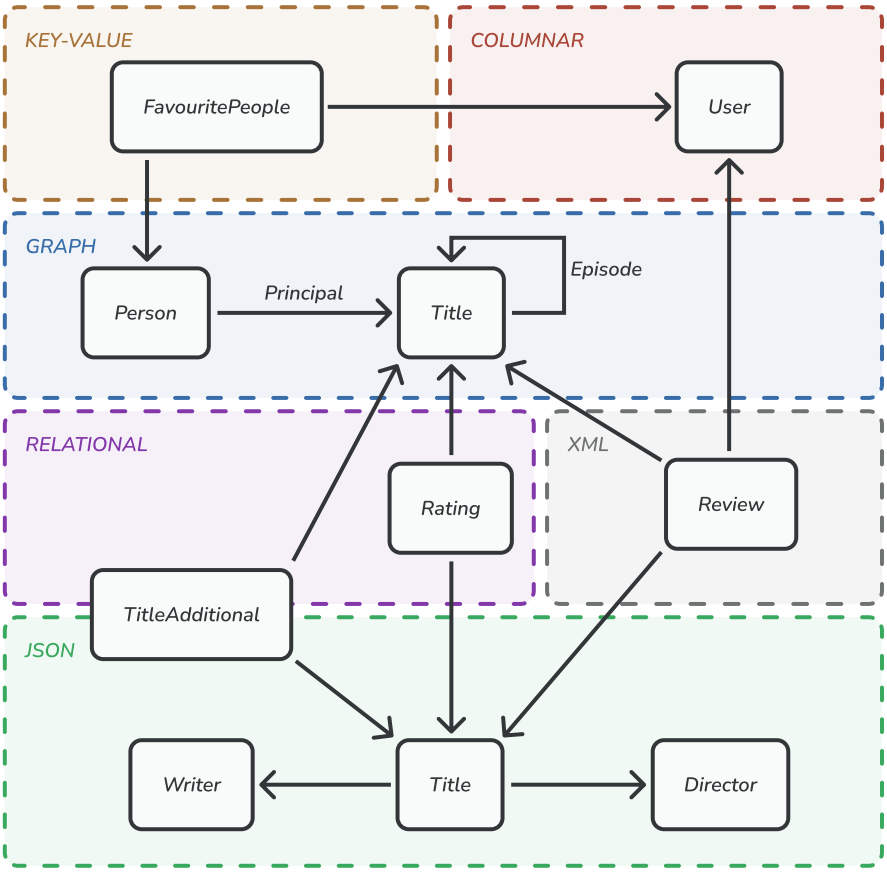

Figure 2.1: Modified multi-model scenario of IMDb dataset

2.1 Data models

This section presents an overview of the data models that we will consider in this thesis for the multi-model schema inference approach.

2.1.1 Relational Model

The most traditional data model for storing data is the relational model. Data in relational databases are stored in relations. The relation (table) comprises a set

of tuples (rows in the table), and each tuple shares a set of attributes (columns in the table). Each row in the table has a unique identifier called the key. It is common for this model to contain a predefined schema, which the individual records then follow. All records represent real-world entities or relationships and are grouped into tables.

Example. An example of relational data is depicted in Figure 2.2. The *Rating* table is an example of single-model data, while table *TitleAdditional* contains document inside its column *additionalData*. This scenario can be achieved using PostgreSQL [3] database system using special data type *jsonb*.

□

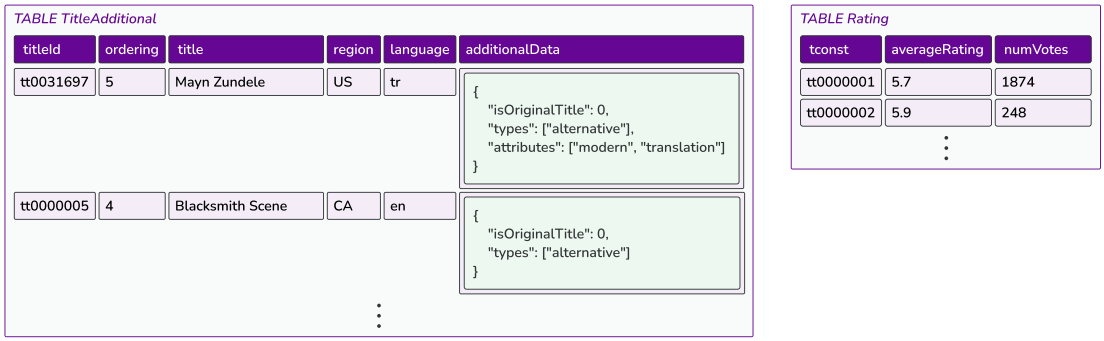

Figure 2.2: Example of relational data

2.1.2 Document Model

Document models are semi-structured, meaning they lack rigid or fixed schema. While the data do not conform to the explicit schema, they contain metadata and tags that group elements together and thus create a hierarchy. Entities can be grouped under the same type, even if they do not have exactly the same structure. There are two principal representatives of the document model – JSON and XML.

JSON

JSON [4] has become one of the most widely used data exchange formats on the Internet in recent years. It comprises two fundamental structures – an unordered collection of key-value pairs (objects) and an ordered list of values (arrays).

Example. An example of JSON document data is depicted in Figure 2.3. The collection *Title* groups together the JSON documents that describe individual titles. The structure is flexible, as can be seen by comparison of field *writers*. In the first document, this field is an array, containing a single nested document, while in the second document, the field *writers* is not present. One of the most popular database system, that are based on JSON document data model is MongoDB [5]. □

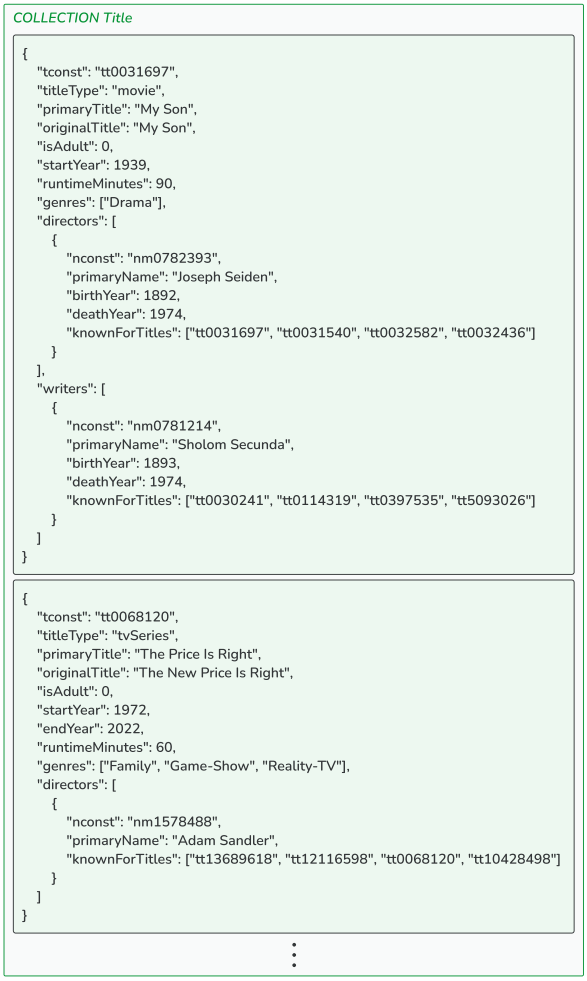

Figure 2.3: Example of JSON data

XML

Although JSON is more popular today, XML [6] is still a widely used standard for representing information in documents. To represent the data, it uses a tree structure of elements containing values of simple types or nested elements. Additionally, elements can contain simple types of attributes.

Example. In Figure 2.4 we can see an example of XML document. This data can also contain a variable structure of user reviews. Comparing the two documents, the field *text* is optional because it is missing in the second document.

□

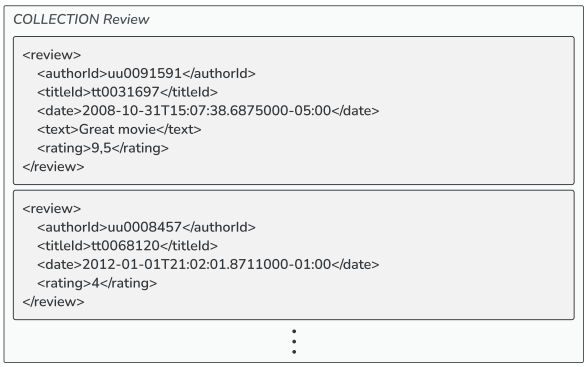

Figure 2.4: Example of XML data

2.1.3 Key-value Model

The simplest data model of the ones we discuss is the key-value model, in which the data is stored in key-value pairs grouped in a so-called bucket. This paradigm is used for storing and managing associative arrays, hashes, and dictionaries.

Example. Figure 2.5 represents a bucket *FavouritePeople* which stores information about which user likes which actors, producers and others. The data are identified by *userId* and are represented by an array of *PersonIds*.

□

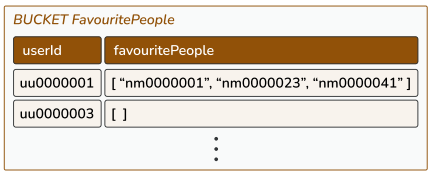

Figure 2.5: Example of key-value data

2.1.4 Columnar Model

A columnar model may be viewed as an extension of the key-value model, where one key uniquely identifies a set of other key-value pairs – a row. These rows

are logically grouped into column families. This data organisation resembles relational models, but in contrast, rows in column families can have a variable structure.

Example. Figure 2.6 describes a column family that holds the data about the users. In this example, the primary key is stored in column *userId*. Note that the optional columns needn't be present in the specific row compared to the relational model where the columns are fixed and optionality is achieved with the usage of empty values.

□

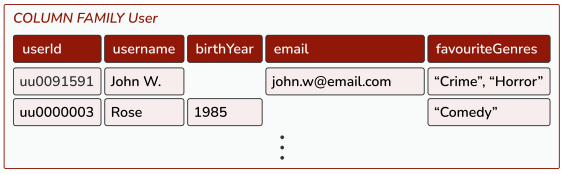

Figure 2.6: Example of columnar data

2.1.5 Graph Model

Nowadays, databases based on the graph model are increasingly being used to describe the relationships and interconnections of entities. The graph model is based on a mathematical representation of graph theory, hence it consists of graph entities – nodes and edges. Graph nodes are tagged with labels, yet they also contain a free data structure in the key-value format. The edges of the graph are directed, they contain a reference to the start and end nodes beyond the structure of nodes.

Example. In Figure 2.7 the interconnections between the nodes labelled *Person* and *Title* are described with the usage of a relationship of type *Principal*. The relationship type connects only the nodes labelled *Title*. Each node has its identifier, while every edge contains references to start and end nodes. □

2.2 Processing Large Datasets

To be able to process large-scale datasets, we want to take advantage of a framework that enables us to process the data in parallel and ensures scalability. We will consider two options: MapReduce [7] and Apache Spark [8].

2.2.1 MapReduce

Presented in paper [7], MapReduce is a programming model and implementation that gives an opportunity to process large-scale datasets in a simple manner. The execution is based on a divide-and-conquer paradigm and is composed of two principal phases – *map* and *reduce*. During *map* phase, we break down a problem into subproblems by processing a key-value pair to generate a set of intermediate

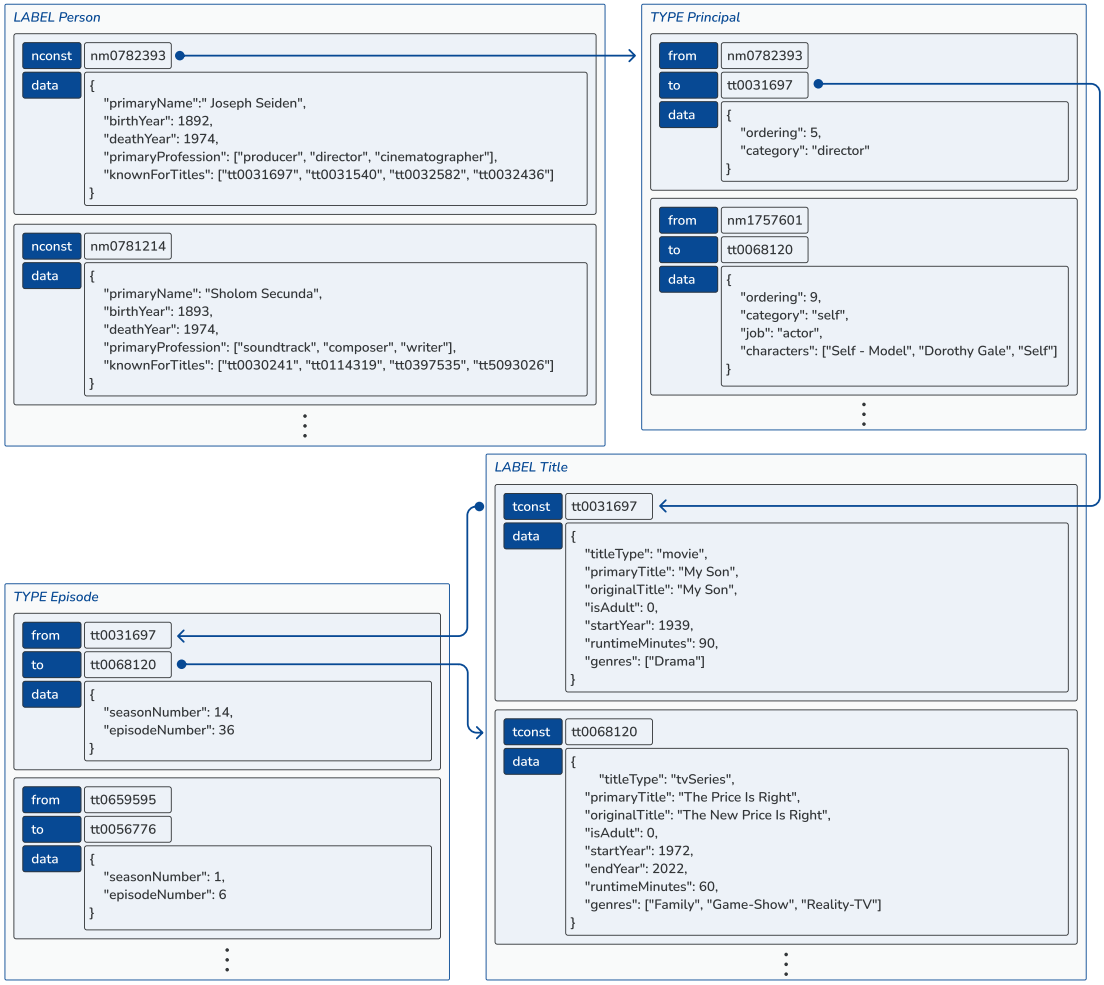

Figure 2.7: Example of graph data

key-value pairs. In the *reduce* phase, we combine the subsolutions to solve the problem by processing intermediate values with the same intermediate key.

When processing the user program, multiple copies of the program are created on a cluster of machines. One copy of the program is called master, which is responsible for assigning work to the other copies, called workers. When all the subproblems are solved by the workers, the master wakes up the user program and points him to the result.

2.2.2 Apache Spark

The Apache Spark [8] is based on the MapReduce model, in addition, it supports in-memory cluster computing. Apart from the support of the MapReduce model, it introduced higher-level tools like Spark SQL, Spark streaming, MLlib for machine learning and GraphX for Graph algorithms. Spark executes operations on a cluster in parallel. The cluster consists of worker nodes that execute assigned tasks and provide the results. An overview of the Apache Spark cluster architecture can be seen in Figure 2.8.

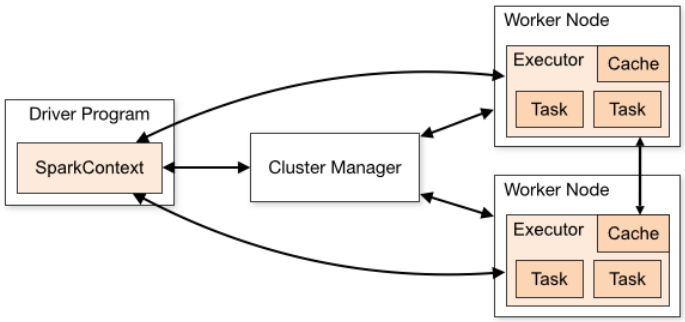

Figure 2.8: Apache Spark cluster [9]

In contrast to the basic MapReduce model, it is able to process the intermediate results in memory, while MapReduce is reliant on the shared file system. Moreover, it supports more functions than simply *map* and *reduce* functions, e.g. *flatMap* function that we will utilise in our approach when we will need to flatten the processed tree.

The basic building block of the Spark processing is an in-memory fault-tolerant collection of elements that can be operated on in parallel called a *resilient distributed dataset* (RDD). We can create RDDs in two ways: either by parallelising a collection of elements or referencing a dataset in a storage system. In our approach, we will be using the latter mentioned.

Due to the reasons mentioned above, we will use the Apache Spark framework in our approach. We will use two notations throughout this thesis, $RDD(x)$, or $RDD(x, y)$. In the former case, the notation describes that the element x is in the form of a resilient distributed dataset. In the latter case, x stands for the key that identifies y – the value. The pair is also in the form of a resilient distributed dataset.

3. Analysis of Existing Approaches

To the best of our knowledge, there is no other approach for multi-model schema discovery. However, there are multiple approaches that deal with NoSQL data in a single-model manner. In this chapter, we inspect the existing schema inference approaches for each model. We choose one approach for each of the models as a representative. We then compare them to find the weaknesses of the approaches, and thus identify the places that need improvement. Our multi-model approach will then be based on these findings.

3.1 Relational Model

For the world of single-model database systems, it may be irrelevant to infer a relational schema algorithmically, because the user defines the schema in advance using the *Data Definition Language* (in the context of SQL). But if we place the relational model in the context of multi-model databases, inference, in order to reveal references and redundancies between models, starts to make sense. However, since we do not know of any other approach that would generate a schema for multi-model databases, we will omit this model in the comparison.

3.2 Document Model

The building block of the document model databases is a document with two principal representatives, JSON and XML. Due to the fact that these formats were popular at different times in the past and they contain differences (e.g. the mandatory order of the elements) in the description of the schema, various approaches were proposed for the individual representatives. Hence, we will look at the approaches individually based on the format of the documents.

3.2.1 JSON Schema Inference Approaches

As for the JSON-based document model databases, there is considerable research on schema discovery. In the paper [10], the authors compared selected JSON schema inference approaches to point out the flaws of existing approaches as well as their strengths.

Paper [11] was one of the first to describe the process of inferring the schema from multiple JSON documents. They proposed a model-based process that makes use of different consecutive calls to the service. JSON documents from the responses are then processed in order to create or refine the schema objects. Finally, a basic approach to resolving the similarity of elements is used to find and merge matching elements. This results in a model that summarises the application domain.

Another approach [12] based the schema generation algorithm on graph theory. The inspiration for such an approach was another research aimed at extract-

ing DTDs from a set of XML documents. When processing documents, trees are created with a unique document key. The trees are then merged into a single tree, with each tree entity carrying information about the documents it occurred in. This approach can reveal mandatory and optional properties and at the same time provide statistics on how regular the data in the collection are.

The approaches mentioned so far have worked on the basis of enrichment and refinement of the created schemes. In [13] they introduced a completely new way of obtaining the schema. The approach is based on the property of Big Data – variety. The idea is that the documents in the same collection can change structurally over time, creating new versions. Therefore, this approach generates a set of simple document schemas, further reducing this set to only a set of structurally different schemas. We then receive a separate version for each of these schemas. Although this approach generates a relatively complex structure, in some cases, the result is usually much finer than previous approaches. In addition, this approach, using simple heuristics, looks for references between the documents.

Baazizi et al. and Equivalence Relations

Due to the growing popularity of Big Data usage, the need to derive a schema from large JSON data arose. One of the ways [14] is focused on massive JSON datasets. The MapReduce principle implemented in Apache Spark is used for document processing. This approach benefits from the possibility of running the algorithms in parallel. A simple notation is used to represent schemas using modified regular expressions. However, this approach does not provide the ability to look for references between documents.

Let us analyse this approach even further, as our reducing approach will be inspired by the fusion presented in this paper. This approach makes use of parameterisation of the reduction of inferred schemas, thus creating more or less fine or verbose results. The reduction is a binary operation invoked over two schemas and returns their generalisation. In order to do so, the schemas are compared based on the parametrised equivalence relation. Suppose the two types are equivalent based on the equivalence. In that case, they are fused, or else the result is just a union of the two schemas.

Two equivalence relations were described in detail: label-equivalence and kind-equivalence. The first mentioned *label-equivalence* fuses only those schemas that contain the same set of labels. When two schemas share different labels, they are joined with a union instead of being fused. The latter, *kind-equivalence*, will be used in our approach with slight modification. This equivalence variant fuses two schemas when they are similar primitive types, are both maps or arrays with no other condition. The primitive types are joined with a simple union. In the case of fusing two maps, the keys in the maps are compared, and the keys without a match are just copied to the fused record type and are marked as optional. On the other side, the matching keys are merged, and their types are recursively fused.

Example. Suppose two schemas of documents s_1 and s_2 in Figure 3.1. Their fusion with respect to kind-equivalence is depicted in Figure 3.2. The result of the fusion based on label-equivalence can be seen in Figure 3.3. Note that much

```
{
person:
  {
   firstname: Str,
   lastname: Str,
   coordinates: [Num + Null]
  }
}
```

(a) Schema s_1

```
{
person:
  {
   firstname: Str,
   lastname: Null,
   coordinates: [Num],
   email:  Str
  }
}
```

(b) Schema s_2

Figure 3.1: Schemas to be fused together [14]

broader domain of documents satisfies the schema of the kind-equivalence fusion comparing the label-equivalence fusion. However, the label-equivalence tends to be more verbose when working with a large variety of the data.

```
{
person:
  {
   firstname: Str,
   lastname: Str + Null,
   coordinates: [Num + Null],
   email:  Str?
  }
}
```

Figure 3.2: Fusion of s_1 and s_2 considering the kind-equivalence [14]

```
{
person:
  {
   firstname: Str,
   lastname: Str,
   coordinates: [Num + Null]
  } +
  {
   firstname: Str,
   lastname: Null,
   coordinates: [Num],
   email:  Str
  }
}
```

Figure 3.3: Fusion of s_1 and s_2 considering the label-equivalence [14]

Wang et al.

The authors of approach described in paper [15] proposed the schema management framework for JSON document stores. The framework discovers and persists the structure of the data in their own proposed data structure called *eSiBu-Tree*. This custom data structure serves as a way to retrieve and store schemas and

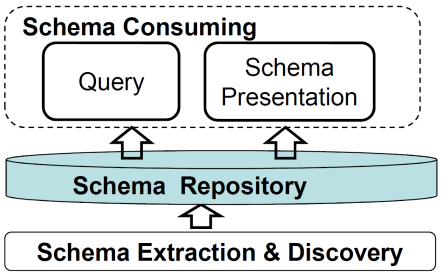

Figure 3.4: Schema management framework [15]

support queries. For the presentation part towards the user, they proposed a *Skeleton* data structure, which summarises individual generated schemas.

The schema management framework depicted in Figure 3.4 is composed of the following components:

1. *Schema Extraction and Discovery* serves as a generator of all distinct document schemas for each input document. This component operates incrementally, so when a new document is being processed, its schema is compared with the currently existing document and persisted if it describes a new version.

2. *Schema Repository* handles the persistence of existing schemas. It also supports the extraction process and schema consumption by the following components.

3. *Query* is a component that provides the functionality to find the exact answer to basic types of queries – schema existence and attribute existence.

4. *Schema Presentation* provides a summarised representation of generated schemas in the form of Skeleton. Skeleton is a generalisation of detailed individual schemas and contains the schema of the highest quality, while the quality is based on the frequency of the schema.

3.2.2 XML Schema Inference Approaches

There is a significant number of approaches and methods for generating an XML schema for a set of documents. Paper [16] describes and compares various approaches. From those, we choose the XTRACT approach [17], which, besides naïve approaches, uses more sophisticated techniques to generate a meaningful description of the resulting schema. XTRACT is a system for inferring a DTD schema for a database of XML documents. The process focuses on individual elements across the input documents and generates a sequence of nested elements for each element with a complex structure. The architecture can be seen in Figure 3.5.

For each such element the following process is performed:

1. *Generalization* uses heuristic algorithms to find repeating patterns of element structures and replace them with regular expressions in input sequences.

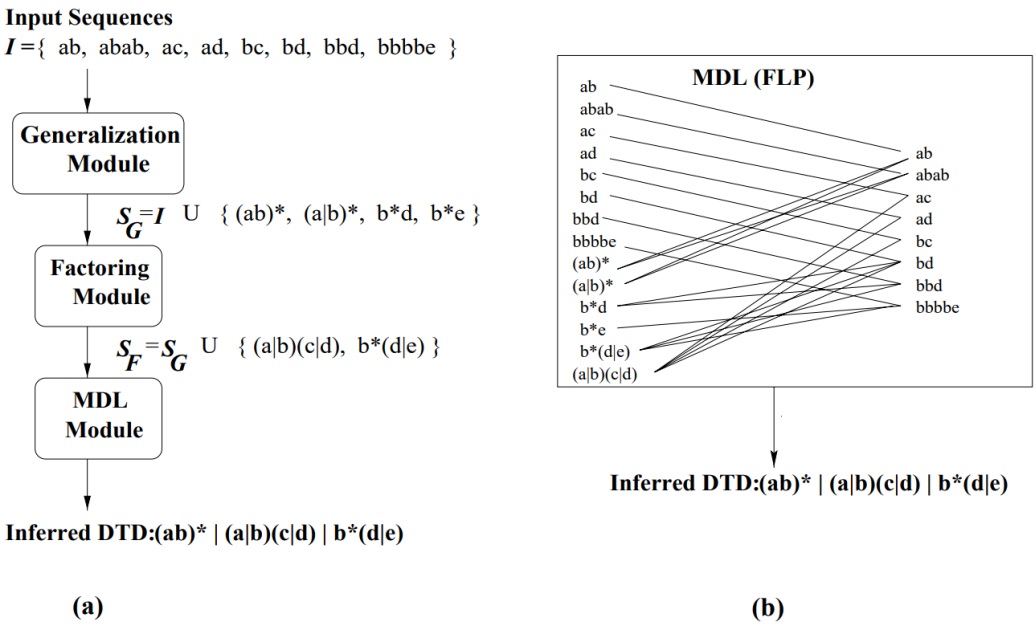

Figure 3.5: Architecture of the XTRACT system [17]

2. *Factoring* step tries to factor individual regular expressions so that the length of the DTD is less than the sum of the factored DTDs. This factoring does not change the semantics of individual candidate DTDs. This step is important because of the third step, when factoring can create a DTD that results in a higher ranking and is therefore considered for the resulting DTD. The authors propose heuristics on how to reduce the combinations of DTDs that will be factored.

3. *Minimum Description Length (MDL) Principle* is used in order to select the best candidate DTDs to represent the final schema. An almost optimal DTD schema describing the input data set is created. The MDL evaluates each DTD in two ways based on the number of bits needed to represent it.

The authors declare that their research showed that in most cases their approach produced a DTD that was identical or very similar to the originally intended DTD.

3.3 Key-value Model

Presumably the only existing schema inference approach for the key-value model is described in paper [18]. It follows the prior approach for generating JSON document schemas [19]. This approach generates a schema over the key-value database Redis [20]. It distinguishes two basic data types:

1. *JSON document* – value is a valid JSON document. A raw schema is generated regarding the hierarchical structure of the attributes of the JSON document.

2. *Byte sequence* – otherwise, the value is a generic sequence of bytes. The raw schema of the instance has only two attributes – key and value, both of the String data type.

After the raw schemes of all records are generated, the next steps are executed in the same way as in the document approach. Thus, the raw schemas are compared with each other, the same schemas are grouped, unification occurs, and finally, the global JSON schema is constructed.

3.4 Columnar Model

A few approaches to generate schema of data stored in columnar models were proposed, one of which is described in paper [21]. This approach traverses a single namespace in a HBase [22] columnar database. The process depicted in Figure 3.6 is rather simple and works as follows:

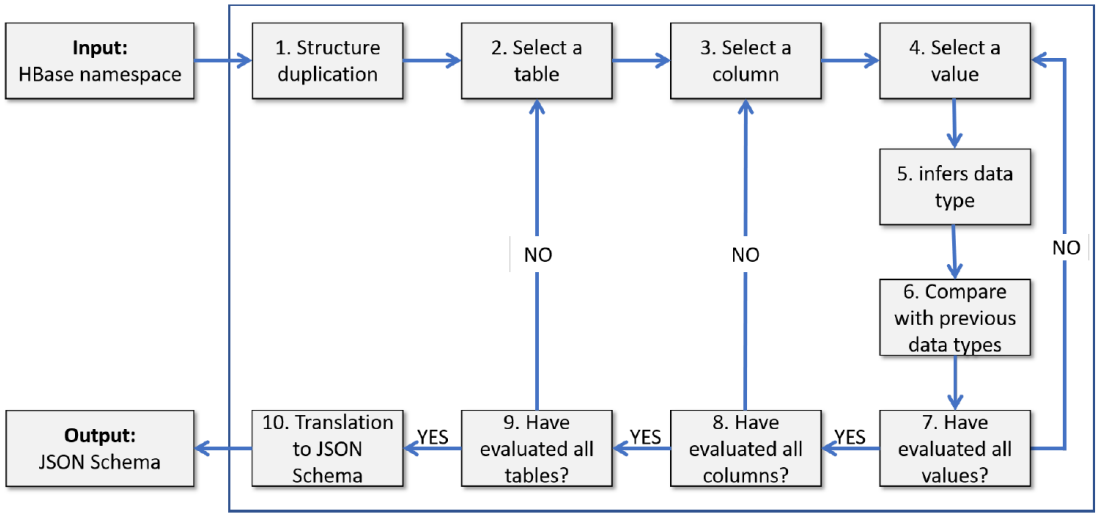

Figure 3.6: The process of converting HBase namespace into JSON schema [21]

1. *Data type inferring* – For every value in each column in each table in the namespace, the data type is inferred.

2. *Combining data types* – When all values in a column are processed, the data types of processed values are compared with each other to define a consensual data type.

3. *Generating global JSON schema* – With all data types discovered, the JSON schema describing the namespace is created.

With this method, no references or integrity constraints are discovered. However, the inference of mandatory and optional columns is considered for future work.

3.5 Graph Model

To the best of our knowledge, to date, there is no other study of the schema inference of graph models than the one proposed in paper [23] for the popular graph database Neo4j [24]. The authors set themselves a goal of creating an approach that could derive simple and complex types of graph entities, reveal the hierarchy of nodes, and also derive the cardinality of edges. The schema generation process depicted in Figure 3.7 works as follows:

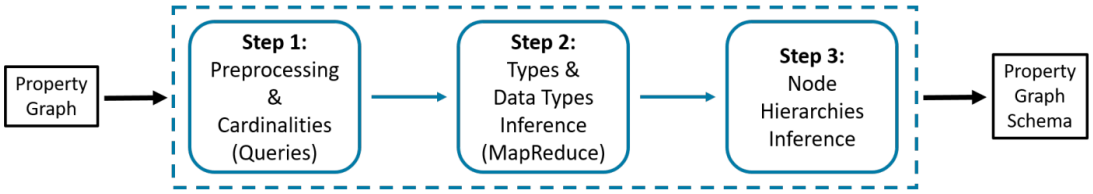

Figure 3.7: Graph schema inference process [23]

1. *Preprocessing* – records corresponding to nodes and edges are extracted using queries. By comparing the number of occurrences of labelled nodes and the number of edges where the source or target is the labelled node, the cardinality of individual types of edges is further determined. Finally, the node and edge information is transformed to JSON for further processing in the second step.

2. *Types and Data Types Inference* – this step is implemented using the MapReduce principle. In the Map phase, the data types of the individual properties are inferred. In the Reduce phase, the fusion takes place according to an equivalence relation. The proposed solution uses the so-called kind-equivalence described in [14], which forms union types.

3. *Inference of Node Hierarchies* – in the last step, hierarchies of node types are discovered. The names of properties of inferred types are compared, and if a pair A, B is found for that it holds that $A \subset B$, then B is supertype and A is its subtype.

3.6 Comparison of Selected Approaches

After looking at the individual database models and the representatives of the schema generation approaches, we now proceed with the comparison of their properties. Table 3.1 compares approaches based on several properties. In this section, we look at each of the properties and describe the extent to which the approaches achieve the following qualities.

3.6.1 Data Types

Most of the schema inference approaches across various database models extract the data types of elements into the schema. However, there are still some approaches, where the authors decided not to resolve data types, as we can see

Table 3.1: Comparison of schema inference approaches

	Wang et al. [15]	**XTRACT** [17]	**REx** [18]	**Frozza et al.** [19]	**Lbath et al.** [23]
Model	Document (JSON)	Document (XML)	Key-value	Columnar	Graph
Input	MongoDB collection	Set of XML documents	Redis bucket	HBase namespace	Neo4j database
Output	Skeleton	DTD	JSON Schema	JSON Schema	Property graph schema
Process	Generating traversable eSiBu-Tree iteratively	Generating candidate DTDs and using MDL principle to find the best suiting one	Mapping records to raw schemas, followed by unification	Generating data type of each value to create raw schemas, followed by unification	Preprocessing followed by MapReduce and finished with inference of node hierarchies
Data Types	No	Yes	Yes	Yes	Yes
Optionality	Yes	Yes	Yes	No	Yes
Uniqueness	No	No	No	No	No
References	No	No	No	No	No
Redundancy/ Subtype	No	No	No	No	Yes
Union Type	No	Yes	Yes	Yes	Yes
Type Versions	Yes	No	No	No	Yes

in the case of the approach by Wang et al. In other cases, the data types are resolved based on the values of elements. An interesting case can be found in the approach by Frozza et al. over the HBase database system. This system stores data only in terms of byte arrays, thus the authors presented a set of rules to determine the most common data types. The rules are based primarily on the length of the arrays.

3.6.2 Optionality

Determining whether a given element in a dataset is mandatory or optional is the most common access property, in addition to detecting data types. From the representatives we have selected, only the approach of Frozza et al. in the columnar model does not resolve the optionality. However, in their article, the authors describe that their approach is prepared for this extension and is planned for future work. Other approaches deal with the optionality of elements as follows:

Wang et al. generates an eSiBu-Tree tree in which individual versions of elements are recorded. From these versions, it is possible to read whether any of the child elements are optional.

XTRACT system generates a DTD with regular expressions from the input XML document set that supports the ? operator, which says that the specified element is optional.

REx translates records into JSON raw schemas and, above that, it determines the number of occurrences of the element. If the number of these occurrences is lower than the number of parent occurrences, the element is marked as optional.

Lbath et al. determines the cardinality of the edges in the Preprocessing phase and observes structural differences in the Reduction phase, which in the end can express the optionality of individual elements.

3.6.3 Uniqueness

The element is called unique if and only if all the values of the element are distinct regarding the input data. To the best of our knowledge, none of the discussed approaches can distinguish between unique and non-unique elements. It is due to the fact that all selected approaches analyse only the structure of elements, while they do not consider the values. Inferring uniqueness is crucial when inferring various integrity constraints.

3.6.4 References

As like in the aforementioned case of resolving uniqueness, none of the approaches manage to resolve advanced references between the elements. In most cases, simple aggregate relationships are being inferred, i.e. the built-in parent-child relationship of the JSON documents. The graph model approach by Lbath et al. can resolve references based on the labels of the source nodes and target nodes of the edges.

3.6.5 Redundancy/Subtype

Only one of the approaches presented here, to some extent, looks for similarities between elements in order to find subtypes. It is the graph model approach by Lbath et al. that compares the labels in the structure of nodes and, based on that, is able to create a hierarchical structure of nodes. However, as said above, the domain analysis of the element values needs to be implemented in order to be able to enhance inferring redundancies and other similarities.

3.6.6 Union Type

Production of a union type is based on the fusion of the various raw schemas of an element. Only the document model approach by Wang et al. is not capable of producing union types of the inferred elements.

3.6.7 Type Versions

In contrast to union types, distinct raw schemas of a single element are considered to be different schema versions of the element. The approach by Wang et al. is generating multiple versions based on the introduction of raw schemas of elements into the process. The approach by Lbath et al. is able to choose between the two reduction equivalences, the label-oriented variant, which produces the union type, or the kind-oriented variant, producing type versions.

4. Proposal of a Multi-Model Schema Inference Approach

In this chapter, we present the proposed approach for generating a schema for multi-model data. First, we will look at a naïve approach, which we will gradually enrich in order to optimise the processing of the algorithm or to enhance the quality of the result.

4.1 Definition of the Problem

The fundamental problem we need to solve is to obtain the schema from multiple database systems, for which it is assumed that the data contained therein will be interconnected. Note that database systems are available in three variants, differing in the existence of the schema. We distinguish between schema-full, schema-less, and schema-mixed variants.

The first mentioned variant says that the data in the database adheres to the data structure, which is defined in advance. The schema-less variant is the exact opposite – the data in the database does not follow the explicitly defined data structure, but the structure is irregular. In this variant, the schema is not explicitly specified, but there is still an implicit schema that the query system maintains. A combination of the mentioned variants is the schema-mixed case, where only some part of the data conforms to the specified schema, while the other part can have variable structure. PostgreSQL [3] database system is an example of this variant – the relational part of the data has to conform to the predefined schema, while the data of the data type jsonb are stored without an explicit schema.

While it is possible to apply different approaches mentioned in Chapter 3 to each of the models in the multi-model data, the approaches differ in the details they provide in the resulting schemas. This can lead to irregular results, which we intend to avoid and solve the inconsistency problem. Besides, the single-model approaches will not be sufficient for exploring the relationships, i.e. references and redundancies. Hence, in order to accurately analyse not only the structure of the data but also the semantics, we need to process raw data in a low-level manner.

For this reason, we will create a unified system for each model. This system helps us to create a universal approach, for which it does not matter from which database model the data come, because all data will be processed in the same way.

Example. An example described in Figure 4.1 shows the multi-model data from various database models as presented in Chapter 2. Each colour corresponds to the particular data model, i.e. *purple* stands for relational data model, *green* is for the JSON document data model, *grey* is for the XML data model. Key-value data model is marked *brown*, columnar model is *red* and finally the graph model is *blue*. In this section, we will consider this model with minor modifications. □

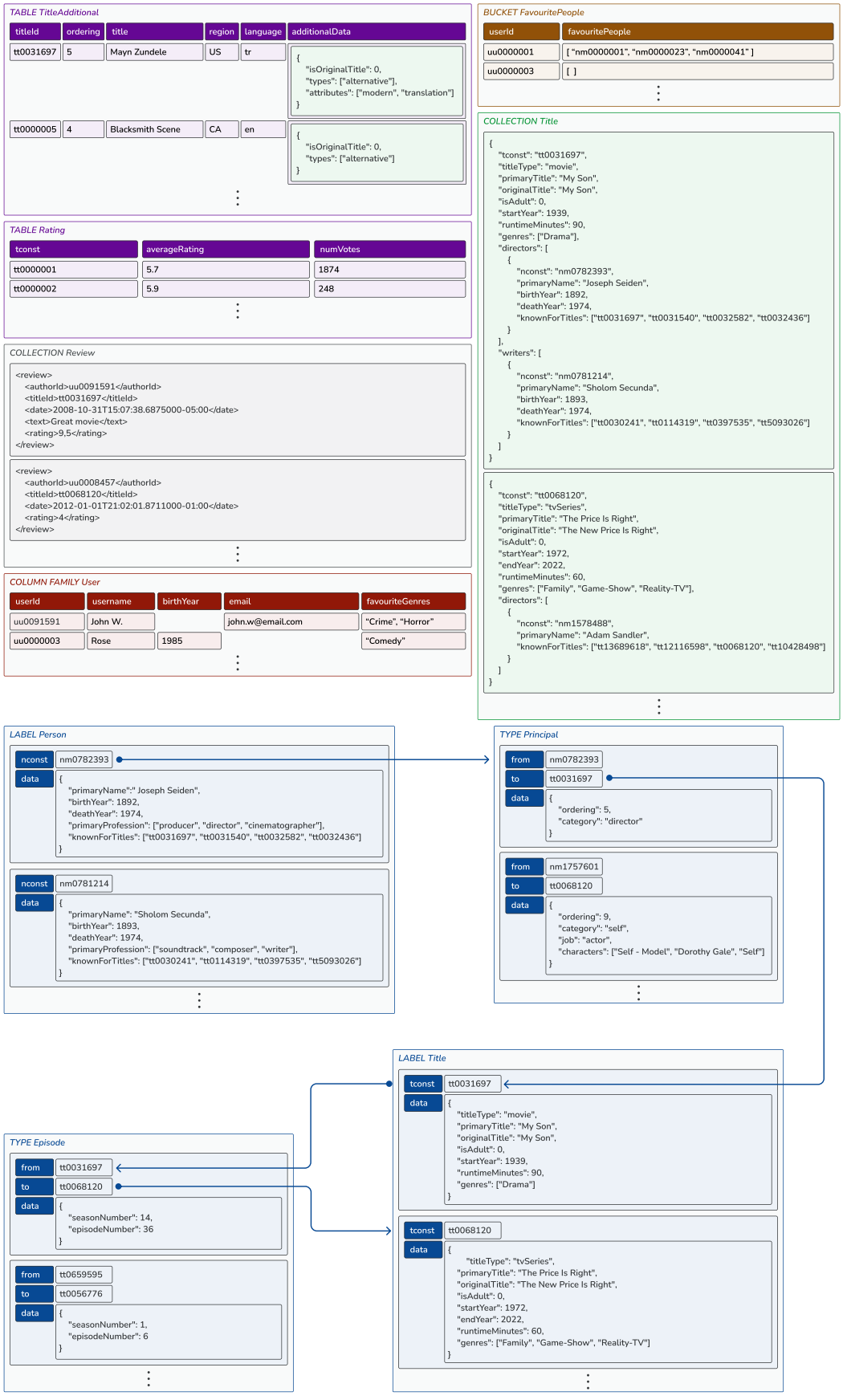

Figure 4.1: Example of multi-model data

4.2 Unification of Models

As discussed in the previous section, we need to unify all processed models to build the universal algorithm. We will achieve this unification by building a so-called *wrapper* for every database system. The wrapper should produce a consistent output for each considered data model. While some concepts in various data models are similar, they are not the same, and thus we need to unify them.

4.2.1 Unification of Terms

Before we can proceed, we need to unify the basic terms used in different models. Table 4.1 shows the fundamental terms in various models and unifies similar concepts into single terms, which we will use in the rest of the text.

Table 4.1: Unification of terms in popular models

Unifying term	**Relational**	**Graph**	**Key-value**	**Document**	**Column**
Kind	Table	Label / Type	Bucket	Collection	Column family
Record	Tuple	Node / edge	Pair (key, value)	Document	Row
Property	Attribute	Property	Value	JSON Field / XML element or attribute	Column
Domain	Data type	Data type	–	Data type	Data type
Value	Value	Value	Value	Value	Value
Identifier	Key	Identifier	Key	JSON identifier / XML ID or key	Row key
Reference	Foreign key	–	–	JSON reference / XML keyref	–
Array	–	Array	Array	JSON array	Array
Structure	–	–	Set / ZSet / Hash	Nested document	Super column

4.2.2 Database Wrapper

A database wrapper is a module with the purpose of providing the data from the database in a unified form. Basically, it generates a raw schema from a database instance in the database system for which this wrapper is intended. The wrapper extracts all the requested records from the database using simple queries that are specially designed for the database system.

We need to choose the most general representation of the records. The representation has to respect the tree-like structure in order to describe the records together with their properties. It will also contain the basic information we can generate at the lowest level of the structural analysis – while processing the data by the wrapper. That is the name of a property, the data type, information

about the optionality of a property and information about the data model that this property is stored in.

We also can distinguish between different wrappers even for the same database system. As stated above, database systems can either be schema-full, schema-less, or schema-mixed. With this knowledge, we can build a schema-full wrapper that retrieves the schema and converts it to the unified raw schema or a schema-less wrapper that does not process or ignores the predefined schema and tries to infer the schema with a universal approach instead.

4.3 Naïve approach

Having unified the various models and being able to generate records, we now describe the process of inference of the schema from these records. We start with the naïve approach and gradually add optimisations and enhancements to reach the set goal.

4.3.1 Inference Algorithm

We implement the inference algorithm utilising the MapReduce pattern (see Chapter 2). The process can be seen in Algorithm 1. Let us now break down each step in the algorithm and discuss them in the following sections.

Algorithm 1: Record-based local schema 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 – Set of RSDs describing the schemas of input kinds
2 Schema $S := \emptyset$
3 **foreach** $name_\kappa$ in N_D **do**
  `// map phase: load and map each record of a given kind to a separate trivial RSD`
4 $RDD(r) := w_D$.mapRecordsToRSDs($name_\kappa$)
  `// reduce phase: each collection of RSDs sharing the same` $name_\kappa$ `is reduced into a single RSD describing a schema of kind` κ
5 $r_\kappa := RDD(r)$.reduce()
  `// schema collection phase`
6 S.add(r_κ)
7 **return** S

Map Phase

The fundamental part of schema inference is generating the structure and data types. Inferring the structure of individual records is a relatively simple task. It is necessary to traverse the record and consider the aggregation relationships. In doing so, we can also analyse the data types of properties with primitive values. Besides that, we will be generating additional information about the individual properties.

Record Schema Description. We will now introduce the Record Schema Description (RSD) tree-like structure. It will serve multiple purposes as the representation of the following:

- raw schema of a single record generated by the wrapper
- the intermediate schema processed by the algorithm
- the final result

Definition 1. *Record Schema Description* is a *9-tuple (name, unique, share, id, types, models, children, regexp, ref)* consisting of

- a *name* of a property or when the property is not named, then the name is anonymous, denoted by an underscore (_).
- a *unique* flag, that describes whether the property has unique values. The values can be either T (true) or F (false). We set U (unknown) in the intermediate representation.
- a *share*, that is a 2-tuple ($share_p$, $share_1$), where $share_p$ is the number of all occurrences of the property and $share_1$ is the number of first occurrences of the property within its parent property, i.e., the number of parent properties in which it occurs.
- an *id* flag, that describes whether the property is an identifier. The values can be either T (true) or F (false). We set U (unknown) in the intermediate representation.
- a set of data *types* of a property. Data type can be either primitive (*String*, *Integer*, ...), or complex (*Array*, *Map*, *Set*).
- a set of *models* of a property that the property is represented in. Model can be one of the following: ($JSON$ = JSON document, XML = XML document, REL = relational, $GRAPH$ = graph, COL = column, KV = key-value).
- a set of recursively defined RSDs of *child* properties.
- an optional *regexp* describing the regular expression of the complex structure. It is used mainly to support the standards of XML format possibly in other models (e.g. the exact order of subelements).
- an optional ref that specifies that the property refers to another property.

Note that we have chosen not to include an optionality flag. It is because more information about the structural consistency in the kind is extracted with the usage of a counter. After inferring the schema, we compare the $share_1$ with the $parent.share_p$. If $share_1 < parent.share_p$, the property is optional, otherwise, when $share_1 = parent.share_p$, the property is required. In addition, we can determine whether this optionality is due to an error in the data, for example, when the property is present in all documents except one (in other words, $share_1 = parent.share_p - 1$). The user then can conclude that there is an unintentional inconsistency in the data.

Moreover, separating the total number of property occurrences ($share_p$) and the total number of first occurrences of a property within its parent property ($share_1$) gives the ability to distinguish the multiplicated properties. A property is multiplicated when it forms an element of an array with multiple elements.

Mapping Process. The fundamental idea behind the map phase can be seen in Mapping Process Algorithm 2 of a single record will be as follows.

1. We traverse all the records from the kind and all their properties.
2. While traversing the records and properties, we map each one into the respective RSD and infer all the features based on the implementation of the wrapper.
3. The record and properties with complex values are processed recursively and then added to the set of children of their parent property.

Algorithm 2: Mapping process

```
1  Algorithm mapRecords()
       Input: w_D – Wrapper for database system (or model) D
2              name_κ – Name of the kind to be mapped
       Output: R – Set of RSDs of the records from the kind κ
3      R := ∅
4      foreach record r in w_D.getRecords(name_κ) do
5          R.add(map(w_D, r))
6      return R
7  Function map()
       Input: w_D – Wrapper for database system (or model) D
8              r – Record or property to be mapped
       Output: rsd – RSDs of the record or property
9      rsd := w_D.inferRawSchema(r)
10     foreach property p in r.getChildren() do
11         rsd.children.add(map(w_D, p))
12     return rsd
```

Example. Mapping of the records from Figure 4.1 can be seen in Figure 4.2 in the form of RSDs for the given kinds. RSDs are depicted as a 9-tuple where the features are in the order as presented in Definition 1 For each model, we have inferred the hierarchy of properties, the names, model and data type together with counters. In the case of the XML model we have extracted the regexp. □

Reduce Phase

Having extracted the schema definitions from every record, we need to merge them to get the schema describing the given kind, i.e., a set of records. That brings us to the reduction phase. We can reduce using the modified kind-equivalence presented in [14] and described in Section 3.2.1. The basic principle of this reduction is to merge the individual properties based on the equality of the keys, i.e., the names of the properties in RDSs.

The reduction is recursive and works as follows:

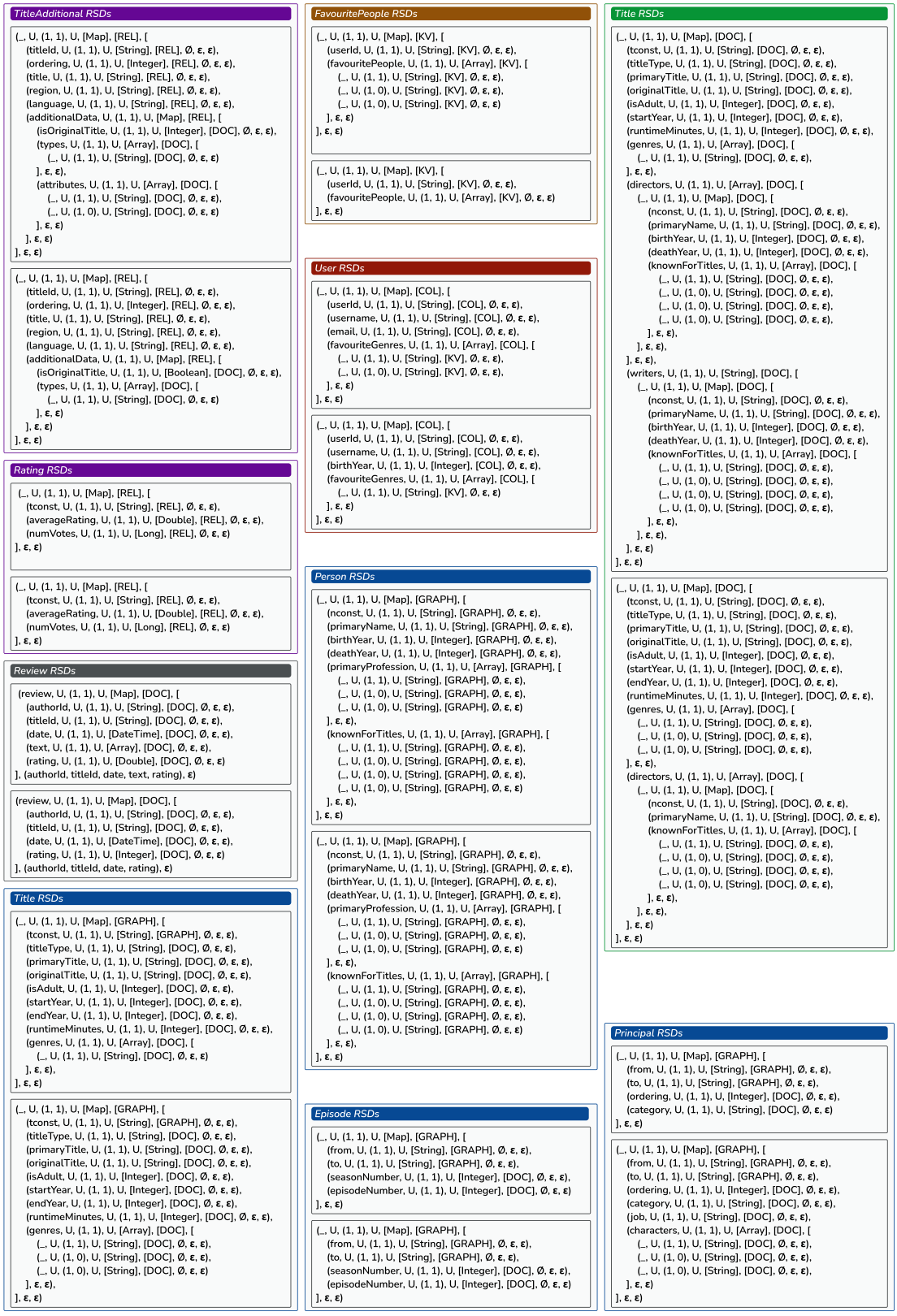

Figure 4.2: Example of RSDs after the map phase

1. We reduce two RSDs with the same name according to Record-Based Local Merge Function (Algorithm 3).

2. Children of RSDs are merged using function *mergeChildren()* in Line 8 of the algorithm:

   (a) Different properties in terms of kind-equivalence are unified and inserted into the set of children of the resulting RSD. All the children are inserted with the flag required set to false.

   (b) The remaining equivalent properties are recursively merged, and the merged results are added to the children of the resulting RSD.

3. The regular expression are merged using function *mergeRegexp()*. This function utilises the existing elaborated approaches (e.g. XTRACT [17] or sk-ANT [25]) that infer regular expressions describing the structure of an XML element.

Algorithm 3: Record-based local merge function

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

Example. The result after the reduce phase can be seen in Figure 4.3. Note that the number of RSDs were reduced by half due to the merging of records in the individual kinds. Analysing the reduced RSDs, we can determine the optionality of the properties. For example the property *birthYear* in User kind has $share_1$ set to 1, while its parent, anonymous root "_" has $share_p$ set to 2. Thus the

property *birthYear* is optional as it is present only in one of the records. We can also derive the multiplied properties, e.g. the anonymous property "_" inside the property *genres* in the Title kind of a JSON document data model. The $share_p$ is set to 8, while $share_1$ is only 4, which means, that at least one record exists, for which there are multiple properties in the same array. □

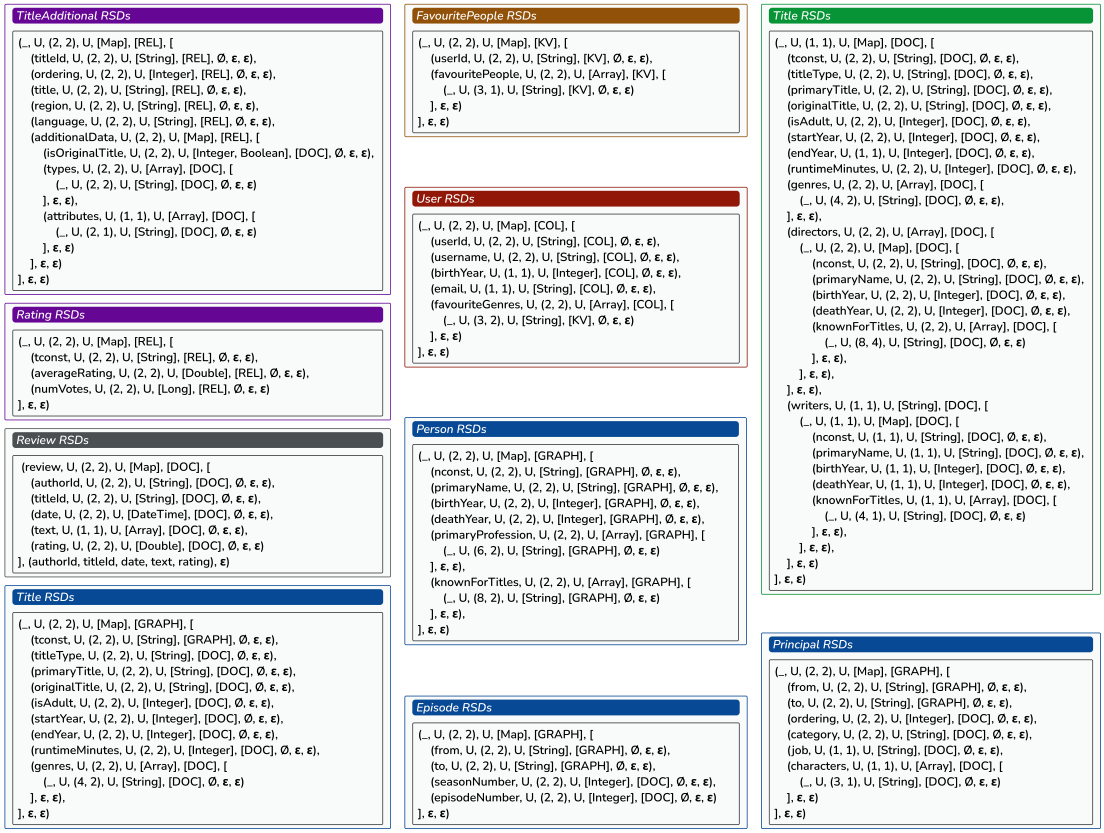

Figure 4.3: Example of RSDs after the reduce phase

Collection Phase

We can now finalise the naïve approach with the collection phase with all the RSDs describing the kinds' schemas in databases. An overall schema will be represented as a forest of generated RSDs. However, we cannot tell whether any RSDs are interconnected, because the focus so far was on inferring the schemas of single kinds.

4.3.2 Analysis of the Naïve Approach

Let us summarise what we have received out of the described approach.

For each kind in every input database, we have generated a single schema description in the form of RSD. From these RSDs, we can inspect the structure of each kind, also with details of the data types of the properties. Furthermore, we inferred the optionality of properties in records. Thus, to some extent, we generated the schema of the multi-model collection of databases.

However, with all of this data, we cannot tell how the involved databases or kinds are interconnected, which is one of the main goals of the proposed approach. In order to provide a more descriptive schema, we need to enhance the approach and consider not only the structure of the kinds but also understand the values of the records and their properties.

4.4 Discovering References

Regarding the discovering of references, one of the existing approaches described in [13] proposed a simple strategy based on the naming conventions of properties. When a property is named $entityName\_id$, it refers to the $entityName$ property if the referenced property exists. While this may apply to the single-model data in a single database, where the naming conventions are respected, it is not a universal solution, and for multi-database and multi-model schema, a more sophisticated technique needs to be applied.

4.4.1 Comparing the Domains

We base our technique on the fact that the domain of the referencing property is a subset of the domain of the referenced property. Hence, we need to perform an analysis of the pairs of properties throughout the entire structure of the records.

We will choose a slightly different strategy of traversing the records than in the naïve approach, where we were processing individual record at a time during map phase. We recursively extract all the properties from each record, together with the hierarchical name of the property. The *hierarchical name* is the name of the database of the record joined with the name of the kind, followed by an absolute path from the root property to the extracted property. In other words, as every record has a tree-like structure, we flatten the tree.

Example. Figure 4.4 depicts naïve approach of generating the domain of values from the properties. Comparing the extracted domains of properties with every other property, we can see that the set of values of the property $rDb.Rating/\_/tconst \subset$ the set of values of the property $dDb.Title/\_/tconst$ and thus the data in kind Rating reference the data in kind Title. □

Having extracted all properties from every record, we now group them by their hierarchical name. As a result, we have explored all the properties and for each property, we have extracted all the values across the records. We now can compare the domains of every pair of properties p_1, p_2 to each other. Let k be a treshold that can be parametrised. If $p_1.domain \subseteq p_2.domain$ and also the $|p_1.domain \cap p_2.domain| >= k$, we denote that property p_1 refers to property p_2.

4.4.2 Comparing the Footprints of the Domains

Although the technique mentioned in the previous section is theoretically acceptable, it does not perform well with a more extensive data set. It is resource-intensive to manage the entire domain at each algorithm step. Therefore, we need to move towards heuristic principles to process large data sets.

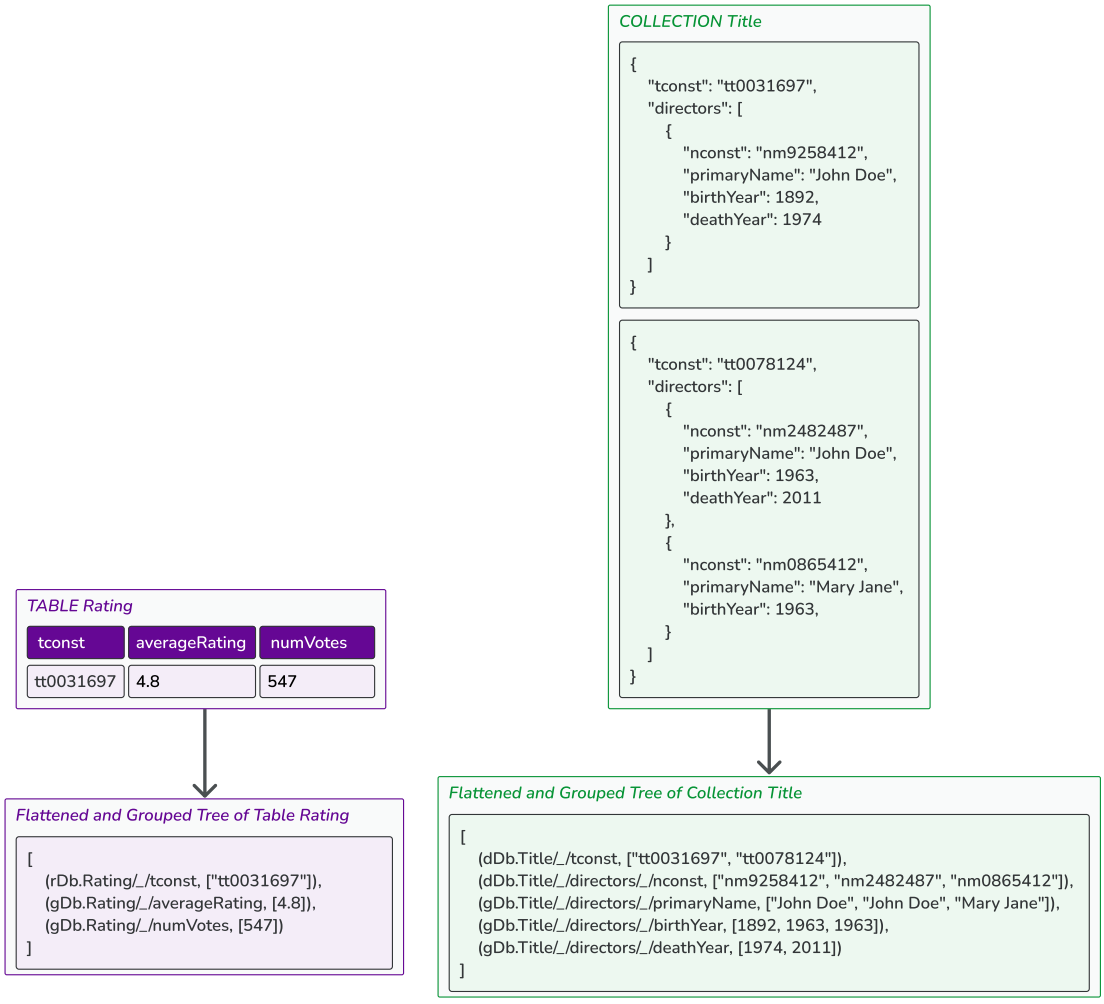

Figure 4.4: Flattened records for the naïve discovery of relationships.

Domain Boundaries

For two sets S_1, S_2 with comparable elements, it holds that if $S_1 \subseteq S_2$, $\implies$ $min(S_1) \geq min(S_2) \wedge max(S_1) \leq max(S_2)$. Using this assumption, we can reduce the properties that need to be compared with each other by comparing the boundaries of the domains.

To apply the comparison of the boundaries, we need to make all values comparable with each other. We will achieve this by comparing the hash of the properties to compare. With this approach we will reduce the time and space complexity when comparing large values.

Bloom Filters

We can extend this technique even further by using a Bloom filter presented in [26]. A Bloom filter is a space-efficient data structure that stores an approximation of the original set. Its principal intention is to decide whether a value is in the set or not. Using this data structure, we do not get any false negatives. However, there is the possibility of false positives.

The implementation of a Bloom filter BF of a set S will be as follows: Let k be the length of a Bloom filter, and let $h : S \to [0..k-1]$ be a hash function. For each value $s \in S$, set $BF[h(s)] := 1$, otherwise, if $s \notin S$, set $BF[h(s)] := 0$.

Note that using a Bloom filter as a footprint of the domain, it is straightforward to determine for two sets whether one may be the subset of the other. Let S_1, S_2 be two sets, $S_1 \subseteq S_2$, with Bloom filters BF_1, BF_2, respectively, both of the same length k. Then $\forall i \in [0..k-1]$, $BF_1[i] \leq BF_2[i]$.

Using this knowledge, we reduce the reference checks of the properties, firstly by the boundaries, then by comparing the Bloom filters. However, while we may have reduced the computation complexity, the space intensity is still too big to be used over large datasets. It is because we still maintain the entire domain together with the Bloom filter. Therefore, instead of maintaining the domain during the grouping phase, we will add the values to the Bloom filter rather than storing them in sets.

4.4.3 Property Domain Footprint

With all the tools prepared, we are now able to define the supporting data structure named *Property Domain Footprint* (PDF). The data structure PDF contains the descriptors of the domain of a single property. Together with the RSD, we will use this data structure to represent raw properties with intermediate results during aggregation.

Definition 2. *Property Domain Footprint* (PDF) is an *8-tuple (unique, required, multiplicated, sequential, min, max, average, bloomFilter)* consisting of:

- a *unique* flag that describes whether the property has unique values. Values can be T (true) or F (false).
- a *required* flag that describes whether the presence of a property is mandatory in the parent property. Values can be T (true) or F (false).
- a *multiplicated* flag that describes whether the property is a forming element of an array with multiple values. Values can be T (true) or F (false).

- a *sequential* flag that describes whether the integer values of the property forms a sequence. Values can be T (true) or F (false).
- a *min* value, that is minimal in the domain of values of the property with respect to the hash values.
- a *max* value, which is maximal in the domain of values of the property with respect to the hash values.
- an *average* value, that is an average in the domain of values of the property with respect to the hash values.
- a *bloomFilter* data structure as described above.

In addition, we store auxiliary features as a *5-tuple(count, total, first, _min, _max)* with the PDF:

- a *count* of the values of properties.
- a *total* value that equals the sum of the values of the properties with respect to the hash values.
- a $first$ count that reflects the total of first occurrences of a property within its parent property.
- $\_min$ and $\_max$ that store the minimal and maximal integer value of the property.

4.4.4 Footprint Miner Algorithm

Let us propose the Footprint Miner Algorithm (see Algorithm 4) that generates the footprints from the set of kinds. We will use the MapReduce pattern once more. The aggregation of the properties in Line 5 of the Footprint Miner Algorithm will be straightforward and can be seen in Footprint Miner Aggregate Function in Algorithm 5. The collection phase is responsible for building a tree out of the pairs of hierarchical name and footprint in order to compute additional features like *required*, *sequential* and *average*. The part that builds the tree is described in function addToForest() in Algorithm 6. The finalise phase computes the features as described in Algorithm 7.

Example. In Figure 4.5, the process of the footprint miner is described. In the first stage, the properties are flattened with inferred features including the auxiliary ones. After the finalise stage, the properties are enriched with remaining properties, and thus the footprints are mined. □

4.4.5 Applying Information to Detect Relationships

Let us now inspect the extracted data to detect relationships between the properties. We start by discovering the identifiers as it is the fundamental part of the reference discovery.

Identifiers

To this moment, we have omitted one fundamental characteristic of a reference relationship. If property p_1 refers to property p_2, then p_2 has to be specified across all the records, and it has to have unique values in all the records.

Note that we can decide whether the value has been probably already added into the set by using a Bloom filter. This can give us approximate results of the

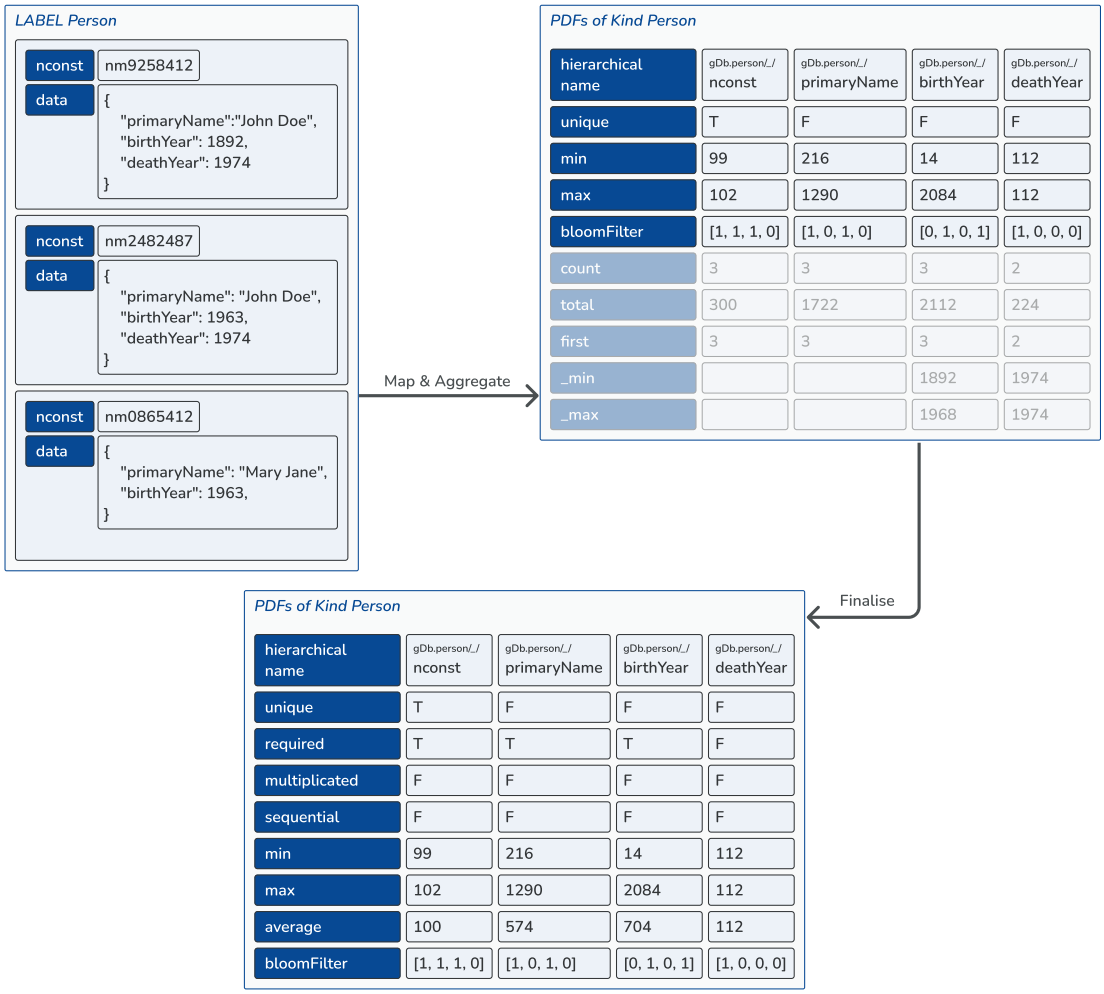

Figure 4.5: Process of the footprint miner

Algorithm 4: Footprint miner algorithm

Input: W_D – Wrapper for database system (or model) D
1 N_D – Set of names of kinds whose footprints are to be mined
2 **foreach** $name_\kappa$ in N_D **do**
`// preparation phase: load and map each property from every record of a given kind to a single pair (hierarchicalName, trivial footprint)`
3 $RDD(name_p, f) :=$ W_D.flatMapRecordToNameFootprintPairs($name_\kappa$)
`// aggregate footprints containing raw values to create a single footprint for each property` p
4 $RDD(name_p, f) := RDD(name_p, f)$.mergeValueDuplicates()
5 $RDD(name_p, F) := RDD(name_p, f)$.groupByKey()
6 $(name_p, f_p) := (name_p, F_p)$.aggregateByHierarchicalName()
`// footprint collection phase`
7 **foreach** $(name_p, p)$ in $RDD(name_p, p)$ **do**
8 addToForest(p, S)
`// footprint finalisation phase - evaluation of features` *required*, *sequential*, `and` *average*
9 finalise(S)

uniqueness of the properties. On the other hand, to determine if the property is required, we do not have any tool that can help us to answer the question.

Let us improve the proposed algorithm yet, mainly the part of flattening the records and grouping the flattened properties. During the flattening phase, we will be storing hierarchical names and values and assigning each flattened property a count preset to 1. After that, we remove duplicates in the values of the properties with the same hierarchical name. With each duplicate removal, we increase the property counter that has a duplicate. Next, in the grouping phase, we sum the counters and add all the unique values into the Bloom filter. If all individual counters of one property had a value of 1, the property is marked as unique.

Thanks to these improvements, we can get more information about the data. We can now undoubtedly tell that a property is unique. When resolving the optionality, we need to consider the aggregation relationship of the property with its parent. If the *count* of the child property is equal to the *count* of the parent property, then the child property is marked as required within the parent property.

As can be seen in the Identifiers Builder Algorithm (see Algorithm 8), if the property is unique and required, it can now be marked as an identifier.

As a side effect of the aforementioned improvements, if we store unique values in the Bloom filter, we can adjust the definition of the Bloom filter. Instead of setting the value of $BF[h(s)]$ to 1, we increase the value by 1 with each insert. When comparing the Bloom filters, we now get more precise results.

Algorithm 5: Footprint miner aggregate function

Input: F – Non-empty list of footprints to be aggregated
Output: r – Aggregated footprint

```
1  r := (T, T, F, ∞, 0, 0, [0, ..., 0])
2  foreach footprint f in F do
3      r.unique := r.unique AND f.unique
       // required and sequential are resolved in finalisation
          phase
4      r.min := min(r.min, f.min)
5      r.max := max(r.max, f.max)
       // auxiliary total and count are computed while average is
          resolved in finalisation phase using the helping values
6      r.total := sum(r.total, f.total)
7      r.count := sum(r.count, f.count)
8      r.bloomFilter := merge(r.bloomFilter, f.bloomFilter)
       // additional helping properties
9      r._min := min(r._min, f._min)
10     r._max := max(r._max, f._max)
11     r.first := sum(r.first, f.first)
12 return r
```

References

With all the data we have collected so far, we can proceed with the lookup of the references between the pairs of properties. We will implement this straightforwardly by comparing the footprints of the domains of each pair of properties as can be seen in Algorithm 9. However, we consider only those pairs where the possibly referenced property is the identifier.

To refine the results even more, we will try to find and distinguish the accidental references which we will mark as *weak*. For this, we introduce the feature of a property called *sequential*. A property is marked as sequential if its integer values form a sequence. To decide whether a property is sequential, we use the following statement: Let *min* be the minimum of values, *max* be the maximum of values, *count* be the count of values. A property is sequential if holds that $(max - min) = count - 1$.

We say that the reference is weak, if both properties of the reference pair are auto-incremented, which means that they are required, unique, sequential and not multiplicated.

Redundancies

It is common in multi-model data to contain relatively the same data in different models. For example, to store the information about users of social networks in a graph-model database to traverse the relationships quickly and in the document-model database to access basic information about the individual users. We call this *data redundancy*.

In addition, we distinguish between full and partial redundancy. The *full*

Algorithm 6: Function addToForest() (forest appender)

```
   Input: name_p – hierarchical name of a property p
1          o_p – object, i.e., a schema description (RSD) or a footprint of
     property p
2          O – schema forest or footprint forest
3  name_κ := name_p.kind()
4  node := O.getOrCreateRoot(name_κ)
5  name := name_p.head()
6  repeat
7      if node.hasChildren(name) then
8          node := node.getChildren(name);
9      else
10         node := node.getOrCreateChildren(name)
11     if name is name_p.tail() then
           // if node represents inner node, its content is merged
           // otherwise (leaf) its placed as is
12         node.placeContent(o_p)
13 until name is name_p.tail()
```

Algorithm 7: Function finalise()

```
   Input: O – footprint forest
1  foreach footprint f in O do
2      f.required := f.first = f.parent.count
3      f.sequential := (max − min) = count − 1
4      f.average := total/count
```

redundancy means that all the values are replicated inside both properties. On the other hand, *partial redundancy* denotes that only a subset of values in one property form the domain of the other property.

As we can compare the active domains of the properties, we will apply this strategy to find redundancies. The algorithm is similar to the reference miner algorithm from the previous section, as can be seen in Redundancies Builder in Algorithm 10.

4.4.6 Optimisation of the Relationship Detection

Using the so far presented approach, we can detect identifiers both with reference and redundancy relationships across different kinds. However, some optimisations should take place as we process all the pairs of properties multiple times, even though we can reduce these comparisons based on the intermediate results.

Note that the property referenced to in the reference pair must be an identifier. Instead of taking a whole set of properties, we can thus use the precomputed set of properties in the previous step and find the properties that refer to them.

Moreover, we can see multiple similarities between the exploration of references and redundancies. Note that for the redundancy to occur, there must be

Algorithm 8: Identifiers builder – naïve algorithm

```
   Input: F – list of (pre)computed footprints
1  C_ident := ∅
2  foreach footprint f in F do
3      if f.unique AND f.required AND not f.multiplicated then
4          C_ident.add(f)
5  return C_ident
```

Algorithm 9: References builder – naïve algorithm

```
   Input: F – list of (pre)computed footprints
1  C_ref := ∅
2  foreach footprint f_1 in F do
3      foreach footprint f_2 in F \ f_1 do
4          if isIdentifier(f_2) ∧ f_1.min ≥ f_2.min ∧ f_1.max ≤ f_2.max then
5              if f_1.bloomFilter ⊆ f_2.bloomFilter then
6                  weak := f_1.isAutoincremented() ∧ f_2.isAutoincremented()
7                  C_ref.add((f_1, f_2, weak))
8  return (C_ref)
```

some reference because one of the domains in the redundancy pair of the property has to be a subset of another. Therefore we can use the references as the basis for the redundancies and search only the neighbourhood of each involved property to find the redundancy pairs. Also, we parameterise the minimum number of involved pairs in a redundancy relationship to eliminate trivial redundancies, such as, for example, the redundancy of a single referencing property to a referenced property, which we consider a reference relationship.

As previously said, we distinguish between the full and partial redundancy. Instead of comparing only the boundaries and the Bloom filters of the domains of the properties, we present a function to resolve whether one footprint forms a subset of another footprint. The algorithm of function $formsSubset()$ can be seen in Algorithm 11. We use this function when comparing the domains of the proposed references or redundancies. Based on the optimisations mentioned above, we propose the algorithm to build the constraints and relationships, as described in Constraints and Relationships Builder in Algorithm 12.

Example. Figure 4.6 shows the process of the mining candidates. Firstly, we generate the PDFs from the records of the individual kinds. Then, we mark the identifiers of the properties and from those, we try to find the references. Finally, the redundancies are discovered based on the previous knowledge. □

Exploring the Neighbourhood of a Property

As mentioned earlier in this section, we need to discover a property's neighbourhood to find redundancy pairs. We benefit from the tree-like structure, and thus we will check the descendants and siblings of the property in the kind tree.

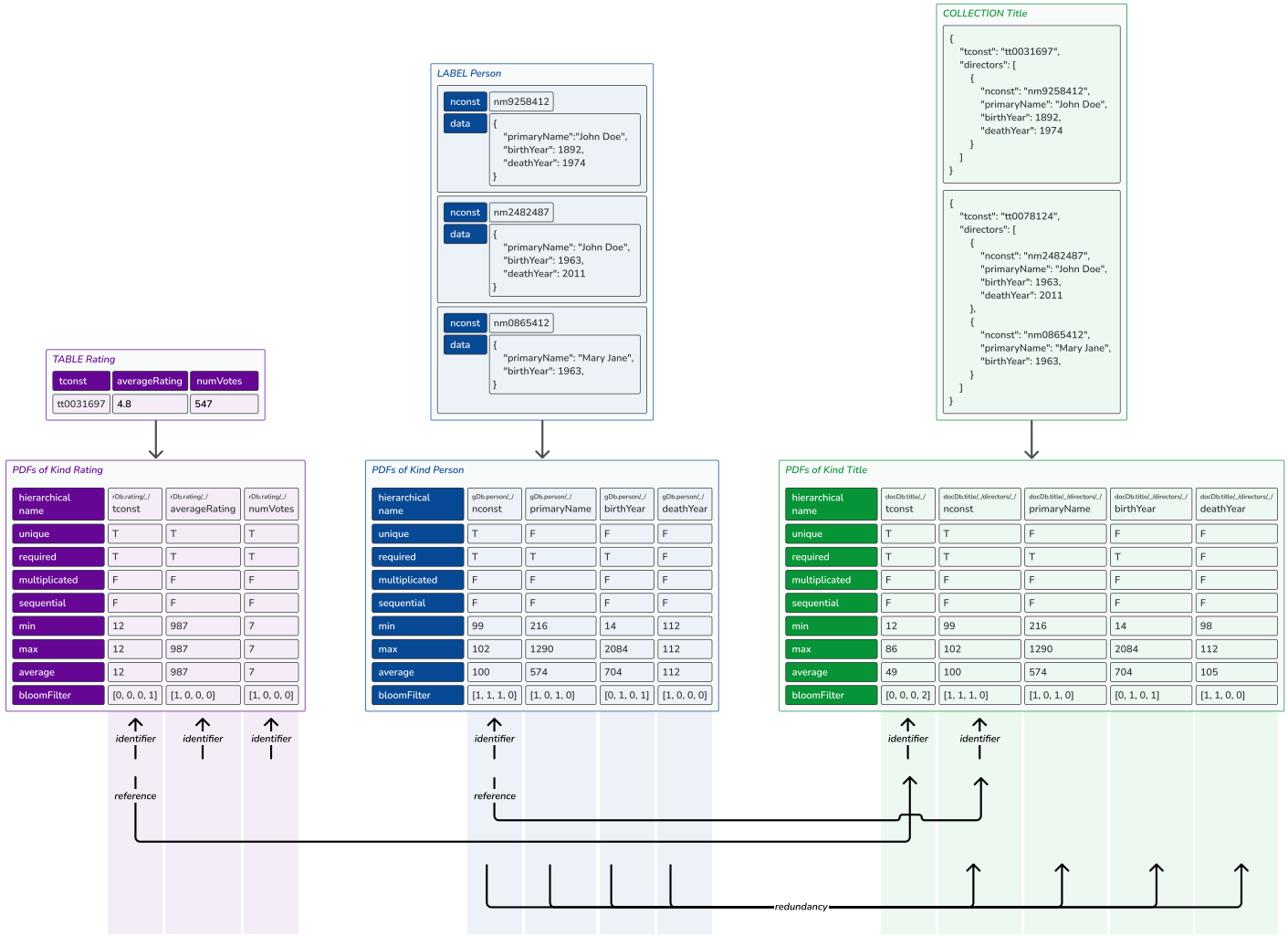

Figure 4.6: Example of the constraints and relationships builder algorithm

Algorithm 10: Redundancies builder – naïve algorithm

```
   Input: F – list of (pre)computed footprints
1  C_red := ∅
2  foreach footprint f_1 in F do
3      foreach footprint f_2 in F \ f_1 do
4          a := f_1
5          b := f_2
6          full := false
7          if f_1.min < f_2.min then
8              a := f_2
9              b := f_1
10         else
11             if f_1.min = f_2.min then
12                 full := true
13         if b.max > a.max ∧ a.bloomFilter ⊆ b.bloomFilter then
14             C_red.add((a, b, false))
15         else
16             if a.max = b.max ∧ full = true ∧
                a.bloomFilter = b.bloomFilter then
17                 C_red.add((a, b, true))
18 return (C_red)
```

For this, we can build the tree like in function *addToForest*() (see Algorithm 6) and then find the desired properties with the help of the graph algorithms. However, a more time-and-space-efficient approach exist if we consider the hierarchical names of the properties, as we can see in Descendant or Sibling Algorithm specified in Algorithm 13. Recall that the hierarchical name has a delimiter "/", so in order to find all the nodes that are lower than the parent of the base property, we take the full hierarchical name except the last part, which denotes the base property.

4.5 Property-based Approach

In the previous section, we have presented an alternative approach to processing the input records in order to generate footprints of the domain. However, we can also adjust this algorithm to generate the schema of the kind.

Instead of generating footprints of the grouped properties, we will generate an RSD of each property as we did in the naïve approach, except for the recursive part of processing children. We benefit from the flattened tree and hierarchical names. After having extracted all the RSDs describing each property, we will build a tree using Algorithm 6, where the input o_p will be the RSD of the property. As a result, we get a schema forest that describes the resulting schema of the kinds, much like the naïve approach.

Algorithm 11: Function formsSubset()

```
   Input: f1 – first footprint
1          f2 – second footprint
   // f1.min == f2.min → "FULL"
   // f1.min > f2.min → "PARTIAL"
   // f1.min < f2.min → "EMPTY"
2  minType := determine(f1.min, f2.min)
   // f1.max == f2.max → "FULL"
   // f1.max < f2.max → "PARTIAL"
   // f1.max > f2.max → "EMPTY"
3  maxType := determine(f1.max, f2.max)
   // f1.average == f2.average → "FULL"
   // f1.average <> f2.average → "EMPTY"
4  avgType := determine(f1.average, f2.average)
   // ∀i : f1.bloomFilter[i] == f2.bloomFilter[i] → "FULL"
   // ∀i : f1.bloomFilter[i] <= f2.bloomFilter[i] → "PARTIAL"
   // ∃i : f1.bloomFilter[i] > f2.bloomFilter[i] → "EMPTY"
5  bfType := determine(f1.bloomFilter, f2.bloomFilter)
6
7  if min(minType,maxType,avgType,bfType) is "FULL" then
8      return "FULL"
9  else if min(minType,maxType,bfType) is "EMPTY" then
10     return "EMPTY"
11 else
12     return "PARTIAL"
```

4.6 Universal Approach

Note that if we want to get the schema description of the databases, both with the identifiers and relationships, we need to run two algorithms over the same data set. However, even this can be simplified with a combination of Property-Based Approach and Discovering References due to their similarities.

For each property, we will generate not only the raw schema, but also the raw footprint of the domain. After the generation, we will group the properties with the same hierarchical name and thus create the RSD describing the property both with the footprint of the domain. Then we will build a schema forest representing the schema of the data set and a footprint forest in order to generate the identifiers and relationships.

4.7 Presentation of Local Schema and Candidates

Before we proceed to the final stage of merging the RSDs by the relationship, we have to recall that all the relationships we have generated are only candidates based on the probabilistic data structure – Bloom filter. Therefore, there is a

chance of false positives and thus we need to let the user decide which relationships (s)he would like to include in the final stage. Using this interaction with the user, we may be able to reduce the false positives even more and generate a more relevant result.

We will talk about this presentation even more in the Chapter 5 dedicated to the proof of concept. We will look at the ways the user can interact with the approaches.

4.8 Merging of Local Schemas using Candidates

After the user resolved all the candidates and confirmed the selection, we can proceed with a so-called *global stage*. In this stage we will produce a schema that will represent the whole input data set.

The fundamental principle will be to connect RSDs according to the relationships. Based on the type of relationship, we define:

- *Reference connection*: We connect the referencing property with the referenced property by populating the field ref in the definition of the RSD with the hierarchical name of the referenced property.
- *Redundancy connection*: The RSDs that represent the properties of redundancy pairs are reduced into a single RSD, and their parents will contain the reduced RSD in the set of children instead of the previous RSD.

Algorithm 12: Constraints and relationships builder algorithm

```
   Input: F – list of (pre)computed footprints
1          k – minimal number of siblings/descendants to form a
    redundancy
2  C_ident := ∅
3  C_ref := ∅
4  C_red := ∅
   // Identifier phase:
5  foreach footprint f in F do
6      if f.unique AND f.required AND not f.multiplicated then
7          C_ident.add(f)
   // Reference phase:
8  foreach ident c in C_ident do
9      foreach footprint f in F \ c do
10         r := formsSubset(f, c)
11         if r is not "EMPTY" then
12             if isAutoincrement(f, c) then
13                 C_ref.add((f,c,"WEAK",r))
14             else
15                 C_ref.add((f,c,"STRONG",r))
   // Redundancy phase:
16 foreach ref (f, c, t, r) in C_ref do
17     D_f := descendantOrSibling(f);
18     D_c := descendantOrSibling(c);
19     R := ∅
20     red := "FULL"
21     foreach d_1 in D_f do
22         foreach d_2 in D_c do
23             type := formsSubset(d_1, d_2)
24             if type is not "EMPTY" then
25                 R.add(d_1,d_2)
26                 red := min(red, type)
27     if R.size() ≥ k then
28         R.add((f,c))
29         red := min(red, r)
30         C_red.add((R,red))
31         t := "WEAK"
32 return (C_ident, C_ref, C_red)
```

Algorithm 13: Descendant or sibling algorithm

```
   Input: P – list of properties
1          p_0 – base property
2  D := ∅
3  s_0 := p_0.hierarchicalName.substring(0, lastIndexOf('/'))
4  foreach property p in P do
5      if p.hierarchicalName.startsWith(s_0) then
6          D.add(p)
7  return D
```

5. Proof of Concept

In order to verify the proposed algorithm and approaches, the proof of concept was implemented in two variants. Let us introduce and describe them.

5.1 Standalone Java Application

The first one we describe is the standalone java application with the architecture that can be seen in Figure 5.1. The main component is the Java application, that processes the user input and based on that selects the required wrapper in order to connect to the databases where the data are stored in. To process the algorithms, the Spark cluster component receives the tasks and execute the jobs. The application is able to process the following algorithms:

- *Record-based inferrer* – corresponds to the naïve approach in Section 4.3.
- *Property-based inferrer* – corresponds to the property-based approach in Section 4.5.
- *Candidate Miner Naïve* – corresponds to the naïve approach of generating candidates in Section 4.4.5.
- *Candidate Miner Optimized* – corresponds to the optimised approach of generating candidates in Section 4.4.5.
- *Universal* – corresponds to the combination of generating candidates and property based approach – the universal approach in Section 4.6.

Also the following wrappers were implemented and can be used:

- *Schemaless MongoDB Wrapper* – generate records from the MongoDB [5] database system.
- *Schemaless PostgreSQL Wrapper* – generate records from the PostgreSQL [3] database system.
- *Schemaless Neo4j Wrapper* – generate records from the Neo4j [24] database system.

5.1.1 Running the Application

The MM-Infer application can be run using JRE 8 with following options:

`algorithm-options:`

- `-algorithm (-a)`

  The identifier of the algorithm to be run. Available arguments are: *record_based_inferrer, property_based_inferrer, candidate_miner_naive, candidate_miner_optimized, universal_inferrer.*

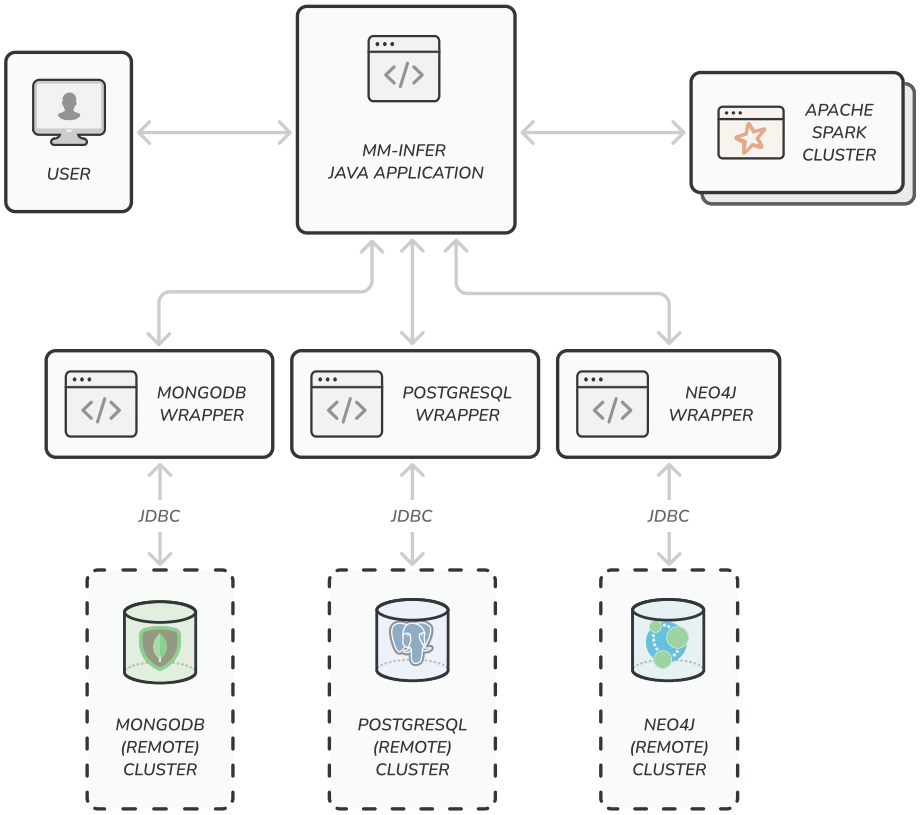

Figure 5.1: High-level architecture of the standalone application

- `-databases (-d)`
  List of [database-options].

`database-options:`

- `-host (-h)`
  The host of the database instance.
- `-port (-p)`
  The port of the database instance.
- `-name (-n)`
  The name of the database in the database instance.
- `-username (-u)`
  The username to connect to the database.
- `-password (-pwd)`
  The password to connect to the database.
- `-kinds (-k)`
  List of kinds in the database to process.

- `-wrapper (-w)`

  Selected wrapper to be used. Available arguments are: *mongo_schema_less, neo4j_schema_less, postgres_schema_less.*

Note that in the case of the record-based approach and property-based approach, only one (first) kind is processed in the algorithm. If the user wants to infer the schema for multiple kinds, the application must run multiple times.

5.1.2 Application Result

The application provides a result encoded in JSON after doing its work based on the selected algorithm. In the case of the record-based approach and property-based approach, the application gives the RSD describing the schema of the selected kind. Both versions of the candidate miner output the 3-tuple (identifiers, references, redundancies). Finally, the universal inferrer combines the results mentioned above.

5.2 Schema Inference Framework

In this solution, we have integrated the aforementioned Java application into a Spring boot [27] application. Together with the persistence storage for user sessions and results, it creates a back-end that is accessible via REST API. On the client-side, the Web application has been developed in Flutter [28] framework.

The core of this application on the server-side is similar as described in the previous section. In addition, we introduced a Web application to allow the user to interact with the MM-Infer application. The architecture of this solution is depicted in Figure 5.2

5.2.1 Common Usage of the Framework

We now provide a description of the usage of this framework by an example. We consider the IMDb multi-model scenario as described in Chapter 2 limited only to the relational, JSON document and graph model.

Firstly, we connect our application to the databases that store the data as can be seen in Figure 5.3. In our case, three databases are used, each for every model. The graph structure is saved in Neo4j database called *db_graph_movies* hosted on localhost, *db_rel_movies* is a PostgreSQL database and stores relational model data and lastly the JSON document model data are stored in a MongoDB database, *db_doc_movies*.

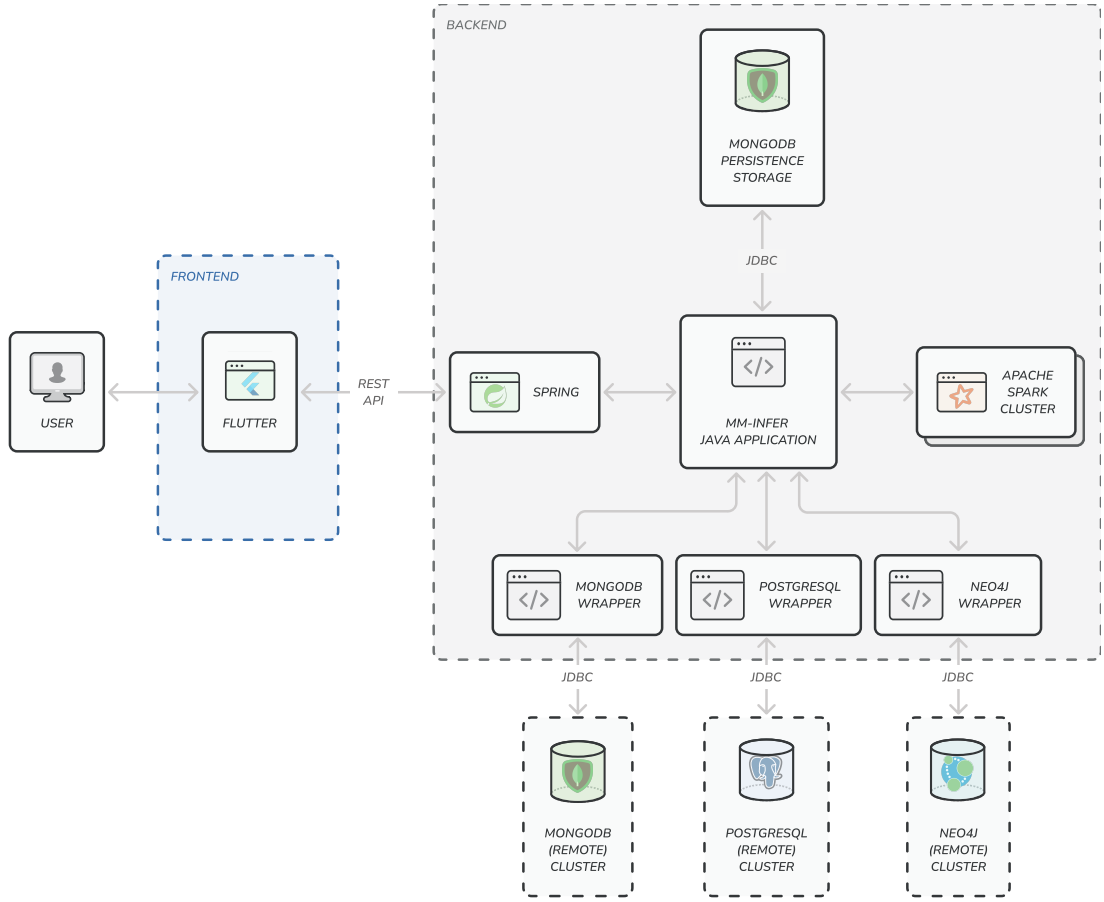

Figure 5.2: High-level architecture of the framework

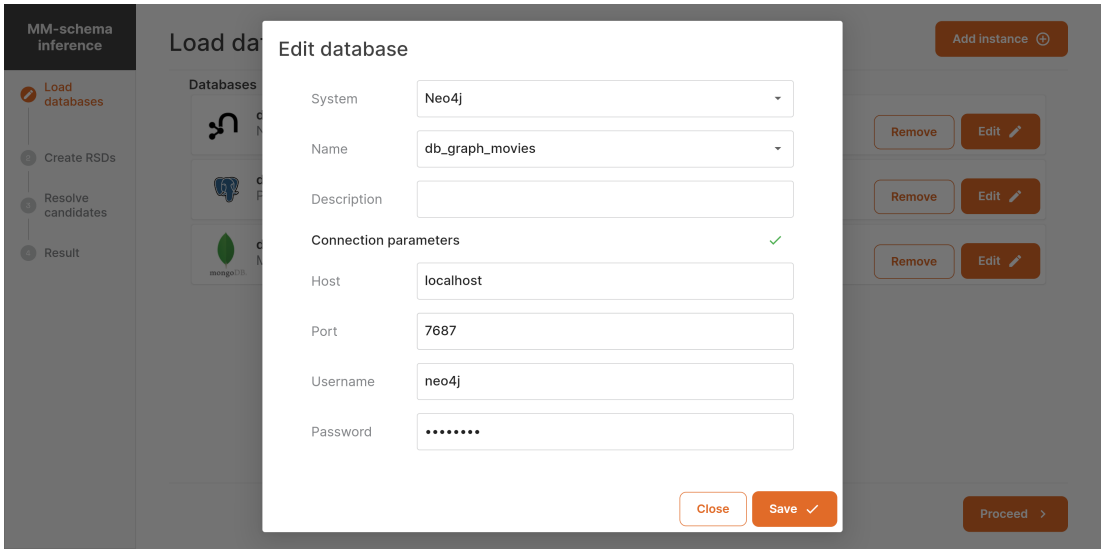

Figure 5.3: Adding a database

After connecting the databases successfully, the application explored the databases and retrieved the kinds, that can be processed. User can now set, as depicted in Figure 5.4, which kinds (s)he would like to include in the follow-

ing phases, as the user may not want to include all of them. The user can also select the strategy to perform, however for the time being, only one strategy is available, the schema-less strategy.

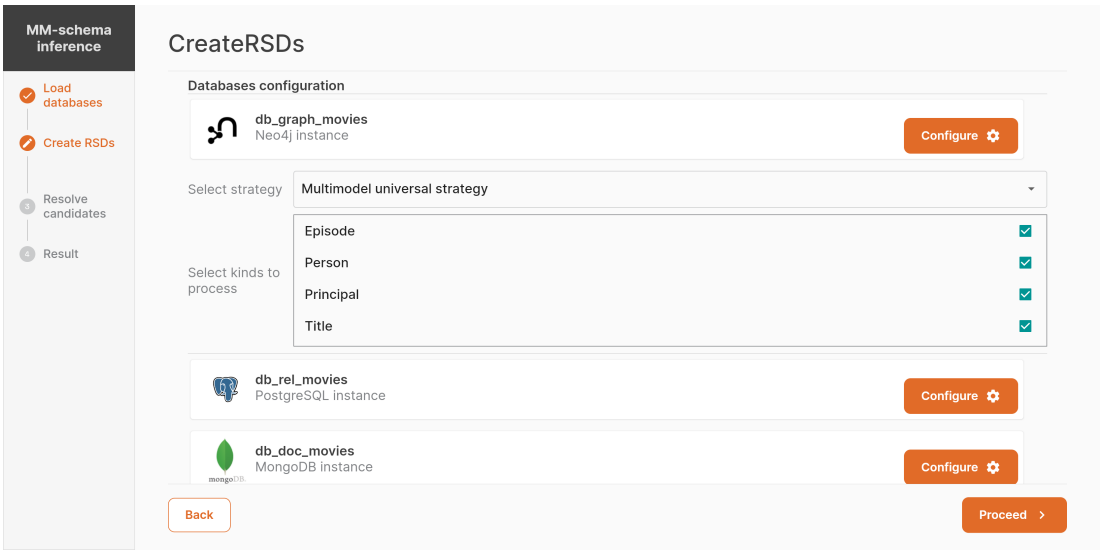

Figure 5.4: Configuring the strategy and the kinds to process

When the user confirms the configuration, the inference process begins and the user can watch its progress as shown in Figure 5.5.

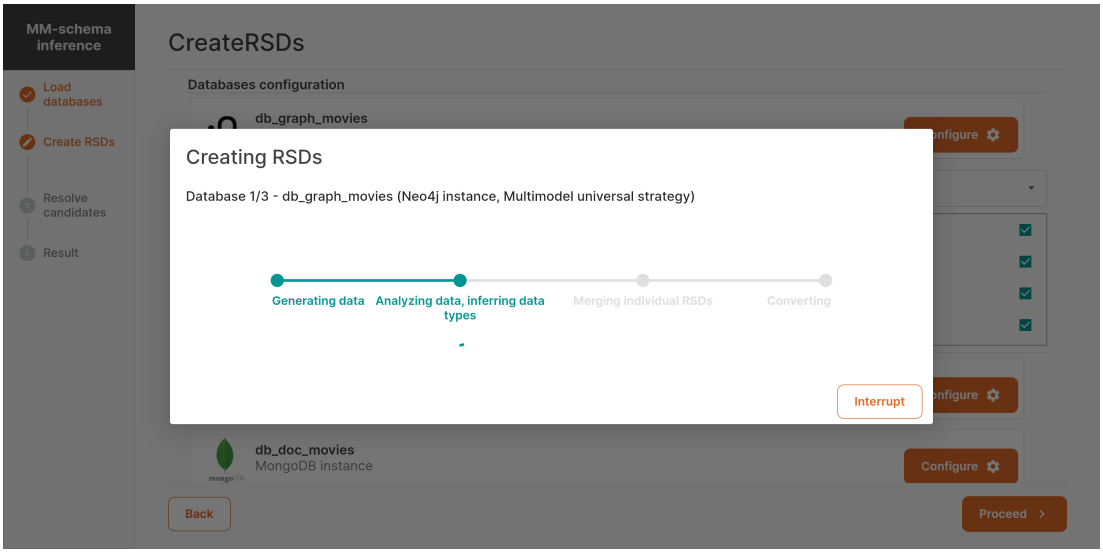

Figure 5.5: Progress of the schema inference

After the schema is inferred and the candidates are extracted, the user is presented with the result as shown in Figure 5.6. Four tabs are available, where the user can resolve the candidates. In the *References* tab (see Figure 5.6), the user can view the inferred references and select, which ones (s)he wants to include in the result. By default, only the strong references are checked and thus included. The user can also specify own references.

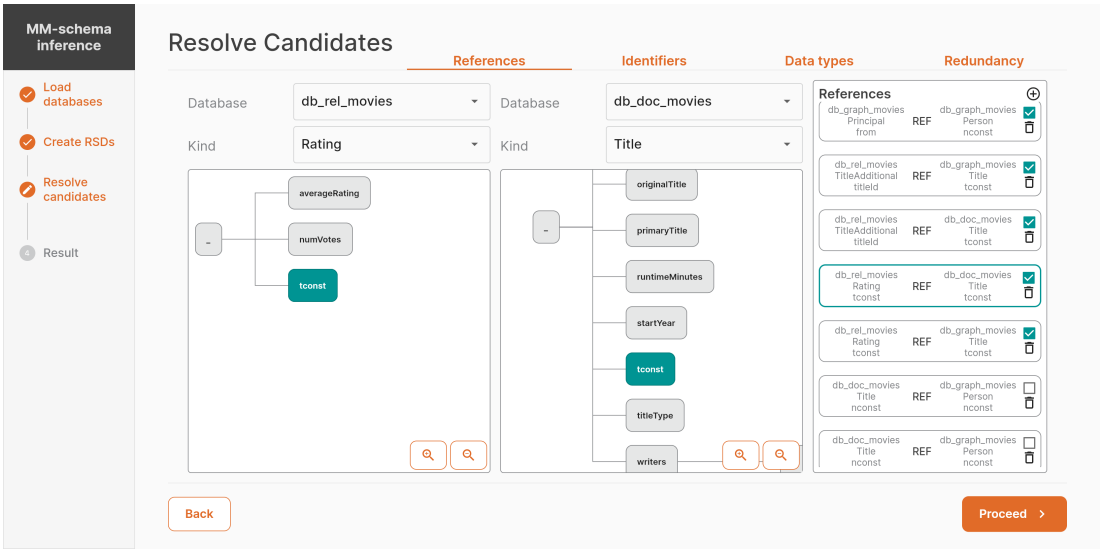

Figure 5.6: Resolving the reference candidates

In the *Identifiers* tab (see Figure 5.7), the user can view the inferred identifiers and also select, which ones (s)he would like to be included in the result. In addition, he can also modify the set of identifiers by selecting the desired options.

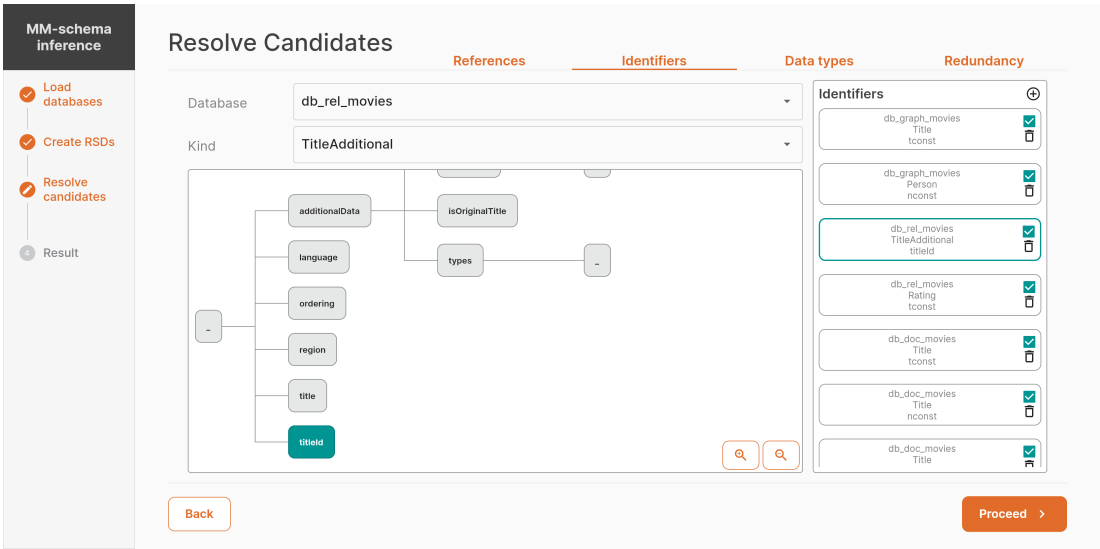

Figure 5.7: Resolving the identifier candidates

Data types tab (see Figure 5.8) shows the inferred data types for each property. By default, the most relevant one is selected, however user can also select another possible data types from the list.

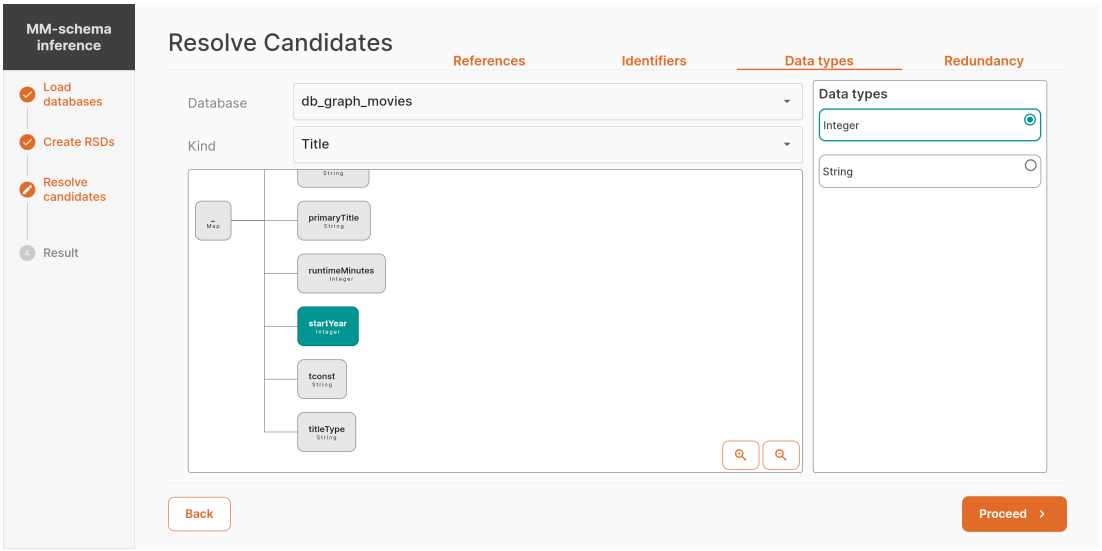

Figure 5.8: Resolving the data types

Lastly, the redundancies are presented in tab *Redundancy* (see Figure 5.9), where the user can view the inferred redundancies and can choose which ones (s)he would like to include in the global result.

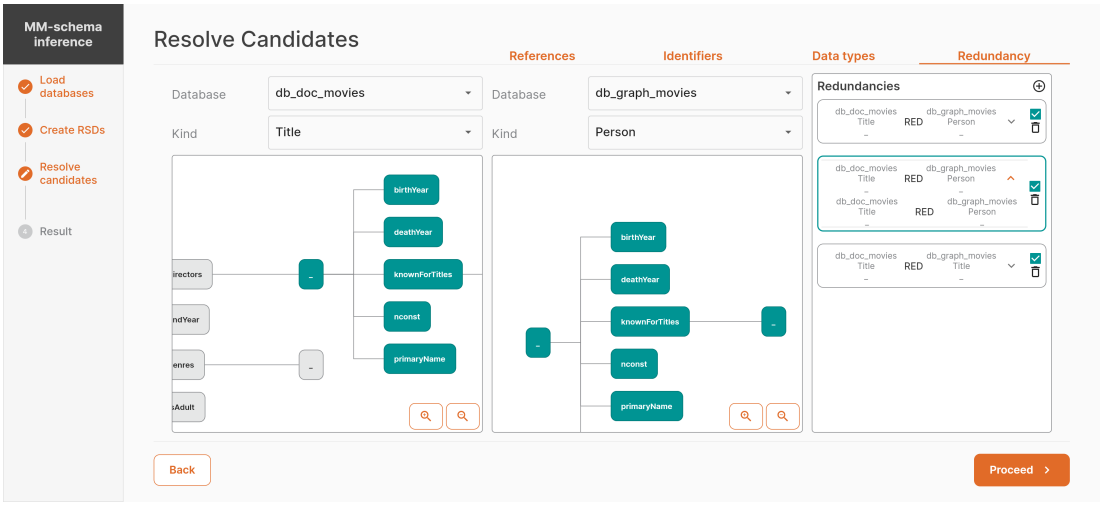

Figure 5.9: Resolving the redundancy candidates

After the user resolves all the candidates, the global phase is performed, where the individual schemas are joined based on the results of local phase and the user input. The result in this scenario can be seen in Figure 5.10. The user can observe the overview of the schema and also for each of the kinds can see its schema. The user can also see, how the kinds are interconnected. To get the raw result, (s)he can also export the schema from the export tab.

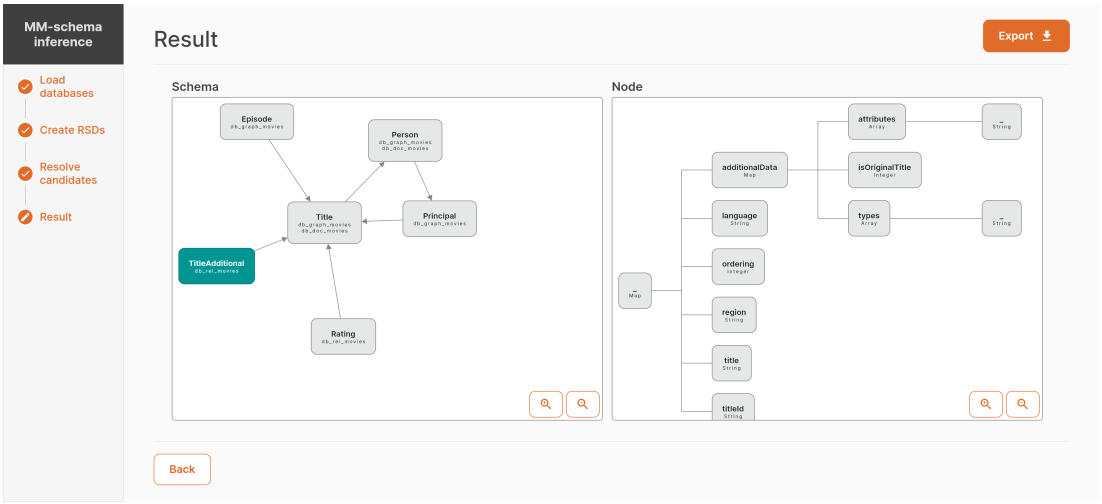

Figure 5.10: Viewing the global result

6. Experimental Results

In this chapter, we will look at the experimental results of the presented approaches. The application was tested in standalone mode as described in Section 5.1 on Ubuntu 18.04.6 LTS server with 62GB of RAM and 8 CPUs. The data used in the experiments were generated by the data generation tool – UniGen [29]. We have generated the data with the following scaling factors: 1, 2, 4, 8, and 16 to test the scalability of the individual approaches. From the generated dataset, the data from the kinds Products, Orders and Invoices were imported into the MongoDB [5] document data store and then the experiments were performed. The sizes of the datasets are shown in Figure 6.1.

Table 6.1: Size of the dataset

Scaling	**Number of documents**				**Size of documents in MB**			
Factor	Orders	Invoices	Products	**Total**	Orders	Invoices	Products	**Total**
1	5000	1156	5000	11156	9,56	4,33	0,99	14,88
2	10000	2312	10000	22312	19,10	8,67	1,99	29,76
4	20000	4624	20000	44624	38,21	17,33	3,98	59,51
8	40000	9248	40000	89248	76,42	34,66	7,96	119,03
16	80000	18496	80000	178496	152,84	69,32	15,91	238,07

6.1 Inferrers

Firstly, we compare the runtime of the approaches that only generate the schema of the data as depicted in Figure 6.1. As the dataset size grows, the difference between the two approaches is more pronounced. In this case, the record-based approach performs better. It is because the documents in the dataset are relatively flat, and with the increasing depth, the potential of the property-based approach's performance may be better exploited.

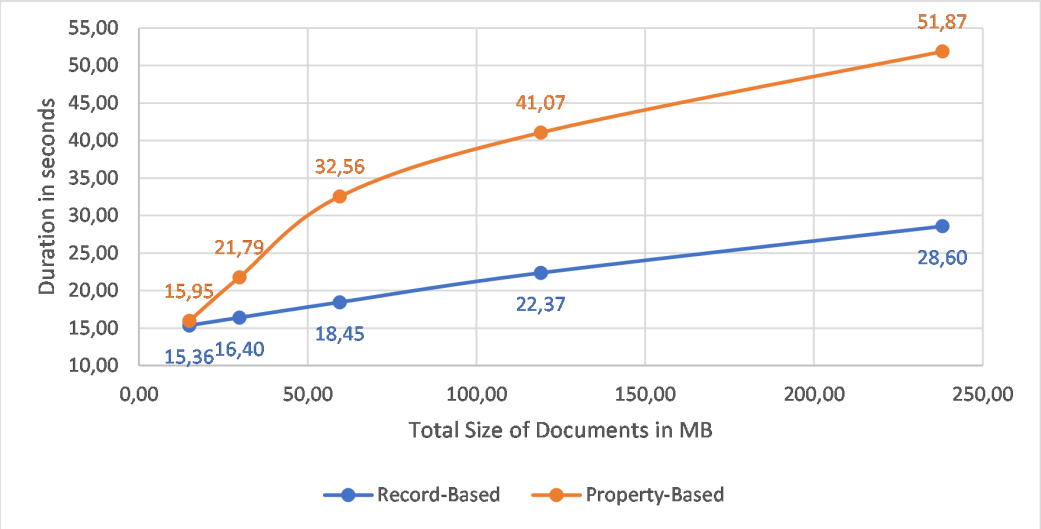

Figure 6.1: Comparison of the inferrer algorithms

6.2 Candidate Miners

In the case of comparison of the mining candidates, both naïve and optimised approaches performed similarly, which is the expected result in the case of the flat structure of the input documents. The results can be seen in Figure 6.2.

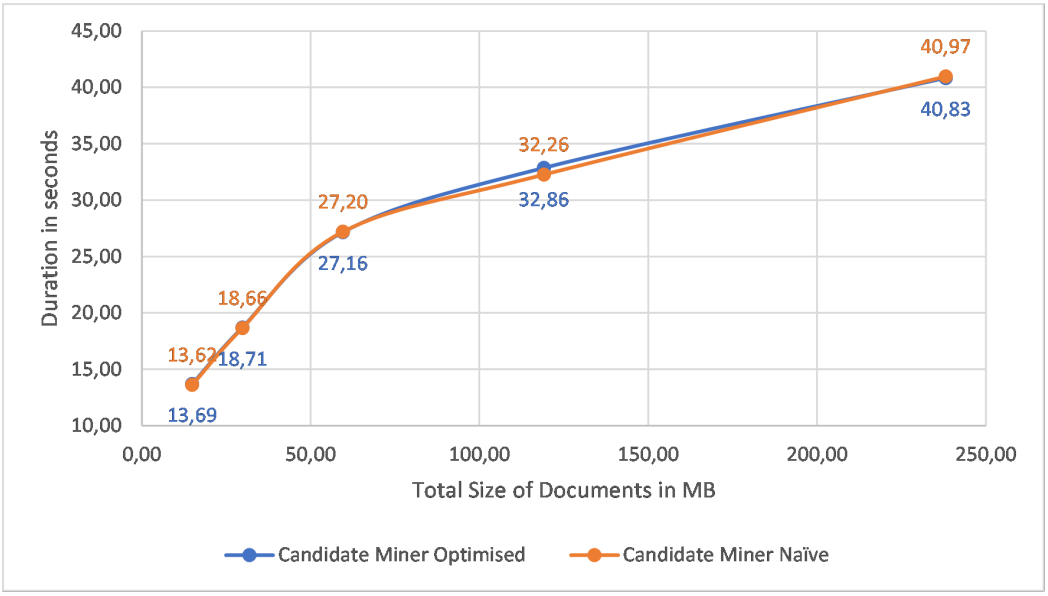

Figure 6.2: Comparison of the candidate miner algorithms

6.3 Universal Approach

Next, we compared the universal approach to the successive runs of inferrers and candidate miners. The universal approach combines schema inference together with exploring the candidates. The universal approach is principally a combination of the property based inferrer and the optimised candidate miner with the possibility of mapping and reducing the input data in a single run. Thus presumably, the universal approach should outperform the combination. We can see the result of this hypothesis in Figure 6.3.

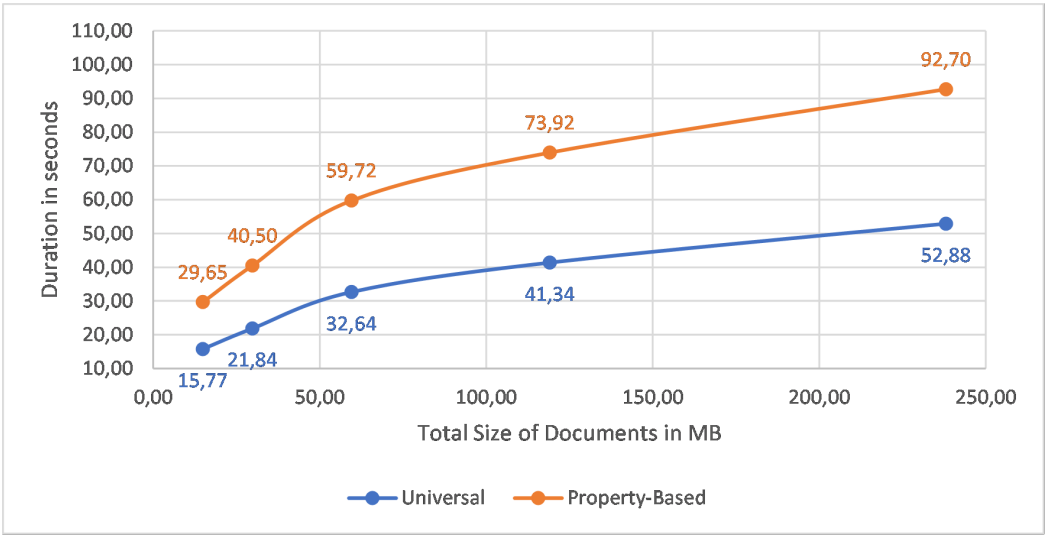

Figure 6.3: Comparison of universal and property-based with candidate miner

Lastly, we compare the universal approach to the combination of the best-performing ones from the previous sections. We can observe from Figure 6.4 that the universal approach outperforms the runtime of the other two approaches in the tested case.

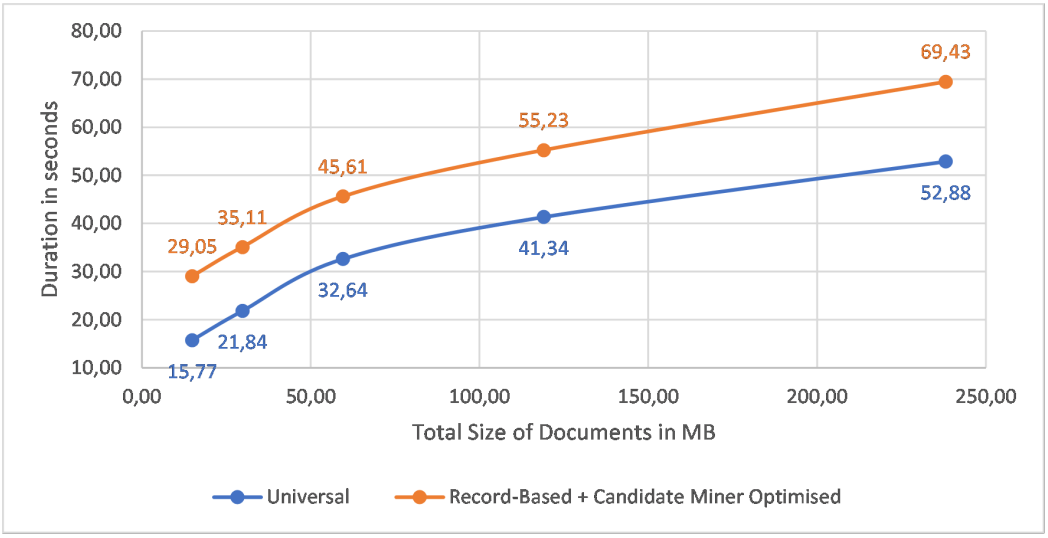

Figure 6.4: Comparison of universal and record-based with candidate miner

7. Conclusion

In this thesis, we presented a novel approach to the process of multi-model schema inference. To the best of our knowledge, our approach is the first one that provides an insight into the structure of a multi-model dataset and the interconnections between the data stored in different models. Apart from the considered approaches for single-model datasets, we presented an approach that is able to process data from various data models in a unified manner.

First, a unification of existing data models was proposed in this thesis. The unification is general enough to cover all the popular data models. In addition, it is also extensible for other data models that were not included in this thesis (e.g. RDF (Resource Descriptive Framework) or array data model). The unification was achieved by building modular wrappers for any datasets, thus being easily replaceable and extensible. Benefiting from the unification, we were able to propose general algorithms that process the data in an agnostic way of the data models behind the data. We also support different variants of database systems with respect to the knowledge of the schema beforehand; i.e. schema-full variant, where the schema is predefined by the user, schema-less variant, where the schema is implicit, and the structure of the data is flexible and schema-mixed, where only a part of the data has to comply with a rigid schema.

Next, we have presented algorithms that utilise the Apache Spark [8] framework and its MapReduce pattern. Benefiting from the selected framework, the algorithms are scalable and can run in parallel; hence they perform well when used over large-scale datasets. We presented two variants of the schema inference algorithm, which process the data either by the individual records or by the properties present in those records. Moreover, we were able to extend the algorithms even further in order to examine the semantics of the data and introduce the integrity constraints to the resulting schema.

Our approach is able to discover the relationships not only within the single dataset but also across multiple datasets, where it is expected that the data are interconnected. Our approach is able to infer the properties that uniquely identify the kinds of the data, i.e. identifiers. Based on the knowledge of the identifiers and the analysis of the data, we were able to discover the candidates of references and redundancies in the data, two principles that are very common in the scenarios where more than a single dataset is involved in the solution. To discover the references and redundancies in the data, we have utilised heuristical techniques and probabilistic data structures. The results of these algorithms are customisable as the user can control which relationships are relevant and should be a part of the global schema of the multi-model dataset.

In contrast with the existing single-model approaches, to the best of our knowledge, we are able to extract more information about the data as we introduced the inference of the integrity constraints and the relationships even for the single-model data. Moreover, in opposition to the JSON schema inference approaches, we are able to infer the order of the child properties with the usage of regular expressions.

The proposed approach was implemented as a proof of concept over the three selected models, namely relational, document and graph). The results of this the-

sis were published in a demonstration paper [30] followed by a journal paper [31], currently under review.

7.1 Future Work

The future work will extend the support for other data models, namely the RDF data model and array data model. We will also propose more advanced techniques to infer the integrity constraints from the datasets. As we consider only simple identifiers and references, we will focus on more complex constraints, e.g. composite identifiers.

In addition, a more experimental evaluation will be performed in order to compare the proposed approaches regarding not only the size of the dataset but also the variability or the large depth of the individual records.

Finally, as the implementation was created as a proof of concept, the optimisation and extension will take place to cover the aspects presented in this thesis fully. We will support more wrappers for the different database systems, enhance the user experience and provide the result schema in different variants (e.g. UML, JSON schema).

Bibliography

[1] Irena Holubová, Martin Svoboda, and Jiaheng Lu. Unified management of multi-model data - (vision paper). In Alberto H. F. Laender, Barbara Pernici, Ee-Peng Lim, and José Palazzo M. de Oliveira, editors, *Conceptual Modeling - 38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings*, volume 11788 of *Lecture Notes in Computer Science*, pages 439–447. Springer, 2019.

[2] Imdb datasets, 2022. [Online; accessed April-2022].

[3] Postgresql: The world's most advanced open source relational database, 2022. [Online; accessed April-2022].

[4] Introducing json, 2022. [Online; accessed April-2022].

[5] Mongodb: The application data platform, 2022. [Online; accessed April-2022].

[6] Postgresql: The world's most advanced open source relational database, 2022. [Online; accessed April-2022].

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[8] Apache spark™ - unified engine for large-scale data analytics, 2022. [Online; accessed April-2022].

[9] Apache spark™ - cluster mode overview, 2022. [Online; accessed April-2022].

[10] Pavel Contos and Martin Svoboda. JSON schema inference approaches. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*, volume 12584 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2020.

[11] Javier Luis Cánovas Izquierdo and Jordi Cabot. Discovering implicit schemas in JSON data. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Web Engineering - 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings*, volume 7977 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2013.

[12] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema extraction and structural outlier detection for json-based nosql data stores. In Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 425–444. GI, 2015.

[13] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring versioned schemas from nosql databases and its applications. In Paul Johannesson, Mong-Li Lee, Stephen W. Liddle, Andreas L. Opdahl, and Oscar Pastor López, editors, *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, volume 9381 of *Lecture Notes in Computer Science*, pages 467–480. Springer, 2015.

[14] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4):497–521, 2019.

[15] Lanjun Wang, Oktie Hassanzadeh, Shuo Zhang, Juwei Shi, Limei Jiao, Jia Zou, and Chen Wang. Schema management for document stores. *Proc. VLDB Endow.*, 8(9):922–933, 2015.

[16] Irena Mlýnková. An analysis of approaches to XML schema inference. In Richard Chbeir, Albert Dipanda, and Kokou Yétongnon, editors, *4th IEEE International Conference on Signal Image Technology and Internet Based Systems, SITIS 2008, Bali, Indonesia, November 30 - December 3, 2008*, pages 16–23. IEEE Computer Society, 2008.

[17] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: A system for extracting document type descriptors from XML documents. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 165–176. ACM, 2000.

[18] Angelo Frozza, Geomar Schreiner, Bruno Machado, and Ronaldo Mello. Rex - nosql redis schema extraction module. *Anais da XV Escola Regional de Banco de Dados (ERBD 2019)*, 2019.

[19] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. An approach for schema extraction of JSON and extended JSON document collections. In *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*, pages 356–363. IEEE, 2018.

[20] Redis: A vibrant, open source database, 2022. [Online; accessed April-2022].

[21] Angelo Augusto Frozza, Eduardo Dias Defreyn, and Ronaldo dos Santos Mello. An approach for schema extraction of nosql columnar databases: the hbase case study. *J. Inf. Data Manag.*, 12(5), 2021.

[22] Apache hbase™, 2022. [Online; accessed April-2022].

[23] Hanâ Lbath, Angela Bonifati, and Russ Harmer. Schema inference for property graphs. In Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 499–504. OpenProceedings.org, 2021.

[24] Neo4j: Graph data platform, 2022. [Online; accessed April-2022].

[25] Chuang-Hue Moh, Ee-Peng Lim, and Wee Keong Ng. Re-engineering structures from web documents. In *Proceedings of the Fifth ACM Conference on Digital Libraries, June 2-7, 2000, San Antonio, TX, USA*, pages 67–76. ACM, 2000.

[26] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[27] Spring boot, 2022. [Online; accessed April-2022].

[28] Flutter - build apps for any screen, 2022. [Online; accessed April-2022].

[29] The data generation tool - unigen, 2022. [Online; accessed April-2022].

[30] Pavel Koupil, Sebastián Hricko, and Irena Holubová. Mm-infer: A tool for inference of multi-model schemas. In Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 2:566–2:569. OpenProceedings.org, 2022.

[31] Pavel Koupil, Sebastián Hricko, and Irena Holubová. Schema inference for multi-model data. Journal paper under review.

List of Figures

List of Tables

A. Attachments

The implementation of the proof of concept (see Chapter 5) is included in the attached package to this thesis. It consists of two individual implementations, *MM_infer_BE* that contains the implementation of the processing side of the whole solution and *MM_infer_FE* that represents a frontend web application of the solution.