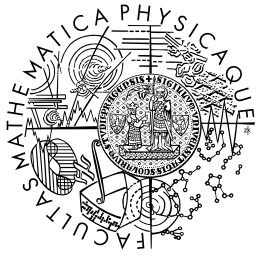


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Ivan Veinhardt Latták

Schema Inference for NoSQL Databases

Katedra softwarového inženýrství

Supervisor of the master thesis: Ing. Pavel Koupil
Study programme: Informatika
Study branch: Softwarové a datové inženýrství

Prague 2021

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

Dedicated to

my supervisor for
always being patient
and lending his expertise,

to the other research teams for
providing free access to their work,

to my wife for her undying love and support,

to my roommate and friend for his words of wisdom,

to our cats for their loyal, although entitled, companionship,

and, most of all, to my mom and grandparents for making me do it.

Title: Schema Inference for NoSQL Databases

Author: Bc. Ivan Veinhardt Latták

Department: Katedra softwarového inženýrství

Supervisor: Ing. Pavel Koupil, Katedra softwarového inženýrství

Abstract: NoSQL databases are becoming increasingly more popular due to their undeniable advantages in the context of storing and processing *big data*, mainly horizontal scalability and the lack of a requirement to define a data schema upfront. In the absence of explicit schema, however, an implicit schema inherent to the stored data still exists and can be *inferred*. Once inferred, a schema is of great value to the stakeholders and database maintainers. Nevertheless, the problem of schema inference is non-trivial and is still the subject of ongoing research. We explore the many aspects of NoSQL schema inference and data modeling, analyze a number of existing schema inference solutions in terms of their inner workings and capabilities, point out their shortcomings, and devise (1) a novel horizontally scalable approach based on the *Apache Spark* platform and (2) a new *NoSQL Schema* metamodel capable of modeling i.a. inter-entity referential relationships and deeply nested JSON constructs. We then experimentally evaluate the newly designed approach along with the preexisting solutions with respect to their functional and performance capabilities.

Keywords: Schema inference, NoSQL databases, Document-based data stores, JSON

Contents

Introduction

The general approach towards efficient data management and storage changed little in the last three decades of the previous century. "The most often used ... were traditional database systems, relational, object, object-relational, XML, or others. Among these the most popular are, without a doubt, relational databases." [1, p. 22] These usually enforce that the data contained conform to a strictly defined schema—a description of the various fields contained, along with what data types the fields are.

In the last two decades, however, many online services and applications gradually started to generate, gather, store, and process data sets with vastly larger *volume* (size), with a vastly higher *variety* (heterogeneity), and coming in at a vastly higher *velocity* (rate of generation or processing). [1] Data having these properties or any subset thereof is commonly dubbed *big data* [1].

Big data's properties make traditional data storage technologies like relational databases unfit for its handling. Thus, big data sparked a need for an alternative at roughly the turn of the century. This need was answered in the form of new database systems—key-value, columnar, document-based, graph, and others. [1, p. 93] All these came to be collectively known as NoSQL, in contrast to the relational databases, majority of which uses Structured Query Language (SQL) as their primary querying language. NoSQL databases often do not enforce a strict schema onto the data contained, example being MongoDB [2], Neo4j [3], or Elasticsearch [4].

Taking a step back, we can see that "data may be present in multiple types and formats — structured, semi-structured, and unstructured." [5, p. 439] Structured data conforms to an explicit schema and includes e.g. data in relational databases. Semi-structured data is not bound by an external schema but carries some schematic information within itself. Its examples include Extensible Markup Language (XML), JSON, and data in columnar databases. Last but not least, unstructured data is not bound by any explicit schema and contains no schematic information, example being a Comma-Separated Values (CSV) file without a header row.

Although semi-structured and unstructured data is not bound by an explicit schema, this is not to say the data is devoid of schema completely. Rather, the schema is implicitly present and can therefore be inferred. A schema inferred from a dataset is of great value—it can be used by stakeholders to reason about the data, or by automated tools for data validation and migration or object code generation. The inference process itself, however, is non-trivial. Several schema inference approaches already exist but many are lacking in various ways.

One important aspect of this topic is how the schema is modeled. A data schema can be modeled on either or both of two layers: (1) Conceptual layer, which deals with conceptual entities and relationships between them as they are in the real world, e.g. entity-relationship (ER) models or Unified Modeling Language (UML). (2) Logical layer, which models data as it exists in the actual system, e.g. UML, relational model schema, XML Schema, or JSON Schema. We are interested in the logical layer since a schema inferred from data is inherently dependent on that data's representation.

Nevertheless, all existing schema-describing metamodels are either not standardized or are insufficient. The only standardized schema description metamodel, UML [6], is unable to handle some concepts of NoSQL data such as union types, etc. [7] This is why we are searching for a schema description format suitable for both semi-structured data (document model) and data from other models.

Our goal is to deeply analyze the problems of modeling of semi-structured data and of inference of its schema, examine a number of existing solutions, understand their strengths and weaknesses, and propose and implement our own solution which would cover their shortcomings. As our area of focus, we choose semi-structured data, particularly *document-based* data, due to the higher complexity of the document model and the overwhelming popularity of document-based databases[1] compared to key-value or columnar ones.

The rest of this thesis is organized as follows: Chapter 1 gives a brief summary of other related work by other researchers in the field. In chapter 2, we conduct a thorough research into the issue at hand and describe a number of existing solutions. In chapter 3, we propose a design of our solution to the problem. In chapter 4, we describe our implementation of the proposed solution. In chapter 5, we present results of an experimental analysis of our solution in comparison to the discussed existing solutions. Finally, in chapter 6, we give an overview of possible future improvements to our solution and state the intended direction of further research.

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

1. Related Work

Research on schema inference of semi-structured data is not a new endeavor as it concerns modern NoSQL databases as well as older technologies such as XML and Resource Description Framework (RDF). There are numerous existing projects related to our work, both in the field of schema inference and distributed data processing. In this chapter, we summarize the most prominent and relevant of these research projects.

Sevilla et al. present an approach for inferring versioned schemas from NoSQL databases based on Model-Driven Engineering (MDE) along with example applications created from such inferred schemas. [8] This research is furthered by Morales in his dissertation thesis. [9] Hernandez et al. tackle the issues of visualization of schemas of aggregate-oriented NoSQL databases and propose desired features visualization tools should support. [10] Most recently, Fernandez et al. expand upon the metamodel from a previous article [8] by introducing a unified metamodel capable of modeling both NoSQL and relational data. [11]

Scherzinger et al. introduce a platform-agnostic NoSQL data evolution management and schema maintenance solution. [12] The same research group later proposes an approach for extraction of schema from JSON data stores, measuring the degree of heterogeneity in the data and detecting structural outliers. [13] Additionally, they introduce an approach for reconstructing the schema evolution history of data lakes. [14]

Baazizi et al. propose a distributed approach for parameterized schema inference of massive JSON datasets and introduce a simple but expressive JSON type language to represent the schema in. [15]

Canovas and Cabot bring an MDE approach for discovering schema of multiple JSON web-based services [16] and later put it in practice as a web-based tool along with a visualization tool. [17]

Frozza et al. introduce a graph-based approach for schema extraction of JSON and Extended JSON document collections [18] and another inference process for columnar NoSQL databases, specifically HBase. [19]

Wang et al. propose a document store schema inference and management method composed of an inference algorithm based on equivalent sub-trees and a new data structure for schema storage and querying. [20]

Moller et al. present jHound, a JSON data profiling tool which can be used to report key characteristics of a dataset, find structural outliers, or detect documents violating best practices of data modeling. [21]

Fruth et al. present Josch, a tool that enables NoSQL database maintainers to more easily extract a schema from JSON data, refactor it, then validate it against the original dataset. [22]

DiScala and Abadi present an algorithm for automatic generation of relational database schema from JSON data along with subsequent transformation of the data itself. [23]

Mlýnková et al. provide an overview of the field of heuristic XML Schema inference and summarize existing approaches and open problems. [24] Similarly, Čontoš and Svoboda bring an article reviewing and comparing existing JSON schema inference approaches and pointing out their shortcomings. [25]

Bex et al. introduce a method of inference of concise XML Document Type Definitions (DTDs) by reducing the problem to one of learning concise regular expressions from positive examples. [26]

Galinucci et al. propose a way to enable non-technical users to enrich RDF data cubes by recognizing recurring patterns in Linked Open Data (LOD). [27]

Bouhamoum et al. delve into the issues of horizontal scaling of existing RDF schema discovery approaches and present a method based on extracting a condensed representation of the initial dataset. [28]

2. Research Work

In this chapter, we present and explain a number of concepts and technologies common to the researched existing inference approaches and define a set of quality measures and properties we consider to be important for a good inference approach. Then, we define a *running example* of JSON documents used to illustrate the functional behavior of researched approaches and explain the rationale behind its structure. Afterwards, we describe in greater detail each of the researched approaches in terms of motivation, technology used, inner operating principles. Finally, we compare the researched approaches w.r.t. the defined quality measures.

2.1 Common concepts

When devising a non-trivial schema inference approach for NoSQL databases, many functional and non-functional requirements taken into consideration stay largely the same from one approach to another. These unchanging requirements lead in turn to some degree of repetition in design choices made during development. So although specifics of existing inference approaches vary considerably, there are also multiple common aspects shared between them which are worth mentioning.

This section enumerates some of these shared aspects. We divide them into four categories:

- *Schema features* – heterogeneity features which can be recorded in the output schema by a given approach.
- *Approach properties* – other external indicators which define what is desirable in a given approach.
- *Principles* – internal characteristics which help achieve the aforementioned desired approach properties.
- *Technologies* – software tools commonly used in existing approaches.

2.1.1 Schema features

NoSQL data schemas carry information about different structural features present in the data. This includes trivial features like entities, their primitive-type properties (boolean, number, string), and structural-type properties (arrays, nested objects). However, even these trivial features have non-trivial aspects to them. For instance, it can prove to be a difficult task to properly name all detected entities and properties. Furthermore, an algorithm must be able to correctly infer all primitive data types present in the target logical data model. For JSON data, these are null, boolean, integer, and string, but this list can be different, if an approach is to be used on data in other data models.

The bread and butter of advanced schema inference approaches is their ability to handle heterogeneity in data. This heterogeneity can come in many forms. Here we list a number of common heterogeneity features.

2.1.1.1 Optional properties

It is often the case in computing that a system must deal with a missing data value. The causes for this are varied—the user opted to not provide it, it is not applicable in the situation, or its retrieval from an external resource, like a database, failed. If the absence of this value is not an error state for the system in question—it is in some way acceptable or tolerated—then we call the value *optional*.

Property optionality should be correctly modeled by an inferred schema, so that the schema consumers can expect the value to be missing when consuming, or choose to omit the value, when producing the data.

2.1.1.2 Entity versions

Given an entity, a set of its required properties, and a disjoint set of its optional properties, one might *incorrectly* assume that the entity is in a correct configuration when *any* subset of the optional properties is present. In reality, however, it may be the case that not every combination of optional properties puts the entity in a valid state.

For example, a point on a plane may be defined either by a pair of Cartesian coordinates (x, y), or by a pair of radial coordinates (r, ϕ). Such point entity could be modeled by having four optional properties, x, y, r, and ϕ. If that were the case, however, this schema would also allow a point entity with a pair of properties (x, r), or with a singular property ϕ, when in reality, such configurations do not actually describe a single point on a plane.

We can see that the entity can only exist in one of two possible configurations (*versions*), and no other combinations of properties are valid. Versioning entities in this manner is an alternative to modeling each property as either required or optional. This alternative brings a trade-off. Using the optional properties approach tends to create more readable schema, while the nuance of entity configuration correctness can be lost. On the other hand, entity versions preserve the inter-property dependencies, but can lead to an exponential explosion in the size of the schema, and in turn vast decrease in readability.

2.1.1.3 Union types

Another subcategory of data heterogeneity, is data type inconsistency. The same property can happen to contain data of different types between its different instances. One of the ways how to handle this inconsistency is to define the property's type as a *union* of the actual types it contains. Care must be taken to ensure that the target schema format of the approach can handle these unions as property types.

An alternative to the union type approach exists in the form of reducing the occurrence types into their most generic type. Here the definition of the word "generic" is left intentionally ambiguous, as the actual meaning is left to each individual algorithm which wants to use this alternative approach. However, this reduction causes us to lose actual type information, so it should only be used when no other options are feasible.

2.1.1.4 Relationships

Inter-entity relationships can be divided into two categories: aggregation relationships (*aggregations* or *aggregates*) and referential relationships (*references*).

Aggregation relationship (aggregate) is a type of oriented relationship between two entities, that expresses containment or ownership between them. There is an aggregation from entity type A to entity type B (we say "A aggregates B" or "B is an aggregate of A") when an instance of B naturally belongs to or is naturally contained by an instance of A. We differentiate between *one-to-one* aggregates, where a single A instance contains or owns a single B instance, *one-to-many* aggregates, where an A instance contains many B instances, and *many-to-many* aggregates, where one A instance contains many B instances, each of which can be contained by many A instances.

In the context of semi-structured data, aggregates can be used as an alternate way of modeling nested JSON objects within documents. Instead of modeling the encountered objects as unnamed entities, they are given a name and bound by an aggregate to its parent object. In this way, one-to-one aggregates can be used to model single nested objects, while objects contained within an array can be modeled by a one-to-many aggregation.

Reference is another type of relationship between two entities. It describes a relation where one entity contains or has access to *something* (identifier, token, tag), which can *uniquely identify* another entity.

In the context of semi-structured data, references describe a state where one object contains as one of its property values a token or tag, which uniquely identifies another object. References are considerably more difficult to infer than aggregations. An algorithm not only has to have a way how to reliably distinguish between a reference and a primitive value without any special meaning, it must then also detect which entity type it is referencing. Both of these partial tasks are non-trivial. Algorithms may commit the mistake of considering a primitive value a reference to an entity, or vice-versa, or even assigning the wrong entity type to the referential relationship. In these scenarios, a direct hint from a human may be necessary for this inference to be corrected.

Reference is another type of relationship between two entities. It expresses a general type of relation where one entity is logically bound to another, without conveying any ownership or containment.

2.1.1.5 Integrity constraints

In addition to schema features describing heterogeneity, integrity constraints are logical rules which must be upheld for the data to "make sense", i.e. be reasonably processable by a user application. In relational databases, for example, integrity constraints are part of the explicitly defined data schema. It follows that in semi-structured data, such integrity constraints still exist, albeit implicitly defined by the existing data. It is then the responsibility of an application to uphold these constraints.

Being part of the implicit schema, it is only natural we want to infer integrity constraints as well. However, inferring them is non-trivial; there are many assumptions to be done on the part of the inference tool, which may or may not be correct. Similarly to what we said before about inferring references, these assump-

tions can be corrected by a hint from a human. In fact, referential relationships manifest as a special type of integrity constraint.

2.1.2 Approach properties

This section summarizes *external* properties of inference algorithms and their benefits. Here, external means to look at an algorithm like a black box, without inspecting its internal components.

2.1.2.1 Input type and format

Approaches differ by what *type* of input data they are able to process, and what *format* that input is expected to be in.

By type we mean, whether an approach is:

- only able to process a single collection of aggregates at a time, consisting of aggregates of only a single entity type; or if it is
- able to process multiple aggregate collections, each containing entities of a single type, all of which are contained in the same logical data domain, and correctly model relationships between these entities.

By format we mean, what physical data formats can an approach handle. Some approaches can only work with JSON data, others are able to handle JSON data with custom extensions, while others still may support other aggregate-oriented data formats as well.

2.1.2.2 Output format

The inferred schema can be outputted in a host of different formats. One of the viable output forms is a data model based on, e.g., the Eclipse Modeling Framework (EMF) Ecore metamodel or its derivative. Users of approaches utilizing this type of output can leverage a number of existing data validation tools and code generation facilities. Another used output format is JSON Schema, which is further described in section 2.1.4.4. Other approaches introduce their own schema description language.

2.1.2.3 Scalability

As was already mentioned, data size and diversity are the main motivations behind the inception of modern NoSQL databases. As such, the requirement of being able to handle a very large amount of very diverse input data and to reasonably scale upwards with growing input size is important in an inference approach.

2.1.2.4 Incremental schema extensibility

Incremental schema extensibility is a property measuring ease with which a schema can be further incrementally extended, possibly multiple times, after its initial creation. The usage of such property is various. A user may decide due to resource scarcity to infer partial schemas from chunks of the data set and merge

them together later. Existing experimental data set may be expanded by another experiment and a user may want to extend the schema inferred from the partial data with the rest, etc.

2.1.2.5 Multi-model context

Document or relational databases are examples of single-model solutions. Here *model* refers to the logical data model, which the database system provides to its users, like JSON documents, or database tables, respectively.

On the other hand, there exist database solutions able to store data in multiple different logical models, example being PostgreSQL, version 9.2 and greater [29], which traditionally provides the relational data model, but also the JSON document data model, using the `json` and `jsonb` data types.

Moreover, applications may choose to store their data in a multitude of database solutions, each with a different model, based on their needs. For example, a social network-like application with a shopping feature may choose to store users and their relationships in a graph database like Neo4j, product information in a document database like MongoDB, and shopping cart information in a relational database.

In this multi-model context, models may even overlap, trading off the cost of data duplication for the benefit of added availability. There can also be present referential relationships between entities in different models and other integrity constraints spanning multiple models. This further increases inference difficulty. Choosing an appropriate schema description language which can describe schema features present in all of the involved models can also be a challenge in itself.

Taking the best practices from existing single-model inference solutions and applying them all in a unified multi-model solution may look like the best approach. However, it is for the aforementioned reasons that the non-trivial task of merging these single-model schemas together still stands.

2.1.3 Operational principles

This section describes *internal* design patterns and paradigms which have proven repeatedly useful in solving problems described in previous sections.

2.1.3.1 Graph representation

Some existing approaches have decided to utilize techniques from graph theory which allows them to infer more precise information about the nature of the input data set. Graphs, and specifically trees, are useful for modeling JSON data and other aggregate-oriented data, because of their conceptual similarity.

The experience of existing approaches shows that tree-like graph representation of aggregates and their properties is a powerful tool in inferring not only the schema of an aggregate collection, but also other related metrics such as the frequency of occurrence of properties and the measure of heterogeneity of documents in a collection.

2.1.3.2 Model-Driven Engineering

Model-Driven Engineering (MDE) is an umbrella term for software and data engineering techniques related to creating and manipulating domain models. In the context of our problem, MDE techniques can be used to first create a domain model for given aggregates, then transform that model to infer the schema features described in section 2.1.1.

2.1.3.3 Performance improvement techniques

Some reviewed algorithms contain recurring techniques or procedures aimed at improving the performance.

One such technique reduces the input size by first transforming each input aggregate into its *raw schema* by replacing all primitive values by their primitive type names, then removing duplicates among these raw schemas. In collections with a low degree of heterogeneity (i.e. high number of structurally identical aggregates), this can considerably improve runtime performance. Moreover, this step can be usually performed in a distributed manner, e.g. using a MapReduce operation.

2.1.4 Technologies

2.1.4.1 JavaScript Object Notation

JavaScript Object Notation (JSON) [30, 31] is an open standard data interchange and storage format based on the JavaScript programming language. It was designed as a human-readable, semi-structured data format for stateless browser-server communication.

Here we define the JSON data model.

Definition 1 (JSON Value)**.** *A JSON Value is:*

- *a Null value –* `null`*,*
- *a Boolean value – either* `true`*, or* `false`*,*
- *a Number value – with format similar to C or Java number literals, e.g.* `5`*,* `3.14`*,* `-2.76e14`*, etc.,*
- *a String value – zero or more Unicode characters, enclosed in quotes (*`""`*), with backslash escapes, e.g.* `""`*,* `"Hello, world!"`*,* `"\r\n"`*, etc.,*
- *an Array value (sometimes also called* list*) – an opening left bracket (*`[`*), followed by zero or more JSON Values, delimited by commas (*`,`*), followed by a matching right bracket (*`]`*),*
- *an Object value (sometimes also* record*,* map*, or* dictionary*) – an opening left brace (*`{`*), followed by zero or more JSON Properties, delimited by commas (*`,`*), followed by a matching right brace (*`}`*).*

Nothing else is a JSON Value.

Definition 2 (JSON Property)**.** *A JSON Property is composed of a JSON String value (called "property key" or simply "key"), followed by a colon (:), followed by a JSON Value (called "property value" or simply "value").*

Order of properties within an Object and white space outside of String values has no effect on the semantics.

While the syntax standard permits duplicate property keys in an object, a majority of existing libraries and solutions disallows them. Because of this, we always assume unique keys in all objects.

Definition 3 (JSON Document Collection)**.** *Let I be a set of all valid JSON String values. Then a JSON Document Collection is a triple (s, D, ι) where:*

- *s (collection name) is a non-empty string of characters, which can be also represented by a JSON String value.*
- *D (collection proper) is a list of JSON Object values (called "Documents" in the context of the collection).*
- *$\iota : D \to I$ (ID function) is a function mapping documents in the collection to their identifiers (IDs).*

The document identifiers are unique within the collection, i.e.:

$$(\forall d_1 \in D)(\forall d_2 \in D) : d_1 \neq d_2 \to \iota(d_1) \neq \iota(d_2)$$

The ID function implementation depends on the library or data store being used. Some systems store document IDs as a special property (e.g. `_id`) in the document root, others keep documents in a map-like collection structure similar to a JSON Object itself, with document IDs as property keys. Others still use a combination of both or a different method.

2.1.4.2 Extended JSON

Extended JSON [32] is an extension of the JSON data format by MongoDB, which increases coverage of use cases by introducing additional data types. It is fully compatible with JSON, utilizing JSON objects as containers for specially named property pairs, which are given special meaning by the Extended JSON data type specification. Extended JSON is used as the data model of the MongoDB database system.

2.1.4.3 Binary JSON

Binary JSON (BSON) [33] is a binary serialization scheme for JSON and Extended JSON formats. According to the authors, it was created to be *lightweight* (minimizing the overhead space needed for storage of large JSON data sets), *traversable*, and *efficient* (allowing for fast serialization and deserialization).

2.1.4.4 JSON Schema

JSON Schema is the result of an ongoing effort to create an open standard for schematically describing, annotating, validating, and documenting JSON data in a machine- and human-readable way. It can be used to define the expected shape of JSON data, validate existing or incoming data against this definition, and provide documentation for other parties.

It has not yet been standardized under any standards organization. The latest draft as of the time of writing is dated December 2020 [34].

JSON Schemas are themselves JSON documents with specially named properties called *keywords* which specify different aspects of each particular schema. Keywords are divided into three categories:

- Meta keywords (e.g. `$schema` or `$id`) – these are used to describe and annotate the schema itself.
- Annotation keywords (e.g. `title`, `deprecated`, or `readOnly`) – these are used to describe, annotate, or otherwise provide additional information about the target JSON data object or about its individual properties or other structural elements.
- Validation keywords (e.g. `type`, `items`, or `properties`) – these are used to establish a set of criteria against which the target JSON data is validated.

Let's focus more on the last subgroup. A very basic yet most notable validation keyword is `type`. It enables to validate data based on basic JSON types, `null`, `boolean`, `number`, `string`, `array`, and `object`. After restricting a value's type, type-specific keywords can be used to further constrain the value.

Type-specific keywords. The `properties` keyword is type-specific for the object type and can be used to impose constraints on the target object's properties. Each property within the `properties` object constrains the property *with the same name* in the schema's target object, if it exists. The property may or may not be present in the target object, but if it is, it must conform.

The property's presence, however, can be made required by using the `required` keyword. This keyword's value is a list of names of properties, which are required to be present in the target object.

The `items` keyword, which is type-specific for an array, can either be of type object, or array of objects. In the former case, the value imposes constraints, to which *each* of the target array's items must conform. In the latter, each of the array's items must conform to the corresponding item in the keyword's value, and additionally the lengths must match.

Figure 2.1 shows an example JSON Schema illustrating the described keywords. Below the schema, a table is included, which shows different target JSON values, and whether each of them conforms to the given Schema, or not.

2.1.4.5 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) is a framework for building tools and developing applications which work with structured data models. It is based on the Eclipse platform. EMF includes in its core module the Ecore metamodel,

```
{ "type": "object", "properties": {
    "reqNum": { "type": "number" },
    "optArr": { "type": "array",
      "items": { "oneOf": [
        { "type": "string" },
        { "type": "boolean" } ] } } },
  "required": [ "reqNum" ] }
```

Target JSON	Valid
`{ }`	✗
`{ "reqNum": 0 }`	✓
`{ "reqNum": 0, "optArr": true }`	✗
`{ "reqNum": 0, "optArr": [ ] }`	✓
`{ "reqNum": 0, "optArr": [ 0 ] }`	✗
`{ "reqNum": 0, "optArr": [ "str" ] }`	✓
`{ "reqNum": 0, "optArr": [ true ] }`	✓

Figure 2.1: An example JSON Schema, along with some JSON values and their validity according to the Schema.

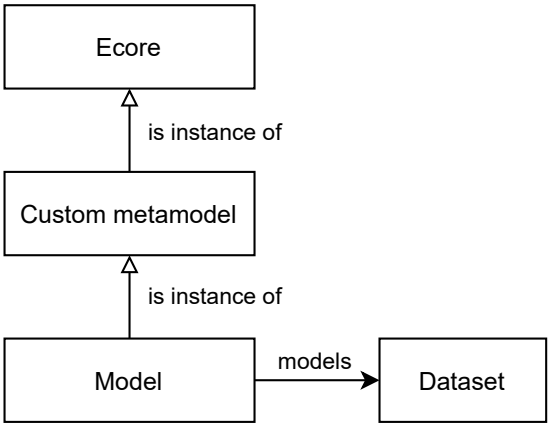


Figure 2.2: Visualization of a common use-case of the Ecore metamodel

which supports i.a. modeling simple data types like integers (as the `EInt` Ecore class) and strings (`EString`), and also classes (`EClass`) and their attributes (`EAttribute`). Ecore is created to be very abstract, it is often too abstract for direct use and is instead used as a *meta*-metamodel for creation of custom metamodels. Instances of these metamodels are then used for actual data modeling. This is visualized in figure 2.2.

2.1.4.6 MapReduce

MapReduce [35] is a programming model for designing highly concurrent, distributed data processing operations in a cluster environment. Its aim is to decouple the task of parallelizing data processing from the implementation itself. It achieves this by allowing the programmer to define a complex operation as a series (a *pipeline*) of atomic steps, which are then picked up by an orchestrator and submitted as tasks to workers.

The most basic MapReduce pipeline consists of three main processing steps – *Map*, *Shuffle*, and *Reduce*. Of these, only the Map and Reduce implementations are provided by the user, while Shuffle is a trivial process.

First the input data is split into small chunks of roughly the same size. Then these chunks are trivially transformed into key-value pairs, where the key is some non-consequential unique identifier, and the value is the chunk itself. Afterwards, a *Map* function is applied to each of these trivial pairs. The function emits a list of zero or more key-value pairs, possibly from a different domain than the function input:

```
function map(pair(k1, v1)): list(pair(k2, v2))
```

Afterwards, the orchestrator *shuffles* all key-value pairs emitted by all map functions. It does this by grouping them by their key. The result of the Shuffle step is a list of zero or more pairs k, V, where $V = (v_1, ..., v_n)$ is the list of all values which were emitted by a map function *with* the key k.

Lastly, a Reduce function is applied to each of the pairs from the Shuffle step. The result is again a possibly empty list of key-value pairs, possibly from another domain than the function input:

```
function reduce(pair(k2, list(v2))): list(pair(k3, v3))
```

The end result of the entire operation is the concatenation of lists obtained from the Reduce functions.

This computational model can be extended by adding more Map steps into the pipeline or introducing new types of operations, like Combine.

2.1.5 Running example

To help provide a baseline for a comparison of existing schema inference approaches, we present a running example of JSON documents in figure 2.3. The running example is used to demonstrate the capabilities of an inference approach to model various schema features, some of which are described in section 2.1.1.

The running example consists of a JSON document collection called `articles`, which could be used to model articles on a news website or a blog. The collection contains two documents with a somewhat similar but different structure.

The properties `_id`, `timestamp`, `author`, `ratings`, and `published` are present in both documents and they have the same type in both of them. Each of the properties is of a different JSON type. The `ratings` array has a different size in each document. The differences between the `author` JSON objects are further described later.

The `comments` and `body` properties are also present in both documents, but they have a different type in each document. The `comments` property is of JSON type array (of strings) in document 1, modeling various comments on an article, and string in document 2, modeling a single comment present. This mirrors the behavior of some existing online services which do not wrap JSON values in an array if the array would only contain one item. `body` is an object in the first document and a string in the second one to model different MIME [36] types of content an article can have.

```
{ "_id": 1, "timestamp": "2021-02-06T16:31:32.029Z",
  "author": {
    "first_name": "John",
    "last_name": "Doe",
    "phone_number": "518-555-0168",
    "location": {
      "latitude": "-48.875000",
      "longitude": "-123.393333" } },
  "ratings": [ 5, 4, 5, 5, 4 ],
  "comments": [ "I like this", ":)" ],
  "attachments": [
    { "url": "/image.png" },
    { "url": "/document.pdf" } ],
  "body": {
    "content": "<p>Article body with HTML tags &amp;amp; entities<p>",
    "mime_type": "text/html" },
  "published": true }
```

```
{ "_id": 2, "body": "Plain text article body",
  "timestamp": "2021-02-10T18:02:29.706Z",
  "author": {
    "first_name": "Václav",
    "last_name": "Novák",
    "phone_number": 321654987,
    "location": {
      "address": "Malostranské nám. 25, 118 00 Praha 1" } },
  "ratings": [ 3, 1, 2 ],
  "comments": "Too plain",
  "article_id": 1,
  "published": false }
```

Figure 2.3: Two JSON documents from the `articles` collection

Finally `attachments` and `article_id` are each present in only one of the documents. `attachments` is an array of objects which tests an approach's ability to describe a one-to-many object aggregation, as different approaches handle this situation differently.

The `article_id` property is used to model a reference relation between entities in the model. It is supposed to represent a one-to-one or many-to-one relation to another article in the collection, in this case the only other one. This reference can express e.g. a parent-child relation or a reply-to relation. The name was chosen deliberately to demonstrate the ability of the Sevilla et al. approach to detect and model reference relations, Sevilla et al. being the only approach possessing this ability that we cover.

The `author` object has two dissimilarities between its instances, first being the `phone_number` property, which is either a string or a number. The other one is the `location` property, which is an object both times. However, it itself has different properties each time, once containing a pair of coordinates and the other time a written address. This inconsistency is used to emulate a situation, where a JSON object always contains specific subsets of all possible properties, i.e. the properties' presence depends on one another. In this case, `latitude` always appears alongside `longitude`, but not if `address` is present.

Another version of the running example is presented in figure 2.4, where some property values are replaced by their more expressive Extended JSON variants. Number-type document identifiers are replaced by Extended JSON type `ObjectId`. Timestamps encoded as ISO 8601 [37] strings are replaced by the Date type. Finally, numeric reference in `article_id` is replaced by the `DBRef` [38] structure.

This version with Extended JSON types is provided specifically for the Frozza et al. approach, which is the only algorithm, among the ones we compare, capable of inferring schema from Extended JSON data.

2.1.5.1 Edge-case example

Complementary to the main running example, we define an additional edge-case example JSON document in figure 2.5. This example is constructed to examine, how well different researched approaches handle JSON array edge cases within documents.

The document contains four properties of type array. The first, `empty_array`, is self-explanatory.

The second two properties, `nested_ints` and `nested_objects`, are multi-dimensional arrays of values. In JSON, arrays can be arbitrarily nested. Such nested structures, however, can not be easily modeled using traditional data modeling means, if at all. These properties are aimed to discover, how well MDE-based approaches handle this situation.

The last property, simply called `values`, is an array containing all different JSON value types, including arrays with and without items, and objects with and without properties. This is to inspect, how approaches behave, when met with this extreme level of heterogeneity.

```
{ "_id": ObjectId("000000000000000000000001"),
  "timestamp": new Date("2021-02-06T16:31:32.029Z"),
  "author": {
    "first_name": "John",
    "last_name": "Doe",
    "phone_number": "518-555-0168",
    "location": {
      "latitude": "-48.875000",
      "longitude": "-123.393333" } },
  "ratings": [ 5, 4, 5, 5, 4 ],
  "comments": [ "I like this", ":)" ],
  "attachments": [
    { "url": "/image.png" },
    { "url": "/document.pdf" } ],
  "body": {
    "content": "<p>Article body with HTML tags &amp;amp; entities<p>",
    "mime_type": "text/html" },
  "published": true }
```

```
{ "_id": ObjectId("000000000000000000000002"),
  "body": "Plain text article body",
  "timestamp": new Date("2021-02-10T18:02:29.706Z"),
  "author": {
    "first_name": "Václav",
    "last_name": "Novák",
    "phone_number": 321654987,
    "location": {
      "address": "Malostranské nám. 25, 118 00 Praha 1" } },
  "ratings": [ 3, 1, 2 ],
  "comments": "Too plain",
  "article_id": DBRef("articles",
                      ObjectId("000000000000000000000001")),
  "published": false }
```

Figure 2.4: `articles` collection with Extended JSON types

```
{ "empty_array": [ ],
  "nested_ints": [
    [ 0, 1, 2 ],
    [ 3, 4, 5 ],
    [ 6, 7, 8 ] ],
  "nested_objects": [
    [ { "key": "value" }, { "key": "value" } ],
    [ { "key": "value" }, { "key": "value" } ],
    [ { "key": "value" }, { "key": "value" } ] ],
  "values": [ null, false, 0, "", [ ], [ 0 ], { },
    { "key": "value" } ] }
```

Figure 2.5: Edge-case example JSON document

2.2 Existing approaches

2.2.1 Sevilla et al.

The approach by Sevilla et al. [8] is a parallelizable and scalable MDE-based schema inference approach for NoSQL databases. It possesses a unique feature of inferring multiple versions per each detected entity type.

The approach first reduces the size of the input using a MapReduce operation. This reduction is done without sacrificing any useful information about the data. Afterwards it uses MDE techniques to infer the collective versioned implicit schema of the entire aggregate-oriented NoSQL database (e.g. MongoDB or HBase). The user is then able to use the inferred schema to generate additional useful automated tools. As a proof of concept, the authors provide a generator for a schema-driven document validator.

2.2.1.1 Motivation

The authors' main motivation is to improve the experience and ease of use of large aggregate-oriented NoSQL databases by inferring the implicit underlying schema of the contained data. The authors claim that the inferred schema can be used not only to improve the programmer's or administrator's understanding of the data via a schema visualization tool, but also to help prevent future data inconsistency thanks to automatically generated data validators. The article then proposes examples of such automated tools.

According to the authors, one of the main advantages of their approach as opposed to existing solutions, is the ability to infer versioned schemas. The inferred versioned schema recognizes multiple versions for each type of entity present in the inspected data. These versions can reflect database schema that evolves over time with changing application requirements. Alternatively, multiple entity versions can also reflect subsets of required properties of an entity.

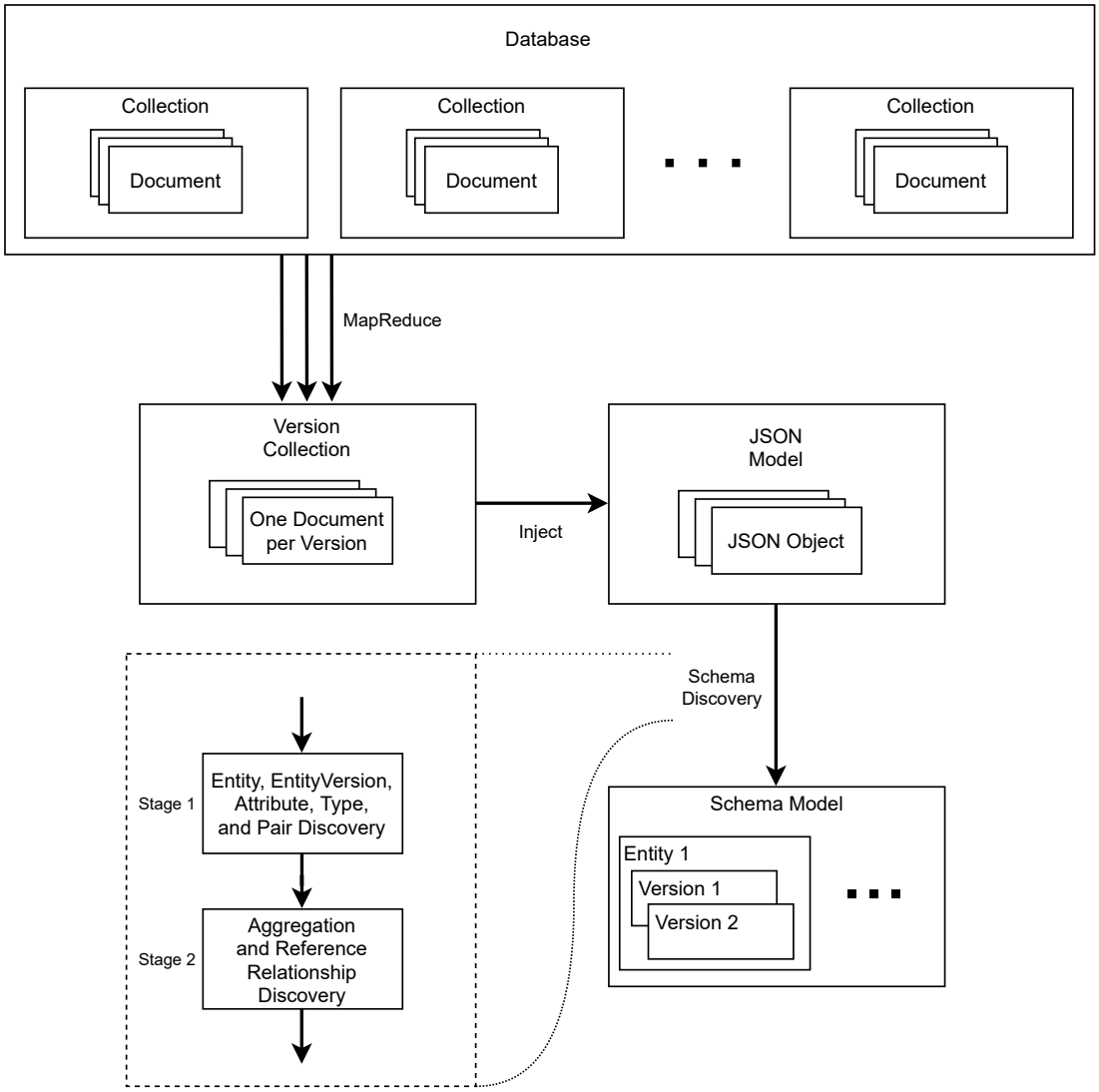


Figure 2.6: Sevilla et al. approach execution high level diagram

2.2.1.2 Requirements

One of the main requirements for the approach is that it can be applied to any NoSQL data store with an aggregate-oriented data model. Currently there are three widely used aggregate-oriented NoSQL data store types, namely document-based, key-value, and columnar.

Having established previously, NoSQL databases are generally used in highly concurrent environments which produce and store large amounts of highly heterogeneous data. Because of this, an important requirement of the approach's design is that it scales well with big data. Furthermore, the efficiency of the inference process plays a large role in the design stage.

2.2.1.3 Technology used

The approach uses the MapReduce computation model to reduce the number of input documents by removing schematically equivalent duplicates. Specifically, a built-in implementation in MongoDB or another database system is used.

MDE techniques are used extensively within the schema inference stage and later also in the stage of automated tool generation.

The authors choose to use the JSON data format and object model as an intermediate representation format within the algorithm to represent any aggregate-oriented data. JSON was selected as the main focus of this approach because of its predominance in the world of document-based NoSQL databases. It is also simple and convenient to convert data from other aggregate-oriented NoSQL data formats into JSON. The authors claim that: "... a piece of semi-structured data can be formalized as a tree whose leaf nodes are atomic values of primitive types (e.g. string, integer, float, or boolean) and the root and intermediate nodes are objects (i.e. tuples) or either arrays of objects or values." [8, p. 469] Such tree structure directly corresponds to the JSON document data model.

2.2.1.4 Detailed algorithm description

First we will start with some definitions.

Definition 4 (Raw schema)**.** *For a JSON value V, the raw schema of V, $S(V)$ is defined as follows:*

- $S(V)$ = *"Number" if V is a number-type primitive value;*
- $S(V)$ = *"String" if V is a string-type primitive value;*
- $S(V)$ = *"Boolean" if V is either "true" or "false";*
- $S(V)$ = *"Null" if V is "null";*
- $S(V) = \{a_1 : S(V_1), a_2 : S(V_2), ..., a_n : S(V_n)\}$ *if* $V = \{a_1 : V_1, a_2 : V_2, ..., a_n : V_n\}$ *is an object, where $a_i : V_i$ are the object's properties, with a_i being the property names and V_i being the property values;*
- $S(V) = [S(V_1), S(V_2), ..., S(V_n)]$ *if* $V = [V_1, V_2, ..., V_n]$ *is an array, where V_i are the array's items.*

Definition 5 (Canonical raw schema)**.** *For a JSON value V, the canonical raw schema of V, $S_c(V)$ is defined as follows:*

- *If V is an object, $S_c(V)$ is created by sorting the properties of $S(V)$ by their names alphabetically;*
- *$S_c(V) = S(V)$ otherwise.*

Definition 6 (Schematic equivalence)**.** *Two JSON values V_1 and V_2 are schematically equivalent if*

$$S_c(V_1) = S_c(V_2).$$

The authors mention, and directly support with their implementation, MongoDB and CouchDB[1], which are document-based databases, and HBase[2], which is a columnar database, but mention that other aggregate-oriented data stores can be used as well.

The approach works on collections of JSON documents. JSON documents are defined as JSON objects with the following additional requirements:

- they have a type name, defined either by a `type` property, or by the name of the containing document collection,
- they have an identifier unique within the collection, defined by an `_id` property, or by other means.

If a user wants to use this approach with a type of aggregate-oriented database other than document-based, the data must be first converted to JSON documents. The authors offer the following as a way how to perform this conversion.

For a key-value store, each key-value pair is converted to a document, where the key of the pair is the identifier of the document. The values must be JSON objects themselves. The type name can either be defined using its property within the object, or by other means, e.g. in Riak KV[3] it can be defined by the name of the bucket containing the key-value pair.

For a columnar store, each table is converted to a document collection with the same name. Then each row of that table is converted to a JSON document within that collection. Lastly, each column-value pair is converted to an equivalent name-value property pair.

The next preliminary step of the inference process comprises the reduction of the input data size. Since NoSQL databases can contain vast amounts of data, a reduction of the input size can be crucial for reducing the overall computational cost of the inference process. This step increases the speed of subsequent steps while leaving the overall database schema information unchanged.

In a database, there will usually be present many schematically equivalent objects, i.e. objects with identical canonical raw schema. Processing multiple schematically equivalent objects would add no information to the schema inference process, but would slow down the process overall. Therefore it is beneficial

[1] `https://couchdb.apache.org/`
[2] `https://hbase.apache.org/`
[3] `https://riak.com/products/riak-kv/`

to eliminate subsets of schematically equivalent objects within a collection, leaving only one representative per schema. Sevilla et al. approach uses MapReduce technology to perform this step.

The MapReduce operation is performed only once on the entire database. The input of the Map step are all the individual documents in the database, along with their type. The Map function is executed once for each of the documents. For a document D of type t, the function first extracts the canonical raw schema of the document $S(D)$. Then it constructs a version identifier by concatenating the type and the canonical raw schema $tS(D)$ and emits a key-value pair $(tS(D), D)$.

The input of the Reduce step are then all the documents in a collection, grouped by their version identifier. The Reduce function is executed once for each unique ⟨type, canonical raw schema⟩ pair in the collection, which would be the first parameter. The second parameter of the Reduce function is a list of all documents with that version identifier. The function arbitrarily chooses and emits one of the documents in the list.

The output of the MapReduce operation is what the authors call a version collection, that is, a set of documents, which is an arbitrary maximal subset of the original database with each document having a different version identifier. More formally:

Definition 7 (Version collection)**.** *For a database $\mathcal{D} = \{D_1, ..., D_n\}$, which is a set of documents D_i, each having a version identifier $id_V(D_i)$: a version collection of $\mathcal{D}$ is any set $\mathcal{S} \subseteq \mathcal{D}$, such that:*

- $(\forall D_1 \in \mathcal{S})(\forall D_2 \in \mathcal{S}) : D_1 \neq D_2 \implies id_V(D_1) \neq id_V(D_2)$ *(version identifier uniqueness)*
- $(\forall D_1 \in \mathcal{D} \setminus \mathcal{S})(\exists D_2 \in \mathcal{S}) : id_V(D_1) = id_V(D_2)$ *(maximality)*

Canonical raw schemas of these remaining documents are later combined to infer the overarching schema. Since only schematically equivalent documents are removed during the extraction of the version collection, no schematic information is lost. Therefore the schema inferred from the version collection is the same as if it was inferred from the entire database.

After the version collection has been extracted from the database, reducing the input size, the inference process itself can start.

The approach uses MDE techniques to extract entity, entity version, property, and association information from the JSON data, and combine them to discover its versioned schema. The JSON documents from the version collection are all injected into the JSON metamodel, creating a JSON model for each.

A series of model transformation is then applied on these models, which gradually transforms and accumulates them into a single collective NoSQL schema model. Detailed information on these transformation steps can be found in the original article.

When all JSON document models are processed, the schema inference process is finished. The inferred NoSQL schema model can then be serialized into textual representation. The authors decided to use their own proprietary textual report language for this serialization. The language is shorter, more concise, and more human-readable than JSON Schema but still expressive enough to describe all the data schema features that the Sevilla et al. approach can infer.

2.2.1.5 Capabilities

When it comes to what schema features the algorithm is able to infer, Sevilla et al. is unique in a number of aspects. It utilizes a unique approach for handling data heterogeneity and evolution over time in the form of entity versions. In their NoSQL schema model, each entity has one or more different versions, with each version having a different set of properties.

Versions of the same entity can represent a state where different database objects of the same type each have a different subset of the entity's properties defined, e.g. because the values of the remaining properties are unknown.

To illustrate, in the running example, either `location` object has defined a different subset of all possible properties. One is missing a string-type property named `address` and the other is missing two string-type properties named `latitude` and `longitude`. In the resulting inferred schema, these two objects would be instances of two different `Location` versions.

A significant feature unique to this approach is the ability to correctly detect and model references to other entities present in the data set. In the running example, the `Article` entity has a property `article_id` which is a JSON number. The algorithm can correctly infer the significance of this value as an identifier of an `Article` entity if:

- the property name is of the form ⟨*entity_name*⟩`_id` or ⟨*entity_name*⟩`_ids` and
- the property value is either
  - a primitive value that is the identifier of a document of type ⟨*entity_name*⟩, or
  - an object which is an Extended JSON `DBRef` referencing a document of type ⟨*entity_name*⟩.

The approach is capable of inferring a collective schema for an entire database, possibly spanning the entire data model of an application. This is in contrast with some other existing approaches which are only capable of inferring the schema of a single document collection at a time, leaving the task of merging such partial schemas to be done afterwards.

The scalability of the approach given by removing schematically equivalent documents from the input before further processing is also worth mentioning as an advantage. However this property is also present in several other compared approaches.

2.2.1.6 Running example

The output of this approach, included as attachment A.1, is a NoSQLSchema model serialized in XML format. The NoSQLSchema root element contains entities as sub-elements. For the running example, these are `Articles`, `Author`, `Attachment`, `Body`, and `Location`. These entity names were inferred from the collection name and from the property names, respectively.

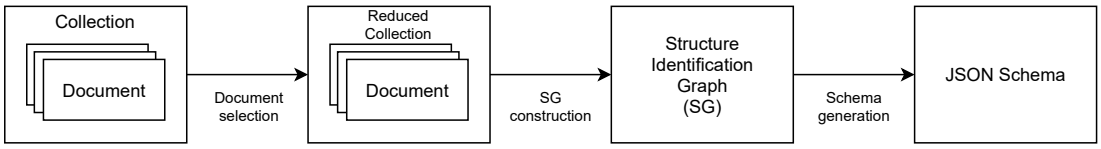


Figure 2.7: Klettke et al. approach execution high level diagram

Each entity has one or more variations[4], each variation having a distinct set of properties and their types. For instance, `Location/1` has a single property of type String called `address`, while `Location/2` has two String properties called `latitude` and `longitude`.

Properties can have three different types: `Attribute`, `Aggregate`, and `Reference`. `Attribute` type properties contain either a single or, in the case of a JSON array, multiple `PrimitiveType` values wrapped in a `PList`.

`Aggregate` type properties contain a reference to a specific entity variation contained in the model. E.g., property `Author/2/location` aggregates variation `Location/2`. Additionally, aggregation multiplicity is modeled by lower and upper bounds.

`Reference` type properties contain a reference to an entity in the model. E.g., property `Articles/1/article_id` references entity `Articles`. Same as with aggregation, lower and upper bounds define the relation multiplicity.

Each property specifies, whether it is optional or not. If a property is contained and has the same type in all variations of an entity, it is required. Otherwise it is optional and has an attribute `optional="true"`.

Any data heterogeneity is handled using the entity versioning system. Both property names and types are taken into account when comparing two entity versions for equality. Alongside the versioning system, optionality of properties is also recorded, as described before. The different sizes of `ratings` array instances are not modeled.

2.2.2 Klettke et al.

The Klettke et al. approach [13] is a schema inference approach for NoSQL data stores based on previous work with XML data [39]. It is designed primarily for work with data in JSON format and outputs the inferred schema as a JSON Schema.

2.2.2.1 Motivation

The approach takes its motivation from work with NoSQL data stores. The authors claim it is crucial to have a reliable understanding of the schema during both application development and analysis of scientific data sets to be able to perform the task. The authors particularly mention the importance of awareness of structural outliers in the data. Unfortunately, the schematic and outlier information is often unknown or incomplete due to rapid application development or mixing of data from multiple experiments within one database.

[4] *variation* describes the same concept as *version*. We use *variation* here because this term is used in the model.

As a result, there is a need for a schema extraction approach that would detect the schema of such heterogeneous data sets. The schema needs to be extracted in a way that would allow us to detect the level of homogeneity of the inspected data, as well as detect outliers in the data structure.

While the approach mainly focuses on data stored in the JSON format in document-based NoSQL data stores, the authors note that it can be extended to work with other aggregate-oriented data stores. The adaptation process is identical to the one used by Sevilla et al., described in section 2.2.1.4.

2.2.2.2 Detailed algorithm description

The schema inference process is divided into several steps. First is the *document selection step*, where the collection of inspected documents is narrowed down. This step is optional. Second step is the *construction of a* Structure Identification Graph (SG) from the selected documents. Third step is the *generation of JSON Schema* from the SG. Additional steps can be then applied to the SG to determine other valuable information on the data set, like structural outliers and the degree of homogeneity.

Document selection step. This is a preliminary manual step during which the user can decide whether to detect the schema of an entire document collection, or to divide the documents into subsets according to some criterion, usually a property value within the JSON document, and detect the schema of the subsets separately.

This grouping property can describe e.g. entities of different versions or data related to different scientific experiments. In the former case, the different output schemas could be used to better illustrate evolution of data in the application. In the latter, this step helps reduce the heterogeneity of the inferred schema.

SG construction step. The SG is a tree with valued nodes and edges. While we summarize its properties, it is defined formally in the related article.

Its vertices reflect JSON nodes present within the inspected documents. Non-leaf vertices represent JSON objects and arrays, while leaves represent nodes of other types.

Let v be a vertex for a JSON node j occurring across one or more documents. v carries the following additional data:

- `NodeList`(v) – a list of NodeID(j) for each occurrence of j across the documents. NodeID(j) is a pair $(docID, i)$, $docID$ is the ID of the document containing j, and i is the unique identifier of j within that document
- Type(v) – information about the data type of j, if v is a leaf.

The graph's edges reflect occurrences of parent-child pairs among nodes in the data set. Given vertices v_1 and v_2 which reflect a JSON object j_1 and another JSON node j_2, respectively, the graph contains an edge $e = (v_1, v_2)$ if and only if j_2 is a property of j_1 in at least one document. Moreover, the edge carries `EdgeList`(e)—a list of NodeID(j_1) for each occurrence of such parent-child pair of nodes across the documents.

The SG is constructed by iterating over the input documents. In each iteration, the document's node structure is processed in preorder[5]. The processing of each node extends the SG by adding new vertices and/or edges, or modifying existing ones by appending to their NodeLists and EdgeLists.

JSON Schema generation step. In this step, the SG generated previously is converted into a textual representation. The authors choose JSON Schema for this purpose. The SG is processed in preorder, each vertex being converted to a JSON Schema definition describing the node which the vertex represents.

When extracting the schema, optional and required properties can be distinguished based on data contained within their related vertices and edges in the SG. Given an object j_1 and its child node j_2, two vertices v_1 and v_2 which correspond to j_1 and j_2, respectively, and an edge $e = (v_1, v_2)$, j_2 is a required property of j_1 if and only if $\texttt{EdgeList}(e) = \texttt{NodeList}(v_1)$. In other words, j_2 is a required property if and only if j_2 is a child of j_1 in *every* document where j_1 is present. This is reflected in the generated JSON Schema using the `"required"` keyword.

For a leaf v reflecting a node j, Type(v) is encoded in the schema generated for j. If j was of multiple different types in different documents, then Type(v) is a union of those types, recorded in the schema using the `oneOf` or `anyOf` keyword.

Homogeneity and outliers inference. The SG is carries more information than is needed just to construct a schema. Given two vertices v_1 and v_2 and an edge $e = (v_1, v_2)$, the size of $\texttt{EdgeList}(e)$ can be compared to the size of $\texttt{NodeList}(v_1)$ to detect the frequency of occurrence of that property in v_1's node.

This information can be used to detect structural outliers in the data set. These outliers are divided into two kinds, *additional properties* and *missing properties*. Additional properties are those that are contained only in a very small percentage of cases. Conversely, missing properties are contained in a very large percentage of cases, but not 100 %. The definition of "very small" and "very large" can be fine-tuned by the user by choosing an appropriate sensitivity.

More formally, the user chooses a number ϵ (sensitivity) between 0 and 1. Then given v_1 and v_2 which correspond to an object j_1 and its child node j_2, and an edge $e = (v_1, v_2)$:

- j_2 is an additional property of j_1 if and only if $\frac{|\texttt{EdgeList}(e)|}{|\texttt{NodeList}(v_1)|} \leq \epsilon$
- j_2 is a missing property of j_1 if and only if $1 - \epsilon \leq \frac{|\texttt{EdgeList}(e)|}{|\texttt{NodeList}(v_1)|} < 1$

The SG can also be used to determine other useful metrics about the inspected data by examining the EdgeLists and NodeLists. One of those is the degree of coverage, which for a subset of documents specifies the level of structural similarity or homogeneity within the subset.

The authors additionally propose a Reduced Structure Identification Graph (RG) as an alternative to the SG. The RG is constructed very similarly to the SG, except instead of each graph element containing a list of node IDs, the vertices contain only a number of occurrences, which in SG would be the size of the list. The edges in RG contain no additional information. This design variation reduces the memory footprint of the approach by trading away some capabilities,

[5] a depth-first tree traversal order in which a tree's root is first processed itself, then the left subtree is processed recursively in preorder, then the right subtree

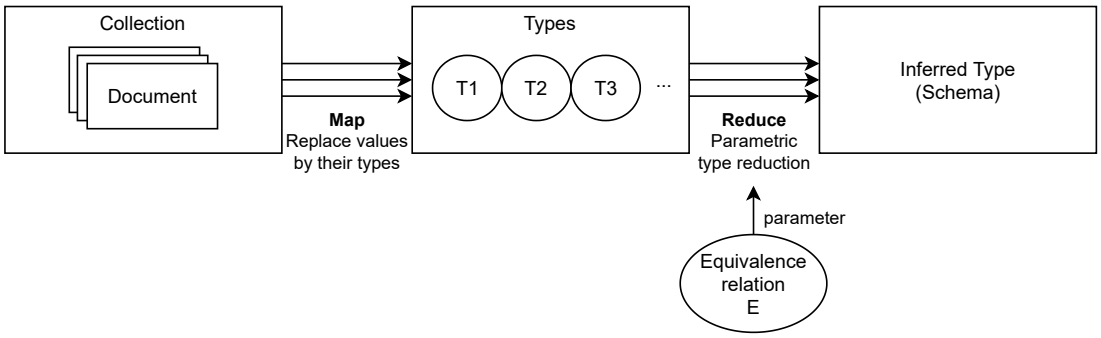


Figure 2.8: Baazizi et al. approach execution high level diagram

e.g. outlier detection. The authors note that the outliers can still be detected even when using the RG in some situations.

2.2.2.3 Running example

The output for the running example is included as attachment A.2. Being in the JSON Schema format, its structure follows that of the individual documents in the collection. Note that this output was created by hand and not by a computer program, i.e. the approach algorithm was executed manually using pen and paper according to the description in the article. The reason being that the implementation of this approach was not acquired from the authors and was not recreated due to time constraints.

All trivial JSON types are handled correctly in properties `_id`, `timestamp`, `ratings`, and `published`. `comments` property is inferred as either an array of strings, or a string, using the `oneOf` JSON Schema syntax. Similarly, `body` is inferred as either a string, or an object with two required properties.

`attachments` property is detected as an array of objects containing a `url` string property. It is marked as optional by not being included in the `"required"` list. Same goes for the `article_id` property, which is inferred as just a number without any special meaning.

The `author` object's `phone_number` property is inferred as a string or a number, again utilizing the `oneOf` construct. Finally, the `location` object has three optional properties, `latitude`, `longitude`, and `address`. No dependency is inferred between these three.

2.2.3 Baazizi et al.

Baazizi et al. introduce in their article [15] a schema inference approach for massive JSON datasets. The approach is based on a formal definition and the authors provide rigorous proofs of its properties. It can be parameterized by the user to trade off conciseness for precision in the inferred schema. It is designed to be easily parallelizable and scalable and outputs the inferred schema in a simple yet expressive JSON-like type language. The authors provide an implementation based on Apache Spark.[6]

[6] `https://spark.apache.org/`

The algorithm accepts a single JSON document collection as input. It operates in two steps. In the first step, a parallel Map operation transforms each of the input documents into their simple type representations. The second step, a Reduce operation over the results of the Map operation, parametrically reduces the simple types into one resulting type.

2.2.3.1 Motivation

The authors of this approach quote as their motivation, the benefits of determining a schema for an already existing JSON document collection. These include the ability to perform static checks of queries over the dataset, ability to use schema-based querying optimization strategies, and the increased ease and speed of producing correct code, which consumes the dataset.

The authors quote *speed* of the algorithm and *precision* and *conciseness* of the resulting schema as their main focus points. They note that the latter two schema properties are inherently contradictory. A precise schema is usually verbose, and a concise schema risks not being able to describe the data completely. This contradiction is solved by letting the user parameterize the inference process to tweak the properties of the inferred schema.

Another focus point of the approach's design is its ability to work with massive datasets. This property is crucial for schema inference approaches in the Big data context. To that end, this approach was designed from the ground up to be parallelizable and horizontally scalable using MapReduce and the Apache Spark framework.

Yet another important desired property of the generated schema is that it is a *path covering* schema, i.e. each path that can be traversed within any of the input documents can also be traversed in the schema. A schema with this property can be used to enable many JSON query optimizations as mentioned in the article.

2.2.3.2 Technology used

The approach uses Apache Spark to provide scaling capabilities necessary to handle even datasets of massive sizes. Apache Spark is a cluster data processing framework with implicit parallelism, work distribution, and fault detection features. It builds upon the MapReduce paradigm by allowing an arbitrary number of steps in the data processing pipeline and storing the intermediate results in distributed data structures in the memory.

2.2.3.3 Detailed algorithm description

As mentioned before, the algorithm divides into two steps, a Map step and a Reduce step. These are executed sequentially on the input collection, however each step can itself be parallelized and executed in a distributed environment. We will now more closely describe their inner workings.

Map step. In this step, each JSON document in the collection is *mapped* into its *type*. While the concept of a document's type is not unlike that of raw schemas, discussed previously in description of the approach of Sevilla et al. in section 2.2.1.4, there are key differences.

Similar to a raw schema, a type keeps the original structure of the JSON value and all primitive values inside it are replaced by the names of their primitive types. However, unlike the raw schema, the types of JSON array elements are not kept separate, instead they are reduced to their common supertype, using the same type reduction process which is used later in the Reduce step. It is non-trivial so we will describe it then.

More formally on the topic of mapping JSON values to types, the algorithm recognizes types of 6 kinds. Four of them, `Str`, `Num`, `Bool`, and `Null`, correspond to the JSON primitive types. The remaining two kinds of types are structural, they describe the JSON array and the JSON object. The type of JSON array A is $[\mathcal{T}]$, where $\mathcal{T}$ is a supertype of all elements of A. The type of JSON object $O = \{a_1 : V_1, ..., a_n : V_n\}$ is a record type $\{a_1 : \mathcal{T}_1, ..., a_n : \mathcal{T}_n\}$, where $\mathcal{T}_i$ is the type of V_i for all i from 1 to n.

Reduce step. In this second and final step of the algorithm, all mapped types from the previous step are *reduced* into one type. The only requirement for this type is that it must be a common supertype for all of the input types. This requirement is somewhat loose as there are many ways how to construct a supertype for a set of given types. This looseness leaves room for parametrization of the reduction process, which enables us to gain some control over the properties of the reduced type.

This parametrization comes in the form of an equivalence relation E over types. If two types $\mathcal{T}_1$ and $\mathcal{T}_2$ are E-equivalent (we write $\mathcal{T}_1 \overset{E}{=} \mathcal{T}_2$), a more succinct and possibly less precise reduction is used. Otherwise, a more precise but possibly more verbose reduction will be used for two types which are not E-equivalent. By correctly choosing the parameter E, we can achieve a desired balance of conciseness and precision in the inferred schema.

Parameterized binary type reduction has been proven in the article to be commutative and associative, which means that to reduce a set of types into a common supertype, one only needs to repeatedly apply the binary reduction onto the elements of the set, and the order of operations does not matter.

Two supertype constructors are used in the approach. The trivial supertype constructor is a union. A union of two different types $\mathcal{T}_1$ and $\mathcal{T}_2$ is $\mathcal{T}_1 + \mathcal{T}_2$ and can be used with all input types. However, it produces supertypes which are verbose and can be hard to read and understand.

Another supertype constructor is a fusion, $\texttt{Fuse}(\mathcal{T}_1, \mathcal{T}_2, E)$. It creates supertypes which are more concise but can lose information in the process. A fusion can only be used for structural kinds and only for two types of the same kind. A fusion of two array types is achieved by recursively reducing their component types. A fusion of two record types is a record type, where:

- properties present in both record types are marked as required and their types are recursively reduced
- properties present in only one of the types are marked as optional

The type reduction function is then defined as:

$$\texttt{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E) = \begin{cases} \texttt{Fuse}(\mathcal{T}_1, \mathcal{T}_2, E), & \text{if } \mathcal{T}_1 \overset{E}{=} \mathcal{T}_2 \\ \mathcal{T}_1 + \mathcal{T}_2, & \text{otherwise} \end{cases}$$

Two different equivalence relations can be used by the user as possible parameters. One of them is a *kind equivalence relation* (KER) which fuses together any two types of the same kind. This means that any two record types are fused together regardless of the properties present. This may in some cases cause loss of information related to property correlation in the data. On the other hand, the resulting record type is nearly as readable as either of the two input types.

The other useful equivalence relation is a *label equivalence relation* (LER). Similarly to KER, it fuses together types of the same kind, except for record types, for which there is an additional requirement. LER fuses record types together only if the property names within them match, i.e. they contain the same number of properties with the same names, but not necessarily the same types. This equivalence relation will preserve property correlation information in the resulting type, at the expense of increasing the size and decreasing readability of the type.

2.2.3.4 Running example

The output of this approach, included as attachments A.3 and A.4 is a textual representation of the inferred type. The output has a similar structure as the input in that the root elements of the schema correspond to the input documents.

We first describe the output of the approach using *kind* equivalence. Then we describe the differences between that and the output of *label* equivalence.

Kind equivalence. The properties `_id`, `timestamp`, and `published` are inferred as `Num`, `Str`, and `Bool` respectively. `ratings` is inferred as `[ Num ]` type, the difference in element count is not modeled.

`comments` property type is a union of `Str` and `[ Str ]`, as these are two distinct types. Similarly, `body` property is inferred as a union of `Str` and a record type `{ content: Str, mime_type: Str }`. And also the `phone_number` property of the `author` record is inferred as a union of `Num` and `Str`. This illustrates the versatility of the union typing used by this approach.

`attachments` is simply modeled as an array of records `[ { url: Str } ]`. The special meaning of `article_id` is not inferred, as it is inferred simply as a `Num`. Notably, the `author/location` record has three properties, all of which are typed as `Str`.

In all of these three instance, properties are missing the quantifier `?`, which the article claims is used to model optional properties. An inspection of the source code reveals that detection of optional properties was not implemented by the author. However, this seems to be merely an oversight, as implementing this functionality would be rather simple and straightforward.

With the detection of optional properties correctly implemented, all three properties within the `author` record would have the type `Str?`. This would not, however, correctly model the correlation between the `latitude` and `longitude` properties, and their mutual exclusivity with the `address` property. This additional information can be inferred using the label equivalence.

Label equivalence. Using label equivalence yields two different versions for the top-level record, as they have different properties each. Since both documents are homogeneous within themselves, there are no other union types anywhere else than the top level. An array with mixed element types could generate unions, however there are no such arrays in the running example.

The output using label equivalence illustrates the aforementioned correlation between the individual properties in the `author/location` record.

2.2.4 Canovas et al.

Canovas et al. propose another schema inference approach [16] based on MDE principles. The algorithm is able to discover the schemas of multiple collections of JSON documents, then merge them to form a complete view of the application domain. In addition, it can use the inferred schema to discover dependencies (like common entities) between multiple services and generate call execution chains for the developer to get the data they require.

The approach's execution consists of three steps, followed optionally by further processing. Firstly, for each collection of JSON documents, those documents are injected into a JSON metamodel generated by Xtext[7], which yields a collection of JSON models, one model for each document. This is called the *Pre-discovery phase*. In the second step, so-called *Single-service discovery*, each JSON model collection is reduced iteratively into a single domain model. Finally, in the third step, so-called *Multi-service discovery*, all domain models are combined into a single domain model for the entire application. Additionally, the domain model can be analyzed to discover dependencies (like common entities) between the collections.

2.2.4.1 Motivation

This approach's motivations lie directly in facilitating developer's work with existing JSON web-based services.

Designing applications which consume multiple related JSON services and exchange data with them is a non-trivial task, during which the developer can unintentionally commit costly mistakes.

A standardized language for schema definition of JSON documents does not exist. The aforementioned JSON Schema project is promising but has not yet reached Request for Comments (RFC) status with the Internet Engineering Task Force (IETF). As a result, most of the available documentation for JSON services is in plain human language with examples of requests and returned data. This makes the issue with designing JSON service consumer applications even more severe.

Generating and visualizing the data model of JSON services on demand would greatly enhance a developer's ability to understand the data format of single services, infer the relationships between services, and even make it possible to generate service request compositions, resulting in faster development of service consumer applications.

JSON web services can be queried repeatedly to obtain JSON documents. Different documents retrieved from a single service usually describe the same entity or concept and are very similar to each other. In this sense, assuming a representative set of documents has been retrieved from a web service, that set of documents, and in fact also the web service itself, is equivalent to a document collection of a document-based database. Then, a web application consisting

[7]Framework for developing programming languages, `https://www.eclipse.org/Xtext/`

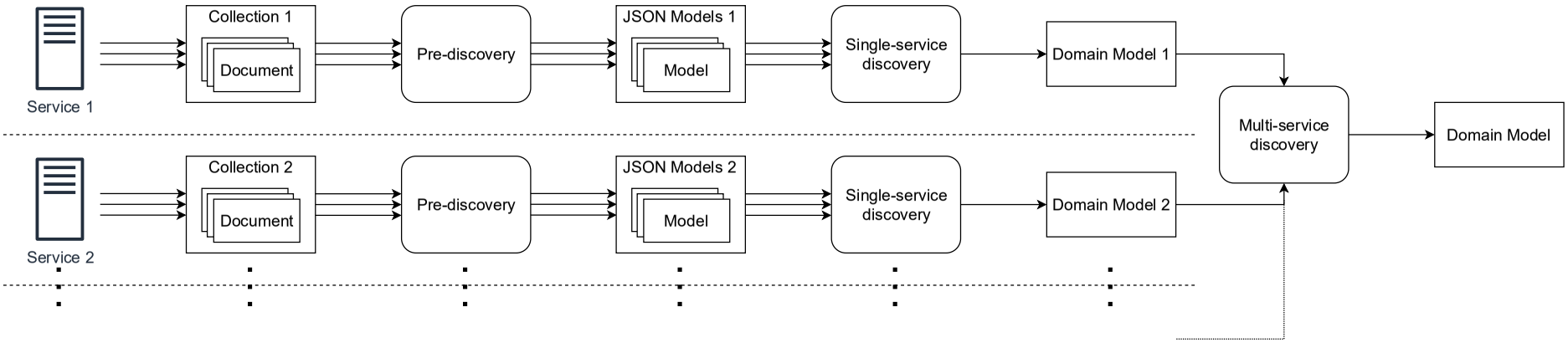


Figure 2.9: Canovas et al. approach execution high level diagram

of multiple services is equivalent to a database containing multiple document collections.

We can then assume that the Canovas et al. approach accepts as input of the algorithm JSON document collections themselves rather than web services that serve them. We assume a representative sample of documents is retrieved from each service to facilitate this reduction. While we still keep in mind the motivational background of the approach and mention it where it's due, this assumption is beneficial for the sake of simplicity of our comparison.

2.2.4.2 Technology used

Xtext is a framework for development of programming languages and domain-specific languages. It enables the user to formally define a custom language's syntax using Xtext's language definition language. The language definition looks and works in a way similar to a formal grammar. Xtext then uses this definition to generate tools like parser, compiler, and converters to various non-textual formats like EMF models.

2.2.4.3 Detailed algorithm description

The first prerequisite step of the schema inference (or schema discovery, as the authors call it) process is provision of JSON documents by the user. The input of the discovery process are textual JSON documents, which must first be extracted, either manually or by an external automated tool, from the web-based services which the user wants to infer the schema of. As we have assumed in section 2.2.4.1, the sample must be large enough to be representative of the service's data model.

The first phase of the discovery process proper is called the *Pre-discovery phase*. As the authors say, the approach works with two different technical spaces, grammarware and modelware. On the one hand, it takes its input in the form of collections of JSON documents. Those conform to the JSON grammar and thus belong to the grammarware technical space. On the other hand, the processing and output of the approach are in JSON models which conform to a JSON metamodel and belong to the modelware technical space.

For this approach, the authors use Xtext's language definition syntax to create an Xtext language definition D describing the syntax of JSON documents. D defines a JSON formal grammar G to which all JSON documents conform. Afterwards, they use Xtext to generate from D:

- a JSON metamodel M, and
- a so-called *injector* I.

I is a program that takes as input JSON documents in their textual representation (conforming to G) and produces as output JSON models (instances of M) which represent the input documents. This entire process is illustrated by figure 2.10.

The second phase is called the *Single-service discovery* phase. During this phase, collections of JSON models from each service, obtained from the *Pre-discovery phase*, are iteratively folded to an Ecore domain model describing that service. The identity for the fold is an empty Ecore model, and the combining

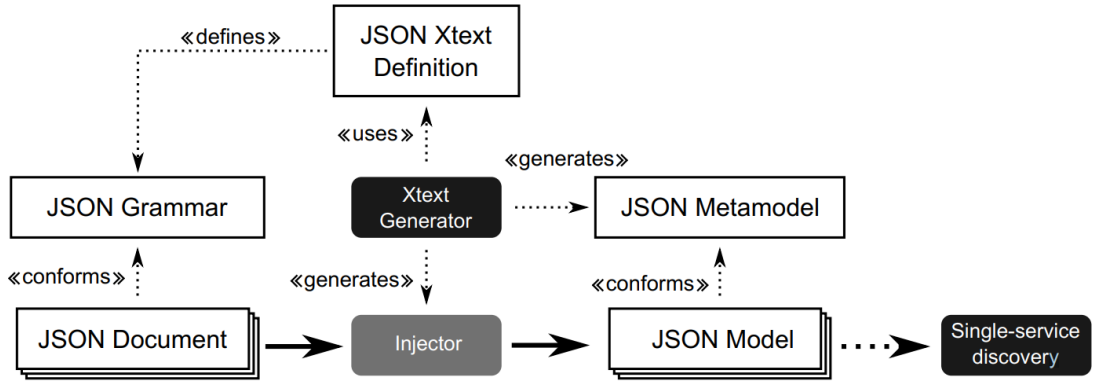


Figure 2.10: Pre-discovery phase

function is defined by a set of model transformation rules. Without further detail, these rules are divided into two categories based on whether they create new previously undiscovered concepts, or refine already discovered concepts.

Individual services provide different viewpoints on the application's data. Their domain models can be thus viewed as sub-models of the entire application's domain model. To be most helpful to the developer, they need to be combined together and accessible as one whole. The third phase called *Multi-service discovery* achieves this result.

This phase begins by creating a new model as a union of all partial models. Afterwards it repeatedly matches the model against a set of transformation rules to try to merge the sub-models. The resulting merged domain model describes data of the entire application and is the result of the schema inference process.

Additional post-processing can be applied to the model. The authors describe adding coverage information to the model, which contains the relationships between individual elements of the model and the services that return them. This meta-data can be used to identify the services that need to be queried in order to obtain a specific subset of properties.

The coverage information can be further expanded by the user manually specifying the input parameters of each service. This information can in turn be used by the algorithm to discover dependencies between multiple services, where the output of one service can be used as input of another. The inter-service dependencies can be organized into an oriented graph. The dependency graph can then be queried to find optimal sequences of service execution to obtain all desired data.

2.2.4.4 Capabilities

This approach is specialized towards JSON data format. This limits the approach's usability, compared to e.g. Sevilla et al. approach which is applicable to all aggregate-oriented NoSQL data stores.

The Canovas et al. approach is not able to detect optional properties of an entity.

The ability to handle situations where the same entity property has values of different simple types is also limited. Namely, the approach does not support union types to resolve these situations, instead it reduces the involved types into

their most generic type.

The approach also cannot handle identifier-based references to other entities in the data.

All phases of the discovery process up until and excluding Multi-service discovery are parallelizable by design, although this fact is not specifically mentioned by the authors. Instances of Pre-discovery and Single-service discovery could be launched manually in parallel on multiple machines, since they are implemented as separate software modules. Their output could then be processed by a single instance of a Multi-service discoverer. This fact could facilitate good horizontal scalability of this approach.

2.2.4.5 Running example

Being an Ecore model serialized as an XML document, this approach's output is similar to the output of the approach by Sevilla et al. It is included as attachment A.5. The root element is an `EPackage` containing `EClass` elements. The inferred `EClass`es are `Article`, `Author`, `Attachment`, `Body`, and `Location`. The names are inferred from the name of the document collection and the names of their respective JSON properties.

`EClass`es in turn contain `EAttribute`s and `EReference`s. The former have an `EDataType` which in our running example can be an `EInt`, (e.g., for `_id`), `EString` (for `timestamp`), or an `EBoolean` (`published`). The latter on the other hand have an `eType` attribute, which defines the type of the referenced entity.

The different sizes of the `ratings` array are not mirrored in any way in the schema. Different types of the `comments` property are reduced into the most generic type, which is `EString`. Conversely, with the `body` property, the `EString` version is lost and only the version referencing a nested `Body` object is retained.

`attachments` property's optionality is not modeled directly using Ecore's `lowerBound` and `upperBound` attributes, but could be inferred from a nested coverage element. Same applies to the optional property `article_id`. Moreover, the special meaning of `article_id` as a reference to an `Article` entity was not inferred.

Finally, let's cover the `author` object. The `phone_number` property's type was reduced into an `EString` which is the most generic type. As for the `Location` `EClass`, only the `latitude` and `longitude` properties are present, while the `address` property is missing entirely.

2.2.5 Frozza et al.

Frozza et al. propose an approach [18] for extracting a schema from a JSON or MongoDB Extended JSON document collection. The approach utilizes aggregation operations to generate schemas for uniquely structured documents in the collection, then makes use of a graph structure to combine the partial schemas into a collection schema. The approach outputs the inferred schema in the JSON Schema format. An implementation of the approach is available as a web tool.[8]

[8] `http://lisa.inf.ufsc.br/wiki/index.php/JSON_Schema_Discovery`

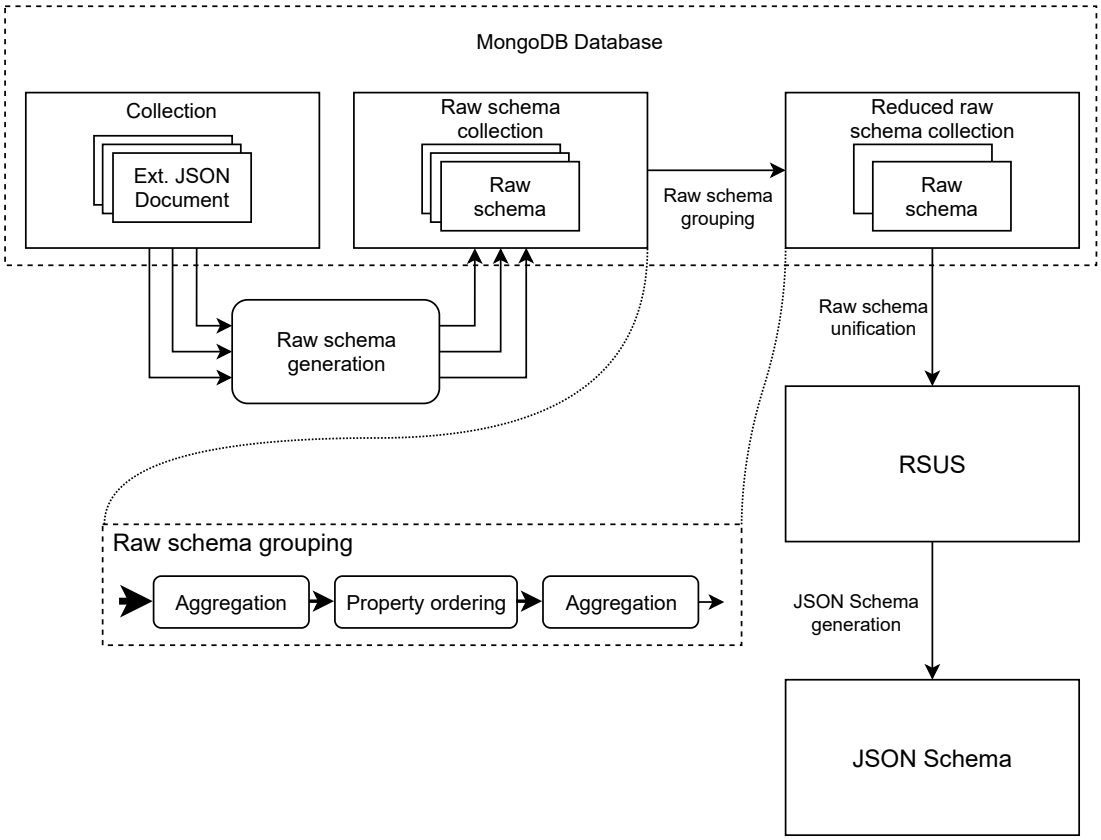


Figure 2.11: Frozza et al. approach execution high level diagram

2.2.5.1 Motivation

The motivation for this approach is similar to previously discussed works. In this instance, the authors focus on JSON document-based NoSQL stores, specifically on MongoDB and its Extended JSON format, i.e. BSON. The fact that there almost always exists implicit schema in the data despite the lack of explicit schema definition is specifically mentioned. Examples are mentioned of web services which provide data in JSON format, but do not define its schema, making querying the data difficult.

2.2.5.2 Detailed algorithm description

The schema inference process is divided into four logical steps: *Document raw schema generation*; *Grouping of raw document schemas*; *Unification of document raw schemas*; and *JSON Schema generation*.

Document raw schema generation. This step is similar to the first step of the Sevilla et al. approach described previously in section 2.2.1.4. A raw schema is extracted from all documents, leaving the document structure with relation to properties, nested arrays and objects intact, but replacing all primitive values with the name of their type. This applies to both JSON and Extended JSON primitive types. The resulting raw schemas are saved in a temporary MongoDB collection.

Grouping of raw document schemas. The next step aggregates the raw schemas in order to find their minimal subset sufficient for correct unified schema generation. The aggregation is performed in three substeps.

First, a MongoDB aggregation is used to remove all duplicate raw schemas from the collection. However, the document properties may not be in the same order in all documents in the collection. The first aggregation would not remove duplicate raw schemas, which have their properties in a different order. Therefore, in the second sub-step, all remaining schemas have their properties sorted alphabetically. This sorting operation is executed on the documents and recursively on their nested objects. Finally, another aggregation removes any remaining duplicates.

Unification of document raw schemas. Thirdly, aggregated raw schemas are unified into a tree-based hierarchical structure called the Raw Schema Unified Structure (RSUS). RSUS is loosely defined in the article. It is a tree composed of vertices of different types: *FieldType*, *PrimitiveType*, *ExtendedType*, *ObjectType*, and *ArrayType*, which represent object properties, base JSON primitive types, Extended JSON types, JSON objects, and arrays, respectively. Additionally, it carries information about the path from document root to any node within the document, and the number of occurrences of those nodes within the collection.

Authors claim that the usage of RSUS and the operations performed on it, both its creation and later conversion into JSON Schema, are inspired by MDE principles, which are suitable for data model transformations.

To construct the RSUS, the aggregated raw schemas are processed one by one. Each raw schema, being a tree of JSON nodes, is processed in preorder. When processing a JSON node, if the node has not been encountered yet, a new RSUS vertex is created. Alternately, if a previously encountered node is encountered again in another document, an extension of an existing RSUS vertex occurs.

JSON Schema generation. Lastly, the RSUS is converted into a JSON Schema representation. A JSON Schema definition is generated for each vertex of RSUS, based on its type, its path from document root, and its number of occurrences, and then they are merged together, preserving the hierarchical structure. Notably, the occurrence count in FieldType vertices is used to determine whether those properties are required or optional.

2.2.5.3 Capabilities

Since the Frozza et al. approach is similar in its methodology to the approach by Klettke et al., this translates into similarities in the approaches' capabilities. Speaking of structural features, the approach is able to detect both the optionality of object properties, and the presence of union types in properties. These are determined based on the numbers of occurrences of properties and data types in the RSUS. The approach is not able to detect properties which represent referential relations to entities.

The approach was built from ground up to work not only with data in JSON but also Extended JSON as well. This gives it a major advantage in versatility compared to other approaches we have researched due to the very high popularity of MongoDB in the field.

Conversely, this approach carries the disadvantage of being unable to handle large data sets efficiently and to scale. While this aspect was not yet tackled by the authors, they claim "adopting parallel processing techniques" as one vector of their future work.

2.2.5.4 Running example

For the Frozza et al. approach, the Extended JSON version of the running example is used to illustrate its ability to infer schemas from Extended JSON data. The output is included as attachment A.6.

The schema inferred for the running example follows the structure of the input documents, as is standard for JSON Schema schemas.

All the common properties have their types correctly inferred, including the Extended JSON-typed ones, `_id`, `timestamp`, and `article_id`. However, the target of the last property was not detected.

Strangely, the number of items in the `ratings` array has been inferred to be at least one.

`comments` property's type was correctly inferred as a union of **string** and **array of string**. The type of `body` was inferred as a union of **string** and **object**, which is also correct. `attachments` property is also correctly modeled as an **array of objects**. Same applies for the property `author/phone_number`, which is modeled as a union of **string** and **number**.

The `attachments` and `article_id` properties are both treated as optional by being omitted from the list of required properties. And, within the `author/location` object, all three properties are optional, as the array of required properties is empty. No property presence correlation is inferred.

2.3 Comparison

After researching and explaining in detail individual existing solutions and presenting how they handle the running example, let us summarize the approaches' properties and performance in regards to the quality criteria we established at the beginning of this chapter.

For this section, difference between input *format* and *type* is as defined in section 2.1.2.1.

2.3.1 Input format

Approach by Baazizi et al. expects input in JSON format and was designed solely with this format in mind.

Canovas et al. approach article talks specifically about JSON documents which are responses from JSON-based web services. Practically speaking, however, there is little difference between those and JSON document collections in a database.

Frozza et al. approach offers a key advantage in that it supports also Extended JSON data types.

Approaches by Sevilla et al. and Klettke et al. work primarily with JSON data, however their articles mention how other aggregate-oriented databases can be trivially converted to JSON. Notably, this conversion process can be applied universally to aggregate-oriented data, so that in practice any schema inference approach for JSON data can be used for key-value and columnar data as well.

2.3.2 Input type

Dividing the existing approaches by input type yields two categories. Sevilla et al. and Canovas et al. approaches can process an entire JSON database with multiple collections, while the other three, Klettke et al., Baazizi et al., and Frozza et al. approaches support only inferring from a single document collection at a time. Merging these collection schemas is then left up to the inference tool user.

Note that both approaches capable of handling multiple collections utilize MDE to do so.

2.3.3 Output format

The output format criterion also divides the approaches into two main categories: textual and model.

Klettke et al., Frozza et al., and Baazizi et al. approaches all output the inferred schema in a textual format. The first two use JSON Schema, while the last one uses its own type description language, more concise and user-readable than JSON Schema, but still expressive enough to describe all schema features the approach is able to infer.

The last two approaches by Sevilla et al. and Canovas et al. both output the inferred schema as a data model. As their model's metamodel, the latter approach uses the basic Ecore model from EMF, while the former approach's authors define a new metamodel called NoSQLSchema specifically for this purpose.

Taking a look at this criterion with the multi-model context in mind, there is a concern, how well-suited are these output formats for describing a schema inferred from a multi-model inference approach. Since all of the existing inference approaches we consider are single-model, it is understandable that their authors did not concern themselves with this aspect. Nevertheless, this question is something that must be taken into account when designing such multi-model solution.

2.3.4 Implementation

All five approaches were implemented for the purposes of their respective articles. During our research, we only managed to obtain four out of five of these implementations. The article by Klettke et al. claims their approach was implemented for performance testing as a Python application, however the source codes of this implementation are unavailable.

The approaches of Sevilla et al. and Canovas et al. were implemented as Java applications running on the Eclipse platform. Both of them offer usable Java API and a simple web application wrapper for convenience.

Baazizi et al. implemented their approach as a Scala application designed to run in an Apache Spark environment.

Frozza et al. approach is implemented as a JavaScript web application. The app's front end is written in TypeScript and provides the user with a presentation layer, while the back end is written using Node.js and contains the actual implementation of the inference tool.

2.3.5 Simple data types

All researched approaches were primarily designed for work with JSON, and as such, all of them support each simple (primitive) JSON data type: number, string, boolean, and null.

2.3.6 Arrays

The approaches by Klettke et al. and Frozza et al. describe arrays by the `{"type":"array"}` descriptor in the JSON Schema. The type of the items of the array is inferred as a union (using `oneOf` keyword) of the individual types present.

Baazizi et al. approach describes arrays in their type description language as a pair of brackets (`[]`), with an element type inside. The element type is the result of the parameterized reduction of the types of all items inside the array. This reduction can result in a union, in which case the union operator (`+`) is used.

To reflect the concept of arrays in their output models, the approaches by Sevilla et al. and Canovas et al. each use a different method to suit their metamodels of choice.

Sevilla et al. approach uses a `PList` type to describe an array of primitives, and an `Aggregate` property to model an array of objects.

When met with an array of simple values, Canovas et al. approach uses an `EAttribute` structural feature with a removed upper bound (`upperBound="-1"`).

When describing an array of objects, an `EReference` with `containment="true"` is used instead.

2.3.7 Objects

The way of handling objects is analogous to handling arrays in all researched algorithms.

In the JSON Schema-based approaches, objects are described by a `{"type": "object"}` declaration, along with the `"properties"` keyword describing the individual properties of the object, and the `"required"` keyword specifying the properties' optionality.

Baazizi et al. type language describes objects as a pair of braces (`{}`) containing a key-value pair for each of the object's properties.

2.3.8 Aggregates

Approaches by Sevilla et al. and Canovas et al. use an aggregation relationship to model a nested object. They create an entity type for the object, then link it with its parent using an appropriate structural feature. Sevilla et al. approach uses the `Aggregate` property type with the `aggregates="..."` attribute to encode this relationship, while Canovas et al. uses an `EReference` with a `containment="true"` attribute.

2.3.9 References

Out of all considered existing approaches, only the one by Sevilla et al. can detect references to other entity type. The heuristics for this detection consists of matching the property name for a specific pattern, e.g. `"*_id"`, and detecting Extended JSON `DBRef` types. Constraint checking is performed on a detected reference to increase the inference certainty.

There can be, and in practice many times are, references not matched by these heuristics. E.g., in a company hierarchy domain, entity type *Employee* can have a property named `manager_id`, or simply `manager`, which is a reference to another *Employee*. In this case, the simple heuristic approach would not detect this property as a reference, because there is no entity type named *Manager*. Moreover, the `-_id` suffix may not be present at all.

Consequently, there is much room for improvement in the field of correctly inferring referential relationships in semi-structured data.

2.3.10 Optional properties

All approaches except for Canovas et al. are able to describe optional properties in their schemas. JSON Schema-based approaches, Klettke et al. and Frozza et al., use the `"required"` keyword to enumerate the required properties, while all omitted are optional. Baazizi et al. approach uses the optionality modifier (`?`) to describe optional properties when a *kind*-equivalence relation is used in the reduction.

Approach by Sevilla et al. can infer optional properties by merging all versions of a single entity together, marking each property as required if it is present in

all of them, and optional if it isn't. This merging process, transformation of multiple entity versions to one version with optional properties, is analogous to switching from using a label-equivalence relation to a kind-equivalence relation in the Baazizi et al. approach.

2.3.11 Entity versions

Two approaches are able to infer multiple versions of entity types; Sevilla et al. and Baazizi et al. The latter can infer multiple versions of the same entity when using the *label-equivalence* relation in the reduction process. No other approaches can infer entity versions.

2.3.12 Union type

Klettke et al., Frozza et al., and Baazizi et al. approaches can infer and express union types in the schema. The first two use the `oneOf` JSON Schema keyword to express them while the last one defines for this purpose the union type constructor (`+`).

Approaches by Sevilla et al. and Canovas et al. do not support union types of properties. Sevilla et al. approach uses entity versioning instead while Canovas et al. uses the alternative approach of reducing different types to their most generic type, like `EString`.

2.3.13 Integrity constraints

As far as integrity constraints go, only one specific type is detected only by the inference approach by Sevilla et al. and that is referential relationships. No other integrity constraints are inferred by any of the researched inference approaches from the given data set.

2.3.14 Scalability

Approach by Sevilla et al. uses parallelizable technology MapReduce to decrease the number of input documents that are considered in the rest of the schema inference process. In the Map step, a raw schema is generated for each input document in the collection and emitted as a key, with the value being the input document itself. In the Reduce step, an arbitrary input document is selected and emitted as a representative for its raw schema.

This process effectively eliminates from the input collection all documents with the same raw schema, leaving only one representative per each raw schema. Since the document's raw schema is used in the remainder of the inference process, this step has no effect on the resulting inferred schema, while also having the potential to significantly reduce the size of the input collection. This reduction step allows the inference approach to scale very well with the input collection size.

Approach by Frozza et al. includes a two-stage grouping step which extracts a sub-collection of unique JSON objects. Having already generated and stored a raw schema for each input document in a NoSQL database, first an aggregation operation removes duplicate raw schemas from the database. Then the properties of the raw schemas are recursively sorted in alphabetical order before another

aggregation operation removes all raw schema duplicates which were semantically equivalent but their properties might not have been originally defined in the same order.

2.3.15 Incremental schema extensibility

The approach by Baazizi et al. is incrementally extensible, given by the formal proof of associativity and commutativity of the type reduction operation.

As far as the other inference approaches go, their main schema construction steps are designed as a *fold* on a collection of partial schemas or documents.

Definition 8 (Fold)**.** *Given*

- $\mathcal{A}$*, a non-empty set of input elements,*
- $\mathcal{B}$*, a non-empty set of output and intermediate elements,*
- $f : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$*, a* combining operator*,*
- $b_0 \in \mathcal{B}$*, an* initial value*, and*
- $L = (a_1, ..., a_n)$*, a finite sequence (list) of elements of* $\mathcal{A}$*.*

Then fold *(also called* reduce*) is a higher-order function* F *defined recursively as:*

$$F(L, b_0, f) = \begin{cases} b_0 & \text{if } L = () \text{ is empty,} \\ f(a_1, F((a_2, ..., a_n), b_0, f)) & \text{otherwise} \end{cases}$$

In the case of Sevilla et al., the fold is performed over a schema collection of documents, the combining operator combines these into an initial empty model, which is then returned. In the algorithm by Klettke et al., the input documents are folded into an SG, which is then converted into JSON Schema. Similarly in the approach by Frozza et al., except here, the documents are folded into an RSUS. Lastly, Canovas et al. approach injects all of the available documents into models, then performs one fold per document collection, then one final fold to merge the collection schemas into one domain schema.

Each of these approaches provide an incrementally extensible schema as long as the associativity and commutativity of the combining operators used in their respective fold operations is guaranteed. The operators seem to have these qualities, however a formal proof would clear out any doubts in this situation.

2.3.16 Multi-model context

All researched approaches are created for inferring schema from JSON documents. These can be trivially extended to work with at most other aggregate-oriented data models, as was described earlier. None of these, therefore, handle any other than the single-model context of aggregate-oriented data.

2.3.17 Edge-case example

In this section we describe the outputs of individual researched approaches on the additional edge-case example defined in section 2.1.5.1.

2.3.17.1 Sevilla et al.

The Sevilla et al. approach implementation has failed to execute when met with the edge-case example. The exception is caused by the empty JSON array value in the `empty_array` property being incorrectly handled. After removal of this property, the approach executes correctly and produces an output. This output is included as attachment B.1.

The `nested_ints` property is handled correctly; it is modeled in the result as a `PList` of a `PList` (a two-dimensional `PList`) of `Number`s.

The `nested_objects` property's value, on the other hand, is not handled correctly. Its type is inferred as a two-dimensional `PList` of an unnamed primitive type. There exists a definition for a `Nested_object` entity in the model, but it is not used.

The `values` property is detected as a `PList` whose element type is a `PTuple`. The `PTuple` element type enumerates the inferred types of all the array elements in order.

2.3.17.2 Klettke et al.

Unfortunately, we do not present an output for the Klettke et al. approach on the edge-case example. There are two reasons for this decision: (1) a machine-generated output can not be created since no implementation of the approach is available, (2) unlike the running example, whose purpose to illustrate the functional capabilities of an inference approach, the edge-case example's purpose is to test the limits of a inference approach's design and implementation. During a hypothetical manual pen-and-paper "execution" of the approach on the edge-case example, one could easily create a different output than what would be outputted by a proper approach implementation. This could in turn vastly change the verdict of how well the approach can deal with edge-case JSON array inputs.

2.3.17.3 Baazizi et al.

Same as with the running example, approach by Baazizi et al. provides two versions of inferred schema for the edge-case example as well, one for the kind-equivalence (attachment B.2) and one for the label-equivalence relation (attachment B.3). We first cover the former version, then we cover the differences between the two.

The `empty_array` property is inferred as an array of the `Empty()` type, which is a type that carries no information. The empty type is denoted in the related article using $\emptyset$.

Property `nested_ints` is inferred as a two-dimensional array of numbers, `[[Num]]`, and the `nested_objects` property as a two-dimensional array of objects with a string-typed property called `key`, `[[{ key: Str }]]`.

Finally, `values` is described as an array of a union of all types which were present in the property, with some of those types merged together. Notably, the empty object and object containing one property were merged together (although the optionality modifier `?` is missing), and the empty array and array of numbers were merged into one type.

The only difference between the kind- and label-equivalence output is that in the latter, the empty object type was not merged with the `{ key: Str }` object type.

2.3.17.4 Canovas et al.

The output of the Canovas et al. approach for the edge-case example is included as attachment B.4.

The `empty_array` property is inferred to be of type `EString`. This is probably an artifact of reduction to the most generic type.

The `nested_ints` property is detected as an `EAttribute` of `EInt`s with an infinite `upperBound`, which is simply an attribute of multiple integers. The multi-dimensionality of the array is not reflected.

The `nested_objects` property is inferred as an unbounded `EReference` to the `Nested_object` entity type – a one to many aggregation which, again, ignores the multi-dimensionality of the array.

Lastly, the `values` property's type is reduced to the most generic type, which is an `EString`.

2.3.17.5 Frozza et al.

Frozza et al. approach's output for the edge-case example is included as attachment B.5.

The `empty_array` edge case is not handled correctly by this approach. The sub-schema generated is of the correct type, `array`, but specifies that its `items` must conform to any of an empty array of schema options (`"anyOf":[]`). This specification is not only wrong semantically, as the existential quantifier for an empty set always yields false, but it also violates the JSON meta-schema, which explicitly specifies there to be at least one member in an `anyOf` value.

The `nested_ints` property is correctly detected as an array of arrays—a two-dimensional array—of numbers. Same goes for `nested_objects`, which is inferred as a two-dimensional array of objects.

The `values` property is described in the output schema as an array with union-typed items. The union contains the types inferred from the individual items in an `anyOf` array.

	Sevilla	Klettke	Baazizi	Canovas	Frozza
Algorithm	Map-Reduce + MDE	Fold into graph	Type reduction in Apache Spark	MDE	Aggregation + fold into graph
Input format	Aggregate-oriented NoSQL data	JSON	JSON	JSON web service responses	Extended JSON
Input type	Multiple collections	Single collection	Single collection	Multiple collections	Single collection
Output format	NoSQL Schema model	JSON Schema	Custom textual type language	Ecore model	JSON Schema
Schema root	Entities	Documents	Documents	Entities	Documents
Implementation	Eclipse bundle	Python application	Apache Spark application in Scala	Eclipse bundle	Node.js web application
Optional	✓	✓	✓	✗	✓
Entity versions	✓	✗	✓	✗	✗
Union type	✗	✓	✓	✗	✓
References	✓	✗	✗	✗	✗
Scalable design	✓	✗	✓	✓	✗
Scalable implementation	✓	✗	✓	✗	✗

Figure 2.12: Comprehensive comparison table for all discussed approaches

3. Design

In this chapter, we provide a detailed description of a novel schema inference approach, explain the motivations behind multiple of its design decisions, and pinpoint the added value it brings.

To reiterate, the goal of our design is to create an approach able to model heterogeneous aggregate-oriented data containing features like optional properties, versioned entities, union types, and aggregation and referential relationships. The approach should then be able to infer the schema of this data in a format, which should be easy to process further both manually and using existing tools and applications. The approach should be able to handle a large amount of highly heterogeneous data, and should be able to reasonably scale with the dataset size.

3.1 High-level design

Our approach is a vertically and horizontally scalable MDE-based Apache Spark application. It is primarily designed with JSON in mind but can be easily adapted for other aggregate-oriented data formats without any conceptual design changes necessary. Along with the inference approach design, a new *NoSQL Schema* metamodel serves as the primary schema representation format. Refer to figure 3.1 for the high-level diagram of the inference process.

As a precondition of the inference, we assume there to exist a possibly massive, possibly highly heterogeneous aggregate-oriented dataset present in a single database. This data set can consist of multiple top-level entity types, each in its own data collection within the database.

This data is loaded into an Apache Spark instance as a *Resilient Distributed Dataset (RDD)* of documents. Then, structural duplicates, which provide no additional schematic information, are removed from the dataset in a two step process: first, all documents are transformed into raw schemas by replacing primitive values with their type names. Second, duplicate raw schemas are then removed from the RDD.

Afterwards, the remaining, unique, raw schemas are injected into the NoSQL Schema metamodel to create NoSQL Schema models. Lastly, all of the models in the RDD are folded in a non-deterministic order using a commutative and associative binary operator, resulting in a single schema model describing the entire input dataset. This concludes the main part of the inference process which is depicted by solid (non-dashed) arrows and boxes within the high-level diagram, figure 3.1.

The resulting NoSQL Schema model can be optionally further processed in a number of ways which are depicted in the high-level diagram using dashed arrows and boxes. One of these optional operations is extending the schema with new data. Since the binary operator used for folding schema models in the inference process is commutative and associative, an existing schema can be extended with a new dataset by first subjecting the new dataset to the main part of the inference process, then folding the resulting new schema into the existing one using the folding operator.

Another optional schema modification is flattening an entity. As the schema is

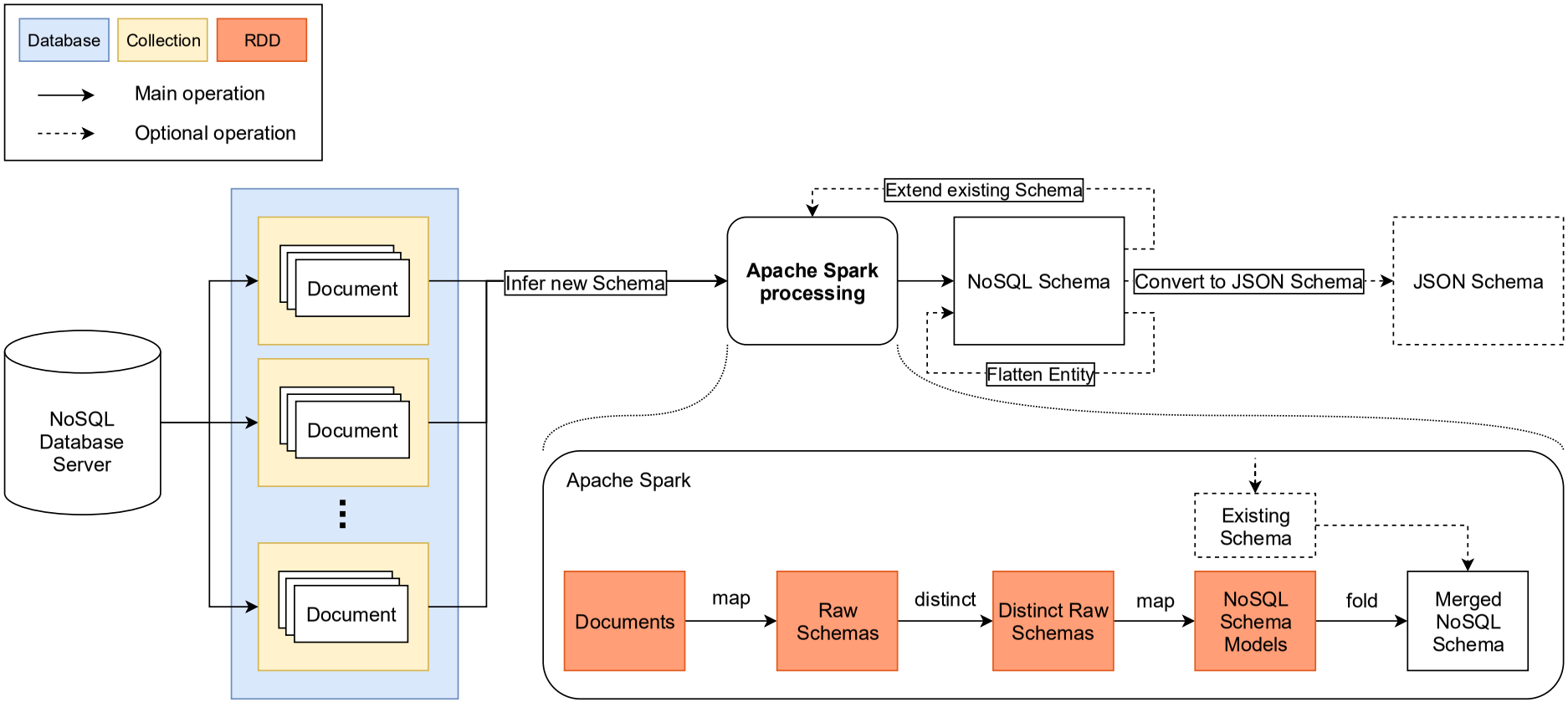


Figure 3.1: High-level diagram of the new inference approach

first inferred, it is *versioned*, meaning that even the most minor structural or type distinction between two input documents results in the creation of two different *entity versions* being modeled. The versions of a single entity can be flattened (merged together), creating a single version, where some properties are possibly optional, and some properties can have union types.

Since our approach focuses mainly on JSON data, we provide as an added benefit a way to convert the NoSQL Schema model into a JSON Schema. The JSON Schema can then be used to generate useful tools, e.g. for data validation, migration, or object-data mapping.

3.2 Inspiration and added value

The approach takes inspiration from the MDE tools and techniques used by Sevilla et al. [8] in their approach. However, unlike their approach, which uses MapReduce framework to reduce the size of the input data in a scalable manner, we use for this purpose the arguably more powerful Apache Spark framework. The benefits of Apache Spark over MapReduce are explained in section 3.3.1. Further, we make improvements over the NoSQL Schema metamodel by Sevilla et al. These are described in more detail in section 3.4.2.

Approach by Baazizi et al. [15] is used as an inspiration for its use of Apache Spark for ensuring horizontal scalability. The main structure of the Spark data pipeline of our approach is very similar to that of the other approach. This is intentional. In this sense, our NoSQL Schema metamodel takes the place of the JSON type system of the Baazizi et al. approach. The operations "*map to model*" and "*fold schemas*" of our Spark pipeline are equivalent to the "*infer type*" and "*reduce types*" operations from the other approach.

The added value lies in the fact that unlike the type system of Baazizi et al. which has documents as the top-level schema element, the top-level element of a NoSQL Schema model is an entity. This allows our approach to infer collective schema of an entire database of interdependent collections and model same-structured nested JSON objects found in different parent entities as one entity type.

3.3 Technology

Let us shortly describe the existing technology used in our inference approach, its purpose and benefits.

3.3.1 Apache Spark

Apache Spark is a distributed data processing framework providing its user with automatic data partitioning and replication, work distribution, checkpointing, and failure detection. It takes inspiration from the MapReduce paradigm and is heavily based on the Apache MapReduce implementation.

Compared to the base MapReduce computation model, Apache Spark provides multiple significant benefits. Spark extends the computation model by

allowing an arbitrary number of steps in the data processing pipeline. These operations' names and contracts are consistent with the collection operations available in Java and Scala: `map` applies a function to each element of the dataset, replacing it with the function value; `distinct` discards duplicate elements from the dataset; `reduce` and `fold` repeatedly apply a binary operator on the dataset, reducing its elements to one; and many more.

Furthermore, Apache Spark achieves much higher performance in the general use case than any MapReduce implementation. In a MapReduce computation, the specification states that the intermediate results of each computation step are written to a Hadoop Distributed File System (HDFS) storage, which writes them to a hard drive. This makes it possible for any other worker node in the cluster to reach that data in the next data shuffling phase. Since hard drives are several orders of magnitude slower than operating memory, this writing and loading takes considerable amount of time. Spark solves this performance problem by keeping the intermediate data purely in distributed memory, instead of a distributed file system.

The flexibility of the Spark computation model and the ability to keep intermediate computation results solely in memory is achieved by introducing a new distributed data storage construct: RDD. RDD is a distributed data collection, meaning that the actual data is spread around the cluster nodes. As an abstraction layer, it provides to the Spark user an opaque handle for a collection, which can be operated on using the aforementioned collection operations. As such, the only responsibility left to the user is to implement the data processing as a series of collection operations, while Spark handles all of the technicalities of the distributed computation: data replication, checkpointing, failure detection, data reshuffling, workload balancing, network usage, etc.

3.4 NoSQL Schema metamodel

Let us describe in detail the newly designed NoSQL Schema metamodel, depicted on figure 3.2, the motivations for its inception and evolution, and the distinguishing factors from a similar solution.

The metamodel is designed to cover aggregate-oriented NoSQL data, with the majority of design decisions being based on JSON, mainly due to its prevalence in the industry. This is not to its hindrance, however, as the metamodel can be easily extended with data types from other formats.

The metamodel is composed of several *classes*. On this layer, we will use this terminology in favor of *entity* to avoid ambiguities in the text, since *Entity* is the name of one of the classes in the metamodel. Each class can contain attributes with simple types, as well as references to other classes in the metamodel.

The *NoSQLSchema* class is the entry point of the metamodel. *NoSQLSchema* instances represent the inferred schemas themselves. For each instance, the `entities` relation leads to the entities contained within the schema and a `name` attribute provides basic context to the schema.

Instances of the *Entity* class represent object classes or entities inferred from the data set. The name of the inferred entity is stored in the `name` attribute. The `versions` relation represents a set of distinct entity versions of this entity that were inferred from the data. In addition, the boolean-typed `flattened`

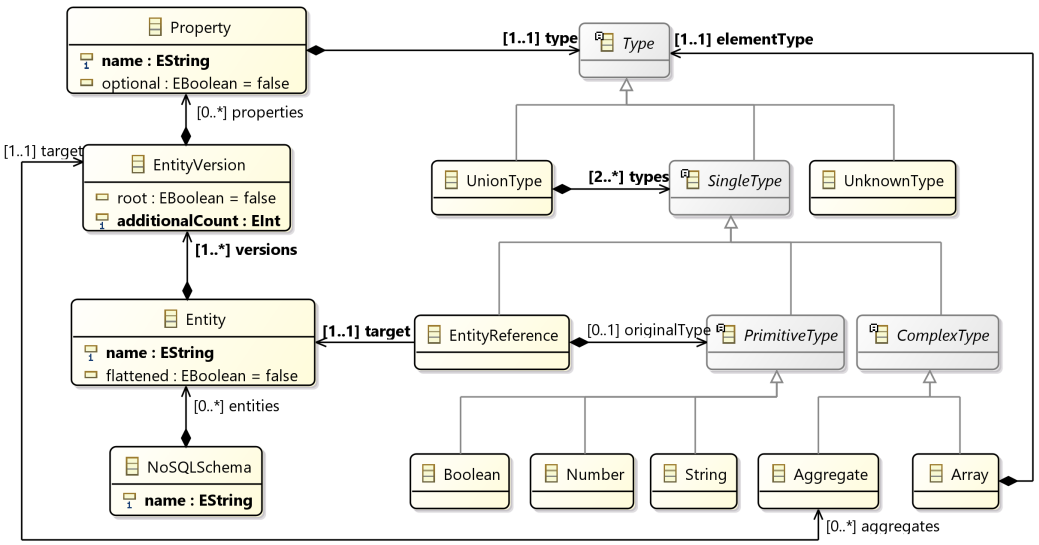


Figure 3.2: NoSQL Schema metamodel

attribute provides information on whether or not the entity is flattened. For more information on *entity flattening*, see section 3.5.7.

The *EntityVersion* class and its instances represent distinct versions of an entity. An entity version is defined by the set of its properties—two entity versions are identical if they belong to the same entity and if their property set is identical, otherwise, they are distinct. An entity version's properties are reachable using the `properties` relation. The `root` attribute contains information on whether or not an instance of this entity version has appeared as the root object (document) in the data set. The `aggregates` relation and the `additionalCount` attribute are explained further below, in the section describing *Aggregate*s.

Instances of the *Property* class represent properties of the entity versions – name-type pairs with `name` being a string-typed attribute and `type` being a relation leading to a *Type* instance. Additionally, whether the property is optional or required can be recorded in the `optional` attribute. The relation of this attribute to the `flattened` attribute of the *Entity* class is better explained in section 3.5.7.

The different property types in the NoSQL Schema metamodel are organized into the type hierarchy. The *Type* abstract class is the root of the hierarchy and it has three direct subclasses: *UnknownType*, *UnionType*, and *SingleType*.

UnknownType is a non-abstract class which, by definition, contains and provides no knowledge about the underlying: A JSON property whose type is inferred as *UnknownType* can contain any value whatsoever. It is the identity element of the type-folding operator (see algorithm 2).

UnionType is a non-abstract class which represents the situation, where the values of a property have been found to contain values of different actual types. These two or more single types are reachable through the relation `types`.

SingleType is an abstract class designed as a logical counterpart to the *UnionType*. Subclasses of *SingleType* represent a situation, where the actual values found within the data were all of the same type. There are three subclasses: *EntityReference*, *PrimitiveType*, and *ComplexType*.

EntityReference represents a data type, which contains the unique identifier of another entity. It is the NoSQL equivalent of the concept of foreign keys in relational DBMS. The referenced entity is reached through the `target` relation, while the actual data type used to refer is reachable as `originalType`.

As it stands, our approach is capable of inferring entity references in two forms:

- references encoded as Extended JSON `DBRef` instances
- references encoded as simple object properties whose name is in the form of `"<entity_name>_id"` or similar, and whose type is either a *PrimitiveType* or an *Array* of *PrimitiveType*s.

However, the NoSQL Schema metamodel is not bound to these methods of entity reference inference, and other heuristic methods can be easily used.

The *PrimitiveType* class is an abstract parent for the three classes representing the three JSON primitive data types: *Boolean*, *Number*, and *String*. The JSON *null* value as it is not a type, rather a meta-value signifying a missing actual value.

The *ComplexType* abstract class is a parent of the two concrete classes representing complex JSON types: *Array* for the JSON array, and *Aggregate* for the JSON object. *Array* contains a relation named `elementType` which encodes the type of the elements of the array. In case of values of different types contained within the array, the `elementType` can be a *UnionType*, where in case of nested arrays, it can be another *Array*.

Finally, the *Aggregate* class represents the *aggregation* relation between inferred entities. In case of JSON, this means objects nested within other objects. An *Aggregate* contains a relation named `target` which is a reference to the *EntityVersion* which represents the aggregated object. This is a bi-directional relation. The other direction is named `aggregates` and provides the *EntityVersion* access to all *Aggregate*s of that version which are present in the schema. To preserve the number of times the entity version was actually present in the data set, the `additionalCount` attribute contains the number of *Aggregate*s of that version which were removed from the schema (merged during type-folding, see algorithm 2). The actual occurrence count of an entity version is then calculated as the sum of the cardinality of its `aggregates` relation and its `additionalCount` attribute.

3.4.1 Metamodel evolution

Since the inception of the NoSQL Schema metamodel, there were significant changes done to it. The initial proposal is depicted in figure 3.3. We describe several key differences and their significance to the overall design.

The `root` attribute for signifying whether an object was present as the root of the document was moved from the *Entity* class to the *EntityVersion* class. This better expresses the property of "being a potential document root" and allows for greater granularity in the decisions done throughout the inference process.

The `id` attribute was removed from the *EntityVersion* class, as it was found to have no use, as the `versions` relation of the *Entity* class is defined to be ordered.

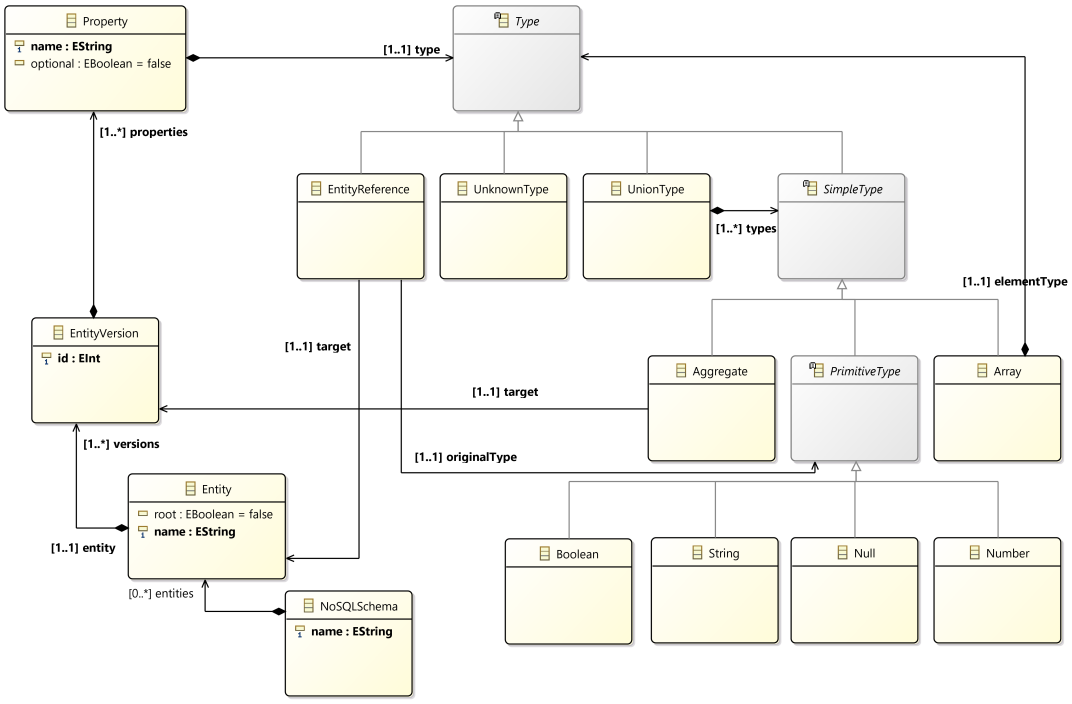


Figure 3.3: Initial prototype of the NoSQL Schema metamodel

The `target` relation from *Aggregate* to *EntityVersion* was made bi-directional to allow for easier traversing of the schema graph. Also, the `additionalCount` attribute was introduced to express the count of aggregates, which were lost in the merging process.

Significantly, *EntityReference* was relocated within the type hierarchy from being a child of *Type* to a child of *SingleType*. This allows it to be included in the list of `types` within a *UnionType*, which we found desirable.

An additional abstract class named *ComplexType* was introduced as a parent of *Array* and *Aggregate* to provide contrast with the *PrimitiveType* class.

Though there were other minor changes made in the design process, they are not crucial.

3.4.2 Differences from the Sevilla et al. NoSQL-Schema metamodel

It is no secret that our NoSQL Schema metamodel is inspired by the metamodel conceived by Sevilla et al. in their related article [8]. Alongside similarities, however, there is a key difference in our metamodel that opens up previously unavailable possibilities.

Starting from the *NoSQLSchema* entry point, there are only minor differences, until one reaches the *Property* class. In Sevilla et al. metamodel, *Property* and related classes form a hierarchy which differentiates between *associations* (aggregations and references) and *attributes*. A given property can be either one or the other, and cannot be both at the same time, e.g. unionized by a *UnionType*. In fact, the Sevilla et al. metamodel does not support the concept of union types at

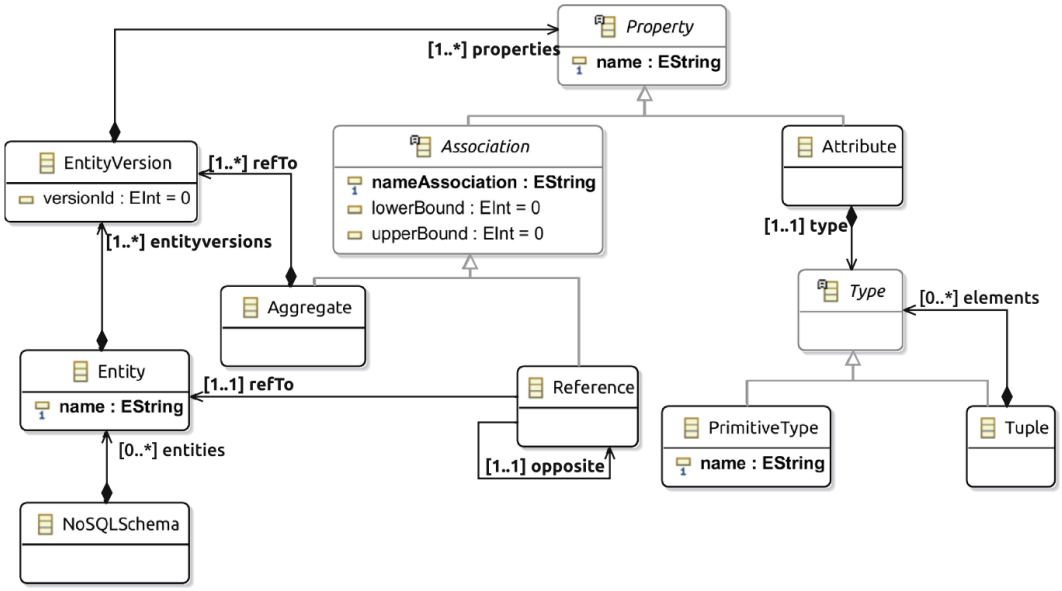


Figure 3.4: Sevilla et al. NoSQL Schema metamodel for comparison

all.

Relocating the *Aggregate* and *Reference* classes to the type hierarchy enables us to fully utilize the potential of union types. It allows us to more closely model complicated JSON data structures with multidimensional arrays (by setting an *Array*'s `elementType` to be another *Array*), arrays containing nested objects (*Aggregate*), and heterogeneous arrays containing values of different types (*UnionType*), e.g. an array containing numbers and objects.

3.5 Detailed design

Here we describe in greater detail the individual steps our inference approach is composed of, their significance, and their technicalities. Descriptions are complemented by pseudo-code where appropriate.

3.5.1 Loading the data

The first step of our schema inference process is loading the data from the database into Apache Spark. We want our approach to be compatible with different JSON and non-JSON database technologies. To achieve this we separate the responsibility of loading the data into Spark to an interface called *DataLoader*. An implementation of this interface compatible with any desired JSON data storage can be easily provided to the inference process. The inference process will then use that implementor to load the dataset into Spark. An implementation for the MongoDB JSON database is provided out-of-the-box.

While we bind our design to being primarily compatible with JSON, we want our inference approach to be usable for other forms of aggregate-oriented data as well. If a user of our inference approach needs to infer the schema of another type of database, like key-value (Riak KV, Redis, etc.), or columnar (Cassandra),

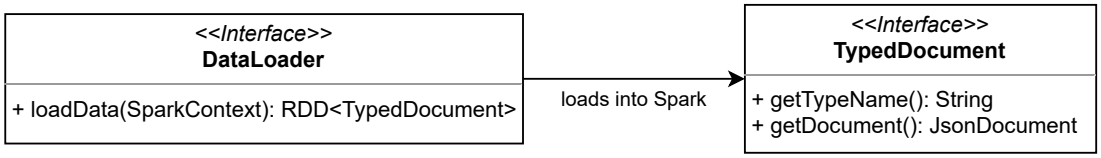


Figure 3.5: *DataLoader* and its relation to the *TypedDocument*

they can provide a *DataLoader* implementation which loads the data from that database, and converts it in a reasonable way into JSON documents. E.g. each row in a Cassandra table can be converted into a simple JSON document by taking each (column name, value) pair and transforming it into a property pair in the document. In case the column value is itself a JSON document, that document can be transformed into a nested object within the resulting document.

3.5.1.1 Data loader interface

The *DataLoader* interface is visualized in figure 3.5 along with its relation to the *TypedDocument* interface. It has one abstract method:

- `loadData(SparkContext)`, which loads the data from the database into Spark as an RDD of *TypedDocument*s (`RDD<TypedDocument>`) using the provided `SparkContext`.

TypedDocument is an interface providing access to a JSON document and the name of its type. This name can be obtained from the name of the collection or other collection-like structure in the database, e.g., table or bucket.

A *DataLoader* may need to perform costly transformations on the dataset before returning it, especially in the case of a non-JSON underlying data storage. Since the *DataLoader* has access to the `SparkContext` of an active Spark session, all of these transformations can be performed within Spark as well, which preserves the scalability of the approach.

3.5.2 Removing structural duplicates

After loading the data, the next step is removal of structural duplicates. This is done, as previously mentioned, by first transforming the *TypedDocument*s into *typed raw schemas*, then removing duplicate schemas from the data collection.

A typed raw schema is composed of two components, similar to a *TypedDocument*: a type name and a raw schema. The type name is taken unchanged from the *TypedDocument*. The other component, the raw schema of a JSON document, is created by traversing the structure of the document recursively, replacing all primitive values in the document with a JSON string, containing the type name of the primitive value.

This process keeps the structural shape of the document mostly intact. The only exception are Extended JSON `DBRef` instances, which are encoded into plain JSON as objects with special properties, similar to other Extended JSON types. These are converted to string representation `"reference(<entity_name>)"`. E.g.

```
{ "_id": "number", "timestamp": "string",
  "author": {
    "first_name": "string",
    "last_name": "string",
    "phone_number": "string",
    "location": {
      "latitude": "string",
      "longitude": "string" } },
  "ratings": [ "number", "number", "number", "number", "number" ],
  "comments": [ "string", "string" ],
  "attachments": [
    { "url": "string" },
    { "url": "string" } ],
  "body": {
    "content": "string",
    "mime_type": "string" },
  "published": "boolean" }
```

```
{ "_id": "number", "body": "string",
  "timestamp": "string",
  "author": {
    "first_name": "string",
    "last_name": "string",
    "phone_number": "number",
    "location": {
      "address": "string" } },
  "ratings": [ "number", "number", "number" ],
  "comments": "string",
  "article_id": "number",
  "published": "boolean" }
```

Figure 3.6: Raw schemas for the JSON documents from the running example

a `DBRef` object `{$ref:"articles",$id:"abcde"}` will be transformed to `"reference(articles)"`. For illustration, figure 3.6 showcases the raw schemas for the running example.

After each *TypedDocument* in the RDD is transformed into a typed raw schema, duplicate raw schemas are eliminated. When comparing two typed raw schemas for equality, both the type name, and the raw schema is taken into consideration. However, the order of properties within the document and nested objects is not.

3.5.3 Injection into model

Afterwards, the typed raw schemas are injected into the NoSQL Schema metamodel, creating a model for each of those typed raw schemas. The recursive IN-

JECT function from algorithm 1 is applied to each of the schemas using a `map()` command in Apache Spark.

One needs to keep in mind that the input JSON document is a raw schema of the actual document, not the original itself, therefore there are no primitive values except for strings, and those strings carry the type information.

There is a number of functions left unimplemented in the algorithm pseudocode because we believe their implementation to be trivial, e.g. GETORADDENTITYWITHNAME or GETPROPERTIES. However we would like to clarify an important detail of the function GETORADDIDENTICALVERSION called on line 33—when comparing entity versions for identity, their property names and the properties' types are considered, but the version aggregates and additional occurrence counts are not.

The FOLDTYPES function is extracted into a separate algorithm definition. This is intentional, as this function is also used at a later stage, during the entity flattening process, section 3.5.7.

Important to note is that the injection step along with the previous raw schema conversion does not detect all entity references. Only references encoded as Extended JSON `DBRef` objects are detected, the remaining simple-property references are instead left as their original primitive types and are detected later. The reason for this distinction is that in this stage of schema inference, only partial information about the names of entity types is available. Since we cross-match the property names with entity names during inference of primitive-typed entity references, all entity type names from the data must already have been detected to correctly find reference targets. Because of this, the inference of this type of references is deferred until after the entire model is merged together.

3.5.4 Folding the models

Definition 9. *Property-dependent difference of sets of entity versions (denoted by $\setminus_V$) is a binary operator on sets of entity versions. Given two sets of entity versions V_1 and V_2:*

$$V_1 \setminus_V V_2 = \{v | v \in V_1 \wedge \neg(\exists v')(v' \in V_2 \wedge \textit{PROPERTIES}(v') = \textit{PROPERTIES}(v))\}$$

Definition 10. *Property-dependent intersection of sets of entity versions (denoted by $\cap_V$) is a binary operator on sets of entity versions. Given two sets of entity versions V_1 and V_2, $V_1 \cap_V V_2$ contains only elements decided by the following rule:*

For each entity version v_1 in V_1, if V_2 contains an entity version v_2 such that $\textit{PROPERTIES}(v_1) = \textit{PROPERTIES}(v_2)$, then $V_1 \cap_V V_2$ contains an entity version having:

- *properties* $= \textit{PROPERTIES}(v_1)$
- *root* $= \textit{ROOT}(v_1) \vee \textit{ROOT}(v_2)$
- *aggregates* $= \textit{AGGREGATES}(v_1) \cup \textit{AGGREGATES}(v_2)$
- *additionalCount* $= \textit{ADDITIONALCOUNT}(v_1) + \textit{ADDITIONALCOUNT}(v_2)$

Algorithm 1 Injection of typed document into NoSQL Schema metamodel

```
1: function INJECT(s, j)                          ▷ s is the type name, j is the JSON value
2:     E ← ∅                                                    ▷ Set of detected entities
3:     CONSTRUCTTYPE(s, j, E, true)
4:     return NOSQLSCHEMA(E)
5: end function
6: function CONSTRUCTTYPE(s, j, E, r)  ▷ r - whether or not j is a root value
7:     if j = "null" then
8:         return UNKNOWNTYPE
9:     else if j = "boolean" then
10:        return BOOLEANTYPE
11:    else if j = "number" then
12:        return NUMBERTYPE
13:    else if j = "string" then
14:        return STRINGTYPE
15:    else if j = "reference(n)" then
16:        return ENTITYREFERENCE(n)
17:    else if j is an array then
18:        T ← UNKNOWNTYPE                                              ▷ Element type
19:        for all e ∈ j do                          ▷ Iterate through array elements
20:            T_e ← CONSTRUCTTYPE(s, e, E, false)                      ▷ Recursive call
21:            T ← FOLDTYPES(T, T_e)
22:        end for
23:        return ARRAYTYPE(T)
24:    else                                                             ▷ j is an object
25:        e ← GETORADDENTITYWITHNAME(E, s)
26:        P ← EMPTY ASSOCIATIVE ARRAY                                  ▷ Property types
27:        for all (s′, j′) ∈ GETPROPERTIES(j) do
28:            P[s′] ← CONSTRUCTTYPE(s′, j′, E, false)                  ▷ Recursive call
29:        end for
30:        v ← GETORADDIDENTICALVERSION(e, P)
31:        if r then
32:            ISROOT(v) ← true
33:        end if
34:        INCREMENTOCCURRENCECOUNT(v)
35:        return AGGREGATETYPE(v)
36:    end if
37: end function
```

Algorithm 2 Folding two types into one

```
1: function foldTypes(t_1, t_2)
2:     T_1 ← wrap(t_1)
3:     T_2 ← wrap(t_2)
4:     return unwrap(T_1 ∪ T_2)
5: end function
6: function wrap(t)                          ▷ wraps type in a set
7:     if t = UnknownType then
8:         return ∅
9:     else if t is a single type then
10:        return {t}
11:    else                                  ▷ t is a union type
12:        return getSingleTypes(t)
13:    end if
14: end function
15: function unwrap(T)                       ▷ unwraps set into a type
16:    if T = ∅ then
17:        return UnknownType
18:    else if t is a singleton then
19:        return single element of T
20:    else
21:        return UnionType(T)
22:    end if
23: end function
```

In this step, all the partial NoSQL Schema models are folded (merged) together into one final model by repeated application of the binary folding operator. The folding operator, foldSchemas function, is defined using pseudo-code in algorithm 3.

In addition to the folding operator, a zero-value element is supplied to the fold operation as an argument. This can either be an empty schema—an identity element for the folding operator—or an existing schema can be passed. If an existing, non-empty schema is used, then the schema is effectively enriched with the schematic information of the new dataset. This is used in the optional *Schema extension* step, which is described in section 3.5.6.

In Apache Spark, the `fold()` command only roughly resembles the *fold* higher-order function we know from functional programming, as the Spark specification provides not guarantees to the order of operations. For maximum efficiency, the fold is usually first performed within each of the Spark runners on the elements present within the runner, then the intermediate results are sent to the Spark master or another runner to be folded with one another, yielding the final answer, but all of this is subject to implementation detail.

If the folding operator is commutative and associative, then for any set of NoSQL Schema models with cardinality n, applying the operator a total of $n-1$ times, each time reducing the cardinality by 1, will reach the same final result, regardless of the order of operations. Therefore, the operation, and the whole algorithm is only sound as long as the aforementioned properties on the folding operator are maintained.

Algorithm 3 Folding two NoSQL Schema models into one

function FOLDSCHEMAS(M_1, M_2) ▷ Two different NoSQL Schema models
  $E_1 \leftarrow$ GETENTITIES(M_1)
  $E_2 \leftarrow$ GETENTITIES(M_2)
  $N_1 \leftarrow$ GETNAMES(E_1)
  $N_2 \leftarrow$ GETNAMES(E_2)
  $E \leftarrow$ GETENTITIESBYNAMES($E_1, N_1 \setminus N_2$)
  $E \leftarrow E \cup$ GETENTITIESBYNAMES($E_2, N_2 \setminus N_1$)
  for all $s \in N_1 \cap N_2$ **do**
    $E \leftarrow E \cup \{$FOLDENTITIES($E_1[s], E_2[s]$)$\}$
  end for
  return NOSQLSCHEMA(E)
end function
function FOLDENTITIES(e_1, e_2)
  $V_1 \leftarrow$ GETVERSIONS(e_1)
  $V_2 \leftarrow$ GETVERSIONS(e_2)
  return ENTITY($(V_1 \setminus_V V_2) \cup (V_2 \setminus_V V_1) \cup (V_1 \cap_V V_2)$)
end function

Theorem 3.1. *The FOLDSCHEMAS function is commutative.*

Theorem 3.2. *The FOLDSCHEMAS function is associative.*

A proof for theorem 3.1 can be found in attachment D. We assume theorem 3.2 to be true despite it not being formally proven. This is due to overwhelming evidence during manual algorithm testing. This property is also later covered by unit tests in the implementation.

3.5.5 Entity reference inference

Only after the partial models are all folded together completely, the primitive-typed entity references can be inferred. This can be considered as the finalization step for the main part of the schema inference. This step is described in algorithm 4.

3.5.6 Schema extension

Since the schema folding binary operator is commutative and associative, existing schema can be easily extended using additional batch of data. To achieve this, the new dataset processed as if a new schema was being inferred, up until the schema-folding step—the existing schema is provided as a zero-value element to the `fold()` command in Spark. Then the resulting folded schema is finalized by the *Entity reference inference* step. The final schema contains schematic information from the original schema, as well as the new dataset.

3.5.7 Entity flattening

Each entity within our NoSQL Schema model is either *versioned* or it is *flattened* (also called *flat*), as recorded by the `flattened` attribute. When an entity is first

Algorithm 4 Inferring entity references within NoSQL Schema

```
1: function inferEntities(M)
2:     E ← getEntities(M)
3:     for all e ∈ E do
4:         for all v ∈ getVersions(e) do
5:             for all (s, t) ∈ getProperties(v) do ▷ Property name and type
6:                 t_n ← getEntityReferenceType(s, t, E)
7:                 if t_n ≠ ∅ then
8:                     setPropertyType(v, s, t_n)
9:                 end if
10:            end for
11:        end for
12:    end for
13:    return M
14: end function
15: function getEntityReferenceType(s, t, E)
16:    if t is a primitive type then
17:        if s matches regular expression "⟨entity⟩(_id|Id)" then
18:            e ← getEntityByName(⟨entity⟩, E)
19:            return EntityReference(e, t)
20:        end if
21:    else if t is an array of primitive type t_0 then
22:        if s matches regular expression "⟨entity⟩(_id|Id)s" then
23:            e ← getEntityByName(⟨entity⟩, E)
24:            r ← EntityReference(e, t_0)
25:            return ArrayType(r)
26:        end if
27:    end if
28:    return ∅
29: end function
```

created during the *Injection* step, it is versioned. It can be flattened by the user on demand with the *Flatten entity* operation.

A versioned entity contains one or more entity versions. For each of those entity versions, for each of that version's properties, the following statements hold:

- that property is *not* optional, and
- that property's type is *not* a union type.

These invariants come from the way how entity versions are created during the *Injection* step: When considering a document, an existing version is used only if all of the properties' types match, otherwise a new version is created instead of using union types. Furthermore, all properties are created as *required*, not *optional*.

The user can choose to flatten an entity on demand—merge all of its versions into one, by applying a combining binary operator. This process is described by pseudo-code in algorithm 5. The algorithm references method foldTypes on line 24, whose pseudo-code is located in algorithm 2.

Algorithm 5 Flattening an entity

```
 1: function flattenEntity(e)
 2:     V ← getVersions(e)
 3:     return foldLeft(V, foldVersions)
 4: end function
 5: function foldVersions(v1, v2)
 6:     isRoot(v1) ← isRoot(v1) ∨ isRoot(v2)
 7:     aggregates(v1) ← aggregates(v1) ∪ aggregates(v2)
 8:     additional(v1) ← additional(v1) + additional(v2)
 9:     for all n ∈ getPropertyNames(v1) \ getPropertyNames(v2) do
10:         p ← getPropertyByName(n, v1)
11:         isOptional(p) ← true
12:     end for
13:     for all n ∈ getPropertyNames(v2) \ getPropertyNames(v1) do
14:         p ← getPropertyByName(n, v2)
15:         isOptional(p) ← true
16:         addProperty(p, v1)
17:     end for
18:     for all n ∈ getPropertyNames(v1) ∩ getPropertyNames(v2) do
19:         p1 ← getPropertyByName(n, v1)
20:         p2 ← getPropertyByName(n, v2)
21:         t1 ← getType(p1)
22:         t2 ← getType(p2)
23:         type(p1) ← foldTypes(t1, t2)
24:     end for
25:     return v1
26: end function
```

The resulting flat entity's version has the following properties:

- the set of names of its properties is a union of the sets of names of the properties of all of the original versions,
- for each of its properties:
  - that property is required if and only if a property with the same name is present in each of the original versions, otherwise it is optional, and
  - that property's type is created by folding the types of properties with the same name present in the original versions—resulting in either a single type, if all of the original properties were of the same type, or a union type otherwise.

3.5.7.1 Flattening entity containing root and non-root versions

In the initial—versioned—state, an *Entity* object of the NoSQL Schema model can contain both root (`root=true`) and non-root (`root=false`) entity versions. This can happen when a name of a document collection (and therefore a document's type name) coincides with the name of a property containing a nested object. Both the entity versions inferred from the documents of that collection, and the entity versions inferred from those nested objects, would belong to the same entity. Flattening that entity would merge together root and non-root entity versions, which might cause a majority or all of the properties to be marked as optional. This behavior may be unexpected by the user, but it is correct by design. Let's take a closer look.

For the sake of illustrating this behavior, let us take the established running example and expand upon it. The running example is composed of two JSON documents contained within the *articles* collection. These documents represent instances of the *Article* entity type. Since they are at the top-level, they contain all or close to all of the information related to the objects they represent, resulting in a high number of properties.

In a real-world application, however, JSON objects representing *Article*s might also be included elsewhere, e.g. as nested objects within documents representing other types. Suppose that alongside the *articles* collection, in the same database there exists collection *article_groups*. Suppose that each document within that collection contains, among others, a property named `articles`, whose value is an array of objects—an enumeration of all articles contained within that group.

Although these nested objects represent instances of the same type as the top-level documents in the *articles* collection, they would usually contain only some information about those articles. The reason for this is that in the use-case of retrieving all relevant data about an article group, complete details about the contained articles would not be necessary to include. Only the identifier (`_id`) and possibly some additional information (e.g. `timestamp`) would usually suffice, and the requester could use the identifier to retrieve the complete object with the remaining data from the *articles* collection if necessary.

Therefore, let's suppose that the nested objects within the `articles` array contain only the `_id` property. Figure 3.7 contains an example JSON document from the supposed *article_groups* collection.

If we subject this extended example (the *articles* and *article_groups* collection) to our inference approach, the *Article* entity will contain three entity versions:

```
{ "_id": 1,
  ...,
  "articles": [ { "_id": 1 }, { "_id": 18 }, { "_id": 347 } ],
  ... }
```

Figure 3.7: An example JSON document from the *article_groups* collection

- Two will be inferred from the documents within the *articles* collection. These versions will be marked as `root=true`.
- The last version will be inferred from the nested objects within the `articles` array in the *article_groups* collection. This version will be marked as `root=false`.

Should the user then flatten the *Article* entity, the resulting entity's version would have all properties other than `_id` marked as optional. The reason for this is that one of the versions that will have been merged into the others—the non-root one—will have contained only the `_id` property.

3.5.8 Conversion to JSON Schema

The NoSQL Schema model, along with the options to extend it with further data, and to flatten inferred entities, is the final product of the main part of the inference process. However, to tie the NoSQL Schema model and the inference approach back to the JSON data model, we provide an out-of-the-box way to convert a NoSQL Schema model to a JSON Schema.

JSON Schema, despite being still a draft specification, has an expansive environment of libraries and tools for data validation, data migration, code generation, web user interface generation, and other utilities.

When a JSON document's schema is inferred using our approach, the hierarchical structure of the document is converted into a flat one, where the top-level elements are unique inferred entities. Since a JSON Schema's structure is hierarchical, similar to a JSON document, the hierarchical structure needs to be recreated during the conversion.

For the purposes of this conversion, let us simplify the NoSQL Schema model to a *version graph*, whose nodes are entity versions, and whose edges represent aggregation relationship between two entity versions. More formally, for a NoSQL Schema model M, its version graph is $G = (V, E)$, where V is the set of all entity versions of all entities within M, and E contains an edge $(v_1, v_2) \in V \times V$ for each *Aggregate* whose `target` is v_2 and which is contained (directly or indirectly through a chain of *UnionTypes* and/or *Arrays*) in a *Property* of v_1. For illustration, figure 3.8 contains the version graph for the running example NoSQL Schema model.

The conversion process starts by the user specifying a root entity version or entity. If an entity version is specified, it is the sole element in the set of starting points of the conversion. If an entity is specified, the set of starting points is a subset of that entity's versions, determined by whether or not that version has

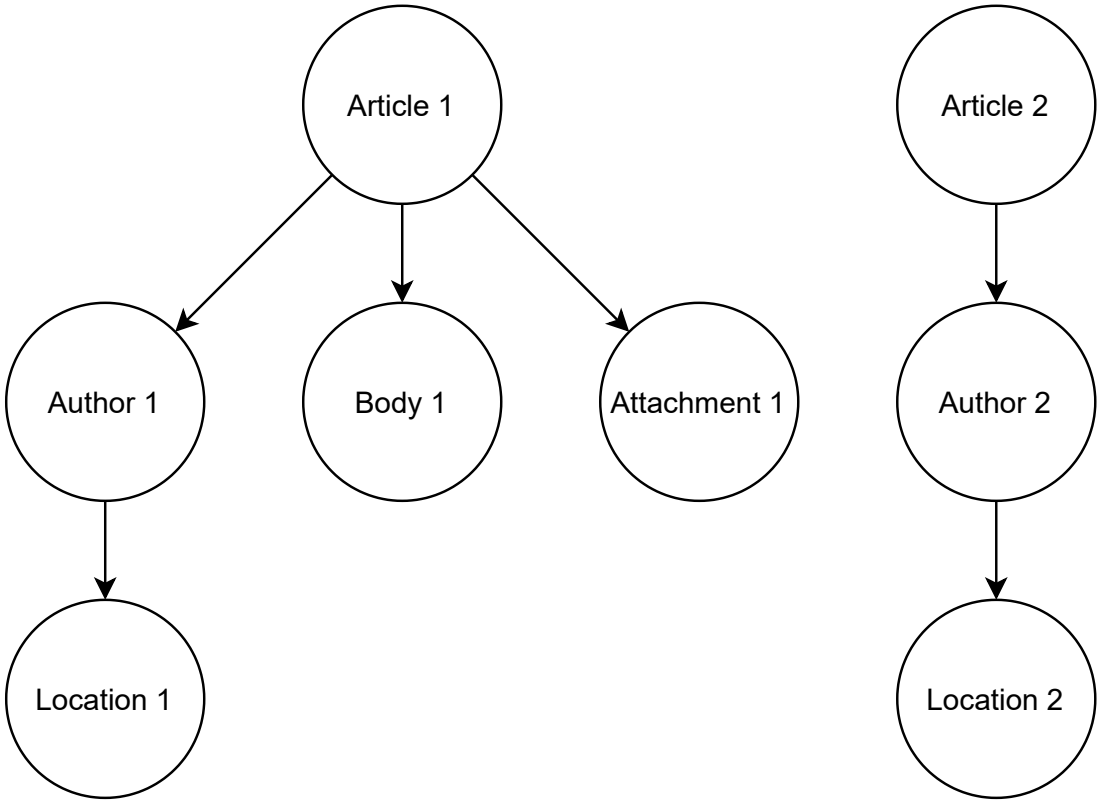


Figure 3.8: Version graph for the running example

been observed in the dataset to be the root of a JSON document, as recorded by the `root` attribute.

The conversion process then performs a search through the graph, starting with the set of starting points, and collects all entity versions visited. Each visited version is then converted into its JSON Schema representation.

An entity version produces the following JSON Schema definition:

- an `$id` keyword set to a value that uniquely identifies this entity version in the generated JSON Schema, e.g. `"article.1"`,
- a `type` keyword set to `"object"`,
- an `additionalProperties` keyword set to `false` – this disallows any other properties than the ones the inference process has detected,
- a `required` keyword containing names of this version's required properties – all other properties are optional
- a `properties` keyword containing the property type definitions

The type definitions for properties are constructed using the following JSON Schema constructs:

- an empty schema (allows everything) for the *UnknownType*,
- `"type":"boolean"` for the *Boolean* type,

- `"type":"number"` for the *Number* type,
- `"type":"string"` for the *String* type,
- for the *Array* type, `"type":"array"` with the `items` keyword containing the schema for the array's `elementType`,
- for the *UnionType*, an `anyOf` keyword containing an array of schemas describing the contained types.

The version definitions—one for each version used in the JSON Schema—are then placed in the `definitions` section of the JSON Schema. The definitions do not recursively contain the definitions of other versions within themselves, rather they link to them using the `$ref` JSON Schema keyword. The root of the schema then contains a reference (again using the `$ref` keyword) to the root entity version or versions, wrapped in an `anyOf` keyword if there are more than one. To better understand the

For illustration, the attachment C.3 contains the converted JSON Schema generated from the running example.

4. Implementation

In this chapter, we present the implementation of the new schema inference approach, its *Application Programming Interface (API)*, and a simple *example application* using the API. Then we describe in detail the implementation of the NoSQL Schema metamodel, the inference approach itself, and the example application. We explain the individual components they consist of, the interfaces those components use to communicate, and the frameworks and libraries they depend on. Later, we describe the unit tests and integration tests used to verify the correctness of the implementation. Lastly, we present the result of running the new inference approach against the running example presented in section 2.1.5.

4.1 Overview

The entire result of the implementation effort is located in a single Git repository, hosted publicly on GitHub[1]. In the repository, we use the Gradle build tool to organize the individual implemented projects and components.

Gradle[2] is a software build tool written in Java. It is based on the Maven build tool and integrates natively with the Maven specification of software artifacts and repositories. Compared to Maven, however, it has a much wider variety of features and is under active development. It is most commonly used for building software written in Java and other Java Virtual Machine (JVM) languages, but can be used for building C++, Python, or others, using its sprawling ecosystem of plugins.

The repository hosts a Gradle multi-project build in its root. A non-exhaustive diagram of the multi-project setup folder structure is shown in figure 4.1. In the diagram, folders containing Gradle projects have blue color, the others have gray color. The first line under each folder is the folder's name. Project folders have an additional line containing the *project path*. The root project's project path is a single colon (“`:`”). The project path is a Gradle project's unique identifier within a multi-project build and we will use it to identify projects also in the text.

The `:impl` project contains the actual implementation of the new schema inference approach. The implementation is made accessible to consumers through an API written in Java.

The root project (“`:`”) contains an example application for the new inference approach. It uses the API within the `:impl` project to infer the schema of a dataset contained within a single MongoDB database, then saves the inferred schema in the NoSQL Schema model format locally to a file.

Additionally, there are four projects located within the `approaches` folder, each containing an implementation of a preexisting schema inference approach. The implementations were provided by their authors and modified for interoperation with our repository structure.

[1] `https://github.com/ivan-lattak/schema-inference`
[2] `https://gradle.org/`

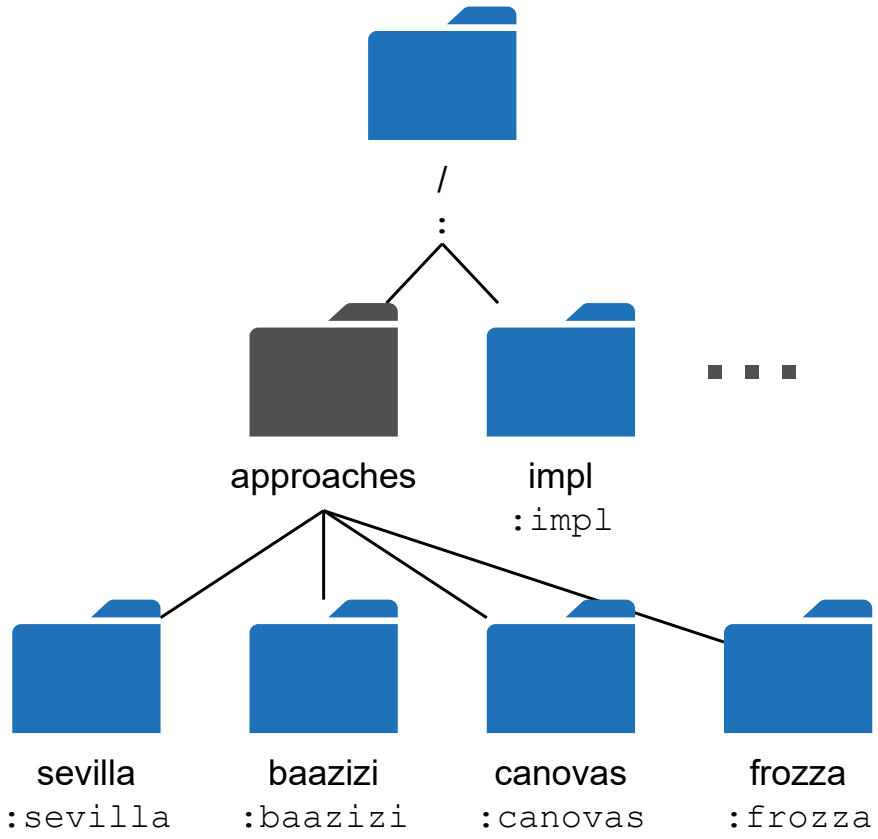


Figure 4.1: A diagram of the repository project structure

Notation:
In this chapter, we use the following path notation for expressing the path to a file or a directory in the repository.

The path to a file or directory is given by "⟨*project*⟩/⟨*file*⟩", where ⟨*project*⟩ is the project path of the file's or directory's nearest enclosing project, and ⟨*file*⟩ is the relative path to that file or directory within that project.

Examples:

- `:impl/model` – The `model` directory within the `:impl` project, translates to path `/impl/model`
- `:sevilla/mapreduce/v2` – The `mapreduce/v2` directory within the `:sevilla` project, translates to path `/approaches/sevilla/mapreduce/v2`

4.2 New approach implementation

The new approach implementation is situated in the `:impl` project. The `:impl/model` directory contains files related to the NoSQL Schema metamodel definition, while the implementation source code is located in `:impl/src`.

4.2.1 Metamodel definition

We used the EMF Eclipse plugin for defining the metamodel. This plugin provides useful features, such as the Ecore master metamodel for defining custom metamodels, graphical model and diagram editors, and a model code generator. The plugin's editors can be used to inspect and edit the definition files.

The metamodel definition comprises three files:

- `:impl/model/nosqlschema.ecore` – contains the definition of the NoSQL Schema metamodel—the file itself is serialization of a model, which is an instance of the EMF Ecore metamodel.
- `:impl/model/nosqlschema.aird` – contains definition of diagrams visualizing the metamodel. In our case, the sole diagram contained is displayed in figure 3.2.
- `:impl/model/nosqlschema.genmodel` – contains the presets for the model code generator.

4.2.2 Implementation source code

It is composed of three source sets: `main` which contains the core of the implementation code and the API, `model` which contains the generated code implementation for the NoSQL Schema metamodel, and `test` which contains the unit tests and integration tests. Let us focus on the `main` source set, where the majority of implementation is—the `model` source set contains only generated code, and we cover the `test` source set in section 4.4.

4.2.2.1 External API

The external API of the `:impl` project consists of the following java packages:

- In the `main` source set:
  - `cz.cuni.mff.ksi.nosql.s13e.impl` – containing the `SchemaInference` class, and the `DataLoader` and `TypedDocument` interfaces.
  - `cz.cuni.mff.ksi.nosql.s13e.impl.inference.mongo` – containing the `MongoDataLoader` class – a `DataLoader` implementation for the MongoDB database system.
- In the `model` source set:
  - `cz.cuni.mff.ksi.nosql.s13e.impl.NoSQLSchema` – containing generated code for the NoSQL Schema model. This code was generated from the metamodel definition using the EMF code generator.

The `cz.cuni.mff.ksi.nosql.s13e.impl` package contains the `SchemaInference` uninstantiable class which is the implementation's entry point. The class's methods can be used to launch schema inference to retrieve the schema as a NoSQL Schema model, extend an existing schema with additional data, save

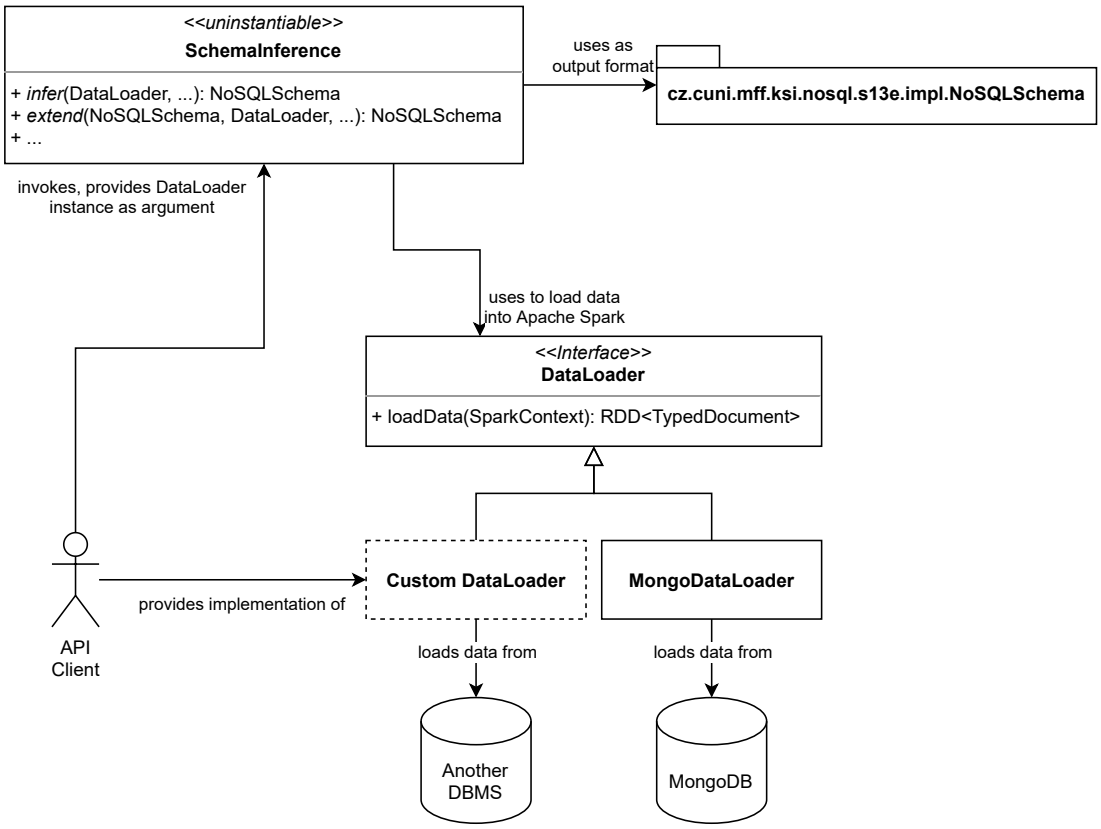


Figure 4.2: Public API diagram of the `:impl` project

and load a schema to/from a file, flatten an entity inside the schema, or convert a schema to the JSON Schema format.

Alongside that, the `DataLoader` and `TypedDocument` interfaces are located in this package. The `DataLoader` interface abstracts away the specifics of retrieving a dataset from a database, from an HDFS disk, from a local disk or from the memory. The `SchemaInference` class's `infer` and `extend` methods take a `DataLoader` parameter and delegate the data loading to it.

A `DataLoader` implementation for MongoDB is provided out of the box. To infer the schema of data located in another DBMS, or data in another format entirely, a user can provide their own `DataLoader` implementation that facilitates the retrieval. This is visualized in the diagram in figure 4.2.

4.2.2.2 Internals

The `cz.cuni.mff.ksi.nosql.s13e.impl.inference` package contains the implementation itself.

Language of choice. Scala, a functional JVM language, was chosen as the main language of the implementation. It was chosen for several reasons.

Firstly, the implementation works closely with Apache Spark. Spark's computational model works well together with the functional paradigm, as functions need to be passed as first-class objects into Spark methods. Although the Java Development Kit (JDK) package `java.util.function` introduced in Java 1.8

provides tools for the functional paradigm, Scala, being a functional language first and foremost, still keeps its edge. Additionally, and for these very reasons, Spark's JVM API is designed to work best with Scala first.

Secondly, the preferred and industry-standard way of writing conditional code in Scala is by using pattern matching (match-case) constructs instead of basic conditional (if-else) constructs. Judging from the pseudo-code algorithm descriptions in chapter 3, it was obvious that the implementation code was going to contain a great number of branching of control flow based on the actual type of a given variable (the `instanceof` keyword in Java). Scala's pattern-matching paradigm made it a great fit for the implementation of the algorithms.

Internal implementation components. Let us take a close look at the implementation of the main part of the inference approach. Following are abbreviated contents of method `createInternal` in object `SchemaInferenceImpl`, lines 37 through 47:

```
37:    val session = SparkSession.builder()
       ...
41:    dataLoader.loadData(session.sparkContext)
42:      .map(TypedDocumentImpl.apply)
43:      .map(_.getRawSchema)
44:      .distinct()
45:      .map(Injector)
46:      .fold(baseSchema)(SchemaFolder)
47:      .named(schemaName)
```

This method's contents nicely fit to the step-by-step design specification of the main part of the inference approach. First, the preliminary. On lines 37 through 40, a `SparkSession` object is initialized using a builder pattern, starting an Apache Spark session.

On line 41, the user-provided `dataLoader` is used to load the dataset from the appropriate data source into Spark as an RDD. The `dataLoader` uses the `SparkContext` argument to create the RDDs. The return type of this call is `RDD[TypedDocument]`.

Line 42 contains the transformation of the user-provided typed documents into an internal type `TypedDocumentImpl` for the purposes of further processing. Then, on line 43, the input documents are transformed into their raw schemas, and on line 44, the duplicate raw schemas are discarded.

Next, line 45 effects the injection of the unique raw schemas into the NoSQL Schema model and its return type is `RDD[InternalNoSqlSchema]`. Afterwards, line 46 folds the schemas along with a `baseSchema` into a single `InternalNoSqlSchema`. The `baseSchema` can be either an empty model, or a schema being extended. Finally, line 47 assigns a name to the folded schema.

For easier visualization, a diagram of components relevant to this process is displayed as figure 4.3.

Generated model code and serializability. In the early stages of the implementation, a problem arose between the model code generated by the EMF code generator and Apache Spark. Since Spark performs computations in a distributed

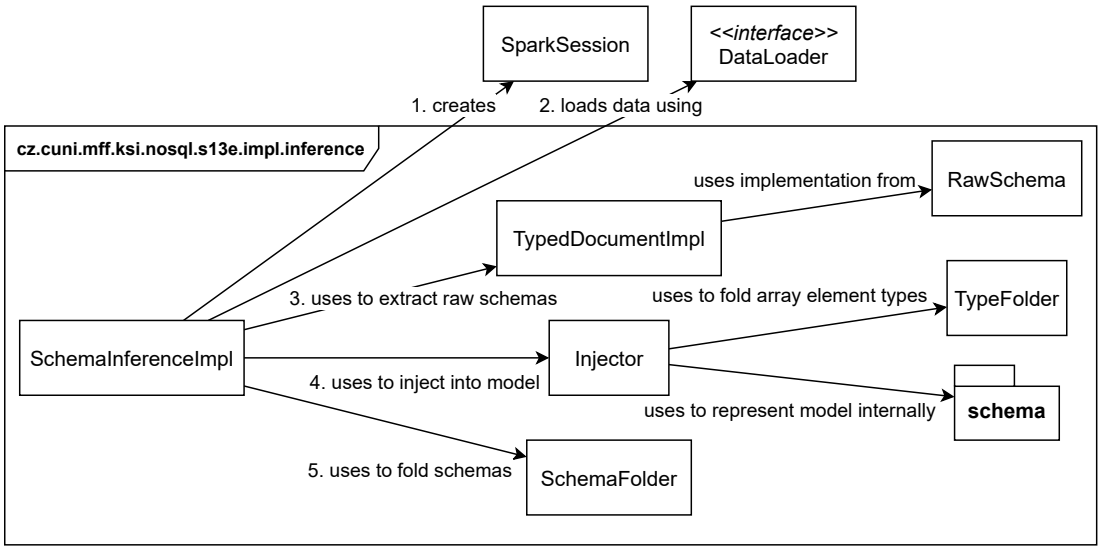


Figure 4.3: A class diagram of classes relevant to the implementation of the main part of the inference approach

manner, data which is being operated with must be sent between individual runners. To do this, Spark uses the default JVM serialization.

The general contract is that any class whose instances are wrapped within a Spark RDD must implement interface `java.io.Serializable` and must otherwise conform to the JVM *serializable* specification. However, the code generator was not designed with serializability of the generated classes in mind.

There were two possible solutions considered for this problem. The first one was to make the generated model code serializable according to the specification. The first step towards this solution was to make all generated classes implement `java.io.Serializable`. This was not difficult, as the code generator specification file, `:impl/model/nosqlschema.genmodel`, contains a `rootExtendsInterface` setting, which can be set to include the `java.io.Serializable` interface. This would cause all the model object interfaces in the `cz.cuni.mff.ksi.nosql.s13e.impl.NoSQLSchema` package extend the required interface.

However, this would solve only part of the problem. The documentation for the `java.io.Serializable` interface, JDK version 1.8, states the following:

"Classes that do not implement this interface will not have any of their state serialized or deserialized. ... To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. ... During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class." [40]

The superclass for the model objects used by the code generator is not serializable and contains private fields. Therefore, these fields would not be serialized, and the serializable subclass cannot take responsibility for their initialization, as it cannot access them. The fields could be accessed using Java reflection but this would be breaking encapsulation of the superclass. While it would be quicker to implement, it would be what is usually called in the industry "a hack" and could become rather expensive down the line by causing difficult to detect runtime is-

sues when updating to a new version of EMF or switching to a different model object superclass.

The second solution was to create a serializable mirror implementation of the model objects, and a utility, which would convert between the two implementations. This solution would be more time-consuming upfront, but would keep best Object-Oriented Programming (OOP) practices and so would be generally better sustainable. For those reasons, the second solution was chosen.

4.2.3 Implementation limitations

There are several limitations to the approach which were discovered during the implementation phase. While are inclined to believe that they are merely limitations caused by the current implementation—not inherent limitations of the approach design—and so could be resolved, further research would be required to prove or disprove this hypothesis.

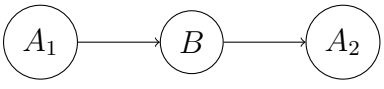
4.2.3.1 No cycles in version graph

Invoking the definition of *version graph* from section 3.5.8, in the current implementation, the version graph of a schema must never contain cycles.

In a NoSQL Schema model, an entity's versions must be unique in regards to their properties, and a union type's contained types must be unique. In our implementation, this requirement is kept by having entity versions be comparable to each other, having types be comparable to each other, and storing them within tree sets implemented by red-black trees.

When operating on these containers, entity versions are compared to each other to establish their ordering. To compare two entity versions, one has to compare their properties by names and types. When comparing types, one might encounter a situation where both compared types are aggregates. To compare two aggregates, one needs to compare their target entity versions. This comparison propagates along the edges of the version graph. If the version graph contains a cycle, the comparison may loop within that cycle indefinitely and never stop. Therefore, for the implementation to behave sanely, a schema's version graph must not contain cycles.

There is one operation in the inference approach design which might in certain circumstances create a cycle within the version graph—entity flattening. Suppose that a schema s has the following version graph. Versions A_1 and A_2 belong to the same entity, A, and version B belongs to another entity.



If the user decided to flatten entity A, then versions A_1 and A_2 would be merged, and a cycle $A_1 = A_2 \to B \to A_1 = A_2$ would be created in the version graph. To prevent this, the implementation of the entity-flattening operation contains a preliminary check:

Let e be the entity being flattened. If the version graph contains a path from any version of e to any other version of e, the flattening is aborted and an exception is thrown alerting the user of the reason for the abort.

4.2.3.2 Entity flattening and schema extensibility

In the current implementation, flattening any entity within a schema makes that schema ineligible for extension with additional data.

Entity flattening merges all versions of one entity, creating a single version which may contain optional properties and properties with union types. Extending the enclosing schema afterwards might then add more versions into the flattened entity, making it versioned again. However, the versioned entity invariants—all properties are required and do not have union type—would be violated.

This would put the entity into an intermediate state, invoking a number of edge cases in other algorithms. To avoid them, we have decided to disallow extending schemas containing flattened entities. With further work invested into investigating and resolving the edge cases, this limitation could be lifted.

4.3 Example applications

Our implementation of the new inference approach is conceived as a Java library. To let a non-technical user use it, it needs to be wrapped in an application with a proper user interface, either command-line or graphical.

We present a simple example application that can be used out-of-the-box to infer the schema of a dataset contained in a MongoDB database using a given Apache Spark instance and then saved the inferred schema to a local file. This application is located within the root project, denoted by the project path ":". The example application's source code can also be examined for an example usage of the approach implementation API.

Alongside the example application for the new approach, we created an example application for four out of 5 discussed existing inference approaches, each except the Klettke et al. approach. These applications provide the same functionality as the first example application: infer the schema of a dataset contained within a MongoDB database or database collection using the given inference approach, then save the inferred schema to a local file. All example applications provide the same user interface and, thanks to Gradle, can even be launched at the same time. This enables us and the user to easily compare the existing approaches on any given input in regards to their inferred schema and runtime performance.

Out of the existing approaches, the most complicated to wrap in an example application was the implementation by Frozza et al. due to the fact, that it is a Node.js application. Unlike other approaches, where delegating the inference to the implementation comprises calling the appropriate API methods, the example application for the Frozza et al. approach has to integrate with the Node.js application's Representational State Transfer (REST) API.

First, the example application logs in to the implementation application and retrieves a session token which is then included in all further requests. Afterwards, it creates a *batch* (an inference request), and waits for it complete. Finally, the example application requests the inferred schema and saves it to a file.

The Node.js server must obviously be running for the inference to be available. Since launching the Node.js server is a resource-heavy task, the launching and

stopping of the server is left up to manual actions by the user.

All approach implementations, both preexisting, and newly created ones, can be launched through the `run` Gradle task in the corresponding Gradle project. These launches can be parameterized using Gradle properties. More detailed information on how to run and parameterize the example applications for the inference approaches is available in the `README.md` file in the repository root.

4.4 Testing

We chose JUnit 5 as the main testing framework for the multi-project setup as it is the recommended and default JVM testing framework for Gradle.

A user can command a test run by running the appropriate Gradle task, `test`. Gradle interfaces with the JUnit Platform Launcher to launch the test run. The JUnit Platform Engine discovers available test engines at runtime and commands each to discover available unit tests on their class path. Then it selects which tests should be actually run, according to the input parameters, and gives the engines commands to run them. Finally, it collects and reports the test results back to Gradle.

As far as test engines go, the JUnit Jupiter Engine, for example, is the default test engine for JUnit 5 tests. A JUnit Vintage Engine is also provided by the creators of JUnit 5 for integrating existing JUnit 4 and older tests into the new platform. To integrate any other custom test API into the JUnit 5 Platform, there must be a test engine for that API available and visible at runtime which will provide the JUnit 5 Platform Engine with found test definitions on the launch of a test run.

Unit tests and integration tests were created alongside the inference approach implementation. These are located in directory `:impl/src/test`. Test code is located in directory `:impl/src/test/scala` while the test resources are in `:impl/src/test/resources`.

Since the majority of the implementation is done in Scala, a popular Scala unit test API called ScalaTest was used for writing the tests. While ScalaTest offers seamless integration into JUnit 4, integration with its successor, JUnit 5, is lacking, as the creators of ScalaTest provide no test engine compatible with JUnit 5 Platform as of yet.

Fortunately, we were not the first to have this issue. A GitHub community member created a library for this use case called *ScalaTest JUnit Runner* and published it via the JCenter Maven artifact repository. The source code and documentation is available on GitHub[3]. This allows us to launch the Scala unit tests written using ScalaTest, through the same interface as all other tests created by the authors of the existing schema inference approach implementations.

4.5 Running example

Finally, let us take a look at the results of running our approach implementation against the running example dataset, as well as the edge-case example dataset, defined in sections 2.1.5 and 2.1.5.1.

[3] `https://github.com/helmethair-co/scalatest-junit-runner`

The output for the running example, included as attachment C.1, is a NoSQL Schema model in XML format. At first glance, it looks very similar to the output by the Sevilla et al. approach. Let us point out the key differences between the two outputs.

In this output, properties are not subtyped themselves, instead they contain a `type` element, which encodes the type information.

Unlike in Sevilla et al. output where aggregation and reference multiplicity is given by `lowerBound` and `upperBound` attributes, in our output it is given by those types being located either directly within a `property` element, or nested within an `Array` type. For example, the property `article.author` expresses a one-to-one aggregation because the `Aggregate` type is not wrapped in an `Array`.

Unlike Sevilla et al., our model does not immediately state the optionality of a property. All properties are required inside a versioned entity. Flattening an entity reveals the optionality of properties.

This can be seen in attachment C.2, which was based on the same dataset, but all the entities were flattened. We can see, that the `article.attachments` property is optional, because it is present in only one of the two original `article` versions. On the other hand, the `article.comments` property is required as it is present in both original versions. However, it has a union type consisting of `String` and `Array<String>` as these were the types of that property in each version.

Finally, attachment C.4 shows the output for the edge-case example. Here we can see that the approach infers the `collection.empty_array` property to be an `Array` of `UnknownType`. The `collection.nested_ints` property is inferred as a two-dimensional array (`Array` of `Array`) of `Number`, and `collection.nested_objects` is a two-dimensional array of `Aggregate`s. Lastly, `collection.values` is inferred as `Array` of `UnionType` containing all different detected types.

5. Experimental analysis

Previously, we have described existing schema inference approaches and created a new approach that tackles unsolved problems. In this chapter we design, execute, and evaluate experiments which better demonstrate the behavior of individual approaches, identify points of failure, and illustrate differences between the approaches.

The first experiment constitutes a functional analysis of the given approaches. It exemplifies the functional behavior of individual approaches when met with datasets containing different schema features.

The second and final experiment—performance analysis—compares the relative runtime performance of the approaches by executing them in an identical environment and against an identical dataset.

5.1 Functional analysis

In a previous section, a running example is introduced which contains all schema features of interest to us, and is used as input to show basic functional behavior of the individual schema inference approaches, both existing ones and the novel one. The running example was chosen to be complete in regards to the schema features contained and small enough to be easily understandable by the reader. However, due to the latter requirement, it may be too short and succinct to clearly separate individual schema features. This may cause cross-influence of the handling of different schema features and distort the results of the functional analysis.

To prevent this we have created by hand a total of eight separate datasets for this experiment, each focusing on a different schema feature: *PrimitiveTypes*, *SimpleArrays*, *SimpleObjects*, *ComplexArrays*, *ComplexObjects*, *Optional*, *Union*, and *References*, in no particular order. They are located in the `/experiment` directory within the root of the GitHub repository[1] as JSON files.

These datasets were each imported to a collection named `articles`, each within a separate MongoDB database. Afterwards, all implemented inference approaches were run against all of these databases. Since a majority of the approaches behaved according to expectation with a majority of the datasets, we will only focus on the abnormalities detected.

The *SimpleArrays* dataset features an empty JSON array in the `nothings` property. The empty array is not handled correctly by the implementation of the approach by Sevilla et al., the implementation throws an uncaught exception during the inference. The implementation by Frozza et al. also has a problem with this edge case, although not as severe one—the resulting JSON Schema is invalid as it contains an invalid definition for the array element type.

The *ComplexArrays* dataset contains a two-dimensional array in the `nested_arrays` property. The dimensionality is not handled correctly by the Canovas et al. approach, which models the property as just a simple one-to-many relationship.

[1] `https://github.com/ivan-lattak/schema-inference`

In the *Optional* dataset, the optional properties were not modeled by Canovas et al. For unknown reason, the `body.compressed` optional property was not inferred in the model at all.

Sevilla et al. approach and our new approach inferred the union types in the *Union* dataset as versioned entities instead. In our approach, union type is used as the element type of the heterogeneous array in the `comments` property. This heterogeneous array is modeled by the Sevilla et al. approach as a tuple containing a string and a number instead. In the schema inferred by the Canovas et al. approach, the heterogeneous types are reduced to the most generic type—string.

Finally, the *References* dataset contains references to entities in two forms: a property named `article_id` and a property containing an Extended JSON `DBRef`. These proved difficult to handle for most inference approaches—the Sevilla et al. approach and Canovas et al. approach output an empty schema and an empty package definition, respectively, while the Frozza et al. approach ends with an error and does not output a schema.

5.2 Performance analysis

In addition to testing functional correctness, it is also necessary to compare the existing approaches and the new one in terms of runtime performance. This was done using a series of performance experiments, running the existing implementations against a number of datasets and recording the runtimes.

5.2.1 Execution

First, we have generated a dataset of 500,000 JSON documents, serving as the master dataset for our experiments. For this generation we have used the *jsongenerator* open-source library, whose source code is available on GitHub[2] and which is published as a Maven artifact in the JitPack repository. This library is able to generate JSON values according to a given JSON Schema. As the schema for the generation, we have chosen the JSON Schema inferred from the running example using our new approach, attached as attachment C.3.

We conducted a total of 8 experiments, differing in sample size. The chosen sample sizes were 1k, 2k, 4k, 8k, 16k, 32k, 64k, and 128k documents. Each experiment was conducted using the following steps:

1. Extract a randomly sampled subset of the given size for the experiment.
2. Run each algorithm on the extracted subset 10 times in succession.
3. Repeat the previous steps a total of 30 times.

Experiments of different sizes were chosen because we want to measure the performance of a given implementation as it changes depending on the number of input documents.

If we had extracted just one random sample of a given size for an experiment, the results could be distorted as the runtime performance of the algorithms could

[2] `https://github.com/jimblackler/jsongenerator`

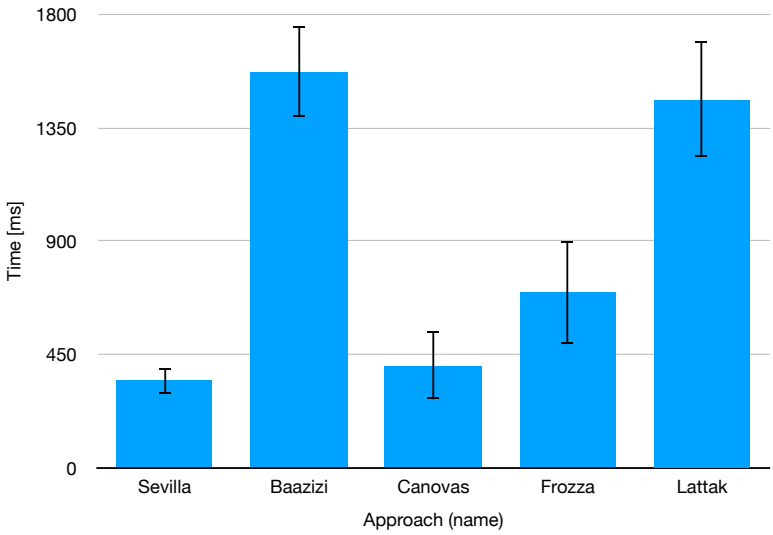


Figure 5.1: Average runtimes of measured inference approaches on data samples of 1k documents

become dependent on the particularities of each random selection. Thirty different random samples of a given size were extracted from the master dataset to mitigate this distortion.

Furthermore, if only a single run of each algorithm was performed for a given size, the results could be distorted by the runtime cost of algorithm initialization and would not reflect the sustained performance of the algorithm. Additional distortion could be caused by momentary decrease of system resources caused by random external influences. To mitigate both of these, ten runs of each algorithm were performed on each extracted subset.

In summary, for each of 8 experiment sizes, for each of 30 random subsets of that size, for each of 5 existing algorithms, 10 measurements, equals a total of $8 \cdot 30 \cdot 5 \cdot 10 = 12,000$ measurements were made.

The experiments were performed on a virtual machine running on VMware[3] infrastructure with 64 gibibytes of memory and 8 single-thread processor cores. Due to the virtual environment, we are unable to state further system specifications at the time of experiment.

When performing the measurements, the first run for each random sample for each approach was significantly longer than the rest, a so-called *warm-up* run. Measurements for these warm-up runs were dropped from the data so as not to skew the results.

5.2.2 Results

Figures 5.1 through 5.8 contain bar charts representing data gathered from individual experiments. The bars represent individual measured approaches. The y-axis is linear and represents the average runtime of that approach in milliseconds. At the top of each bar, the standard deviation for that approach is expressed by whiskers.

[3] `https://www.vmware.com/`

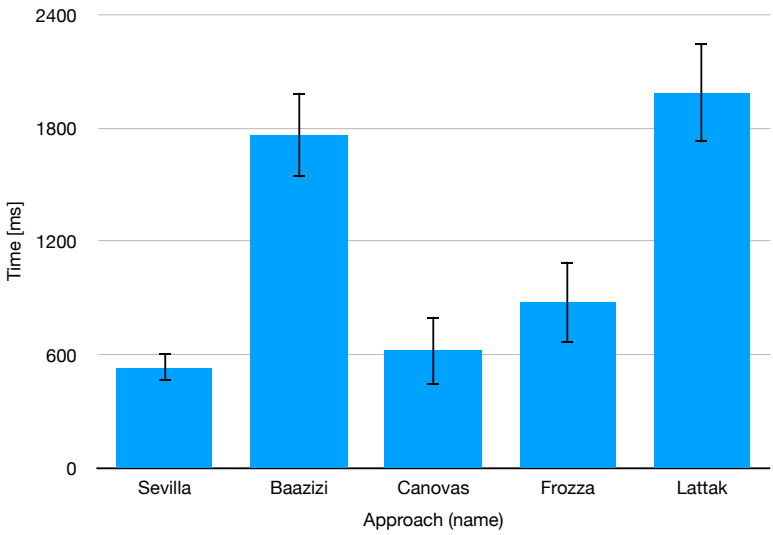


Figure 5.2: Average runtimes of measured inference approaches on data samples of 2k documents

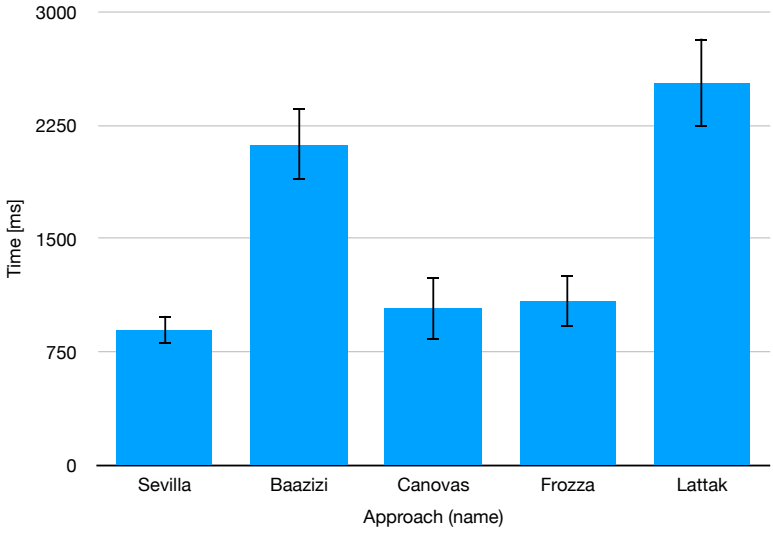


Figure 5.3: Average runtimes of measured inference approaches on data samples of 4k documents

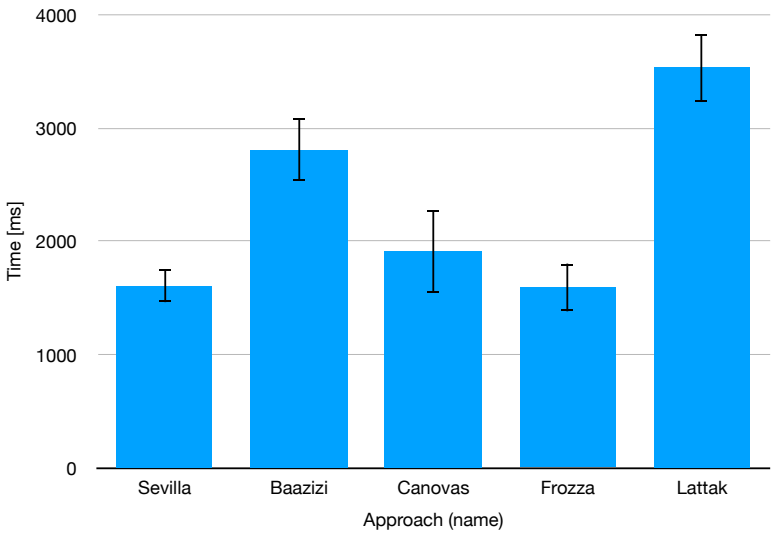


Figure 5.4: Average runtimes of measured inference approaches on data samples of 8k documents

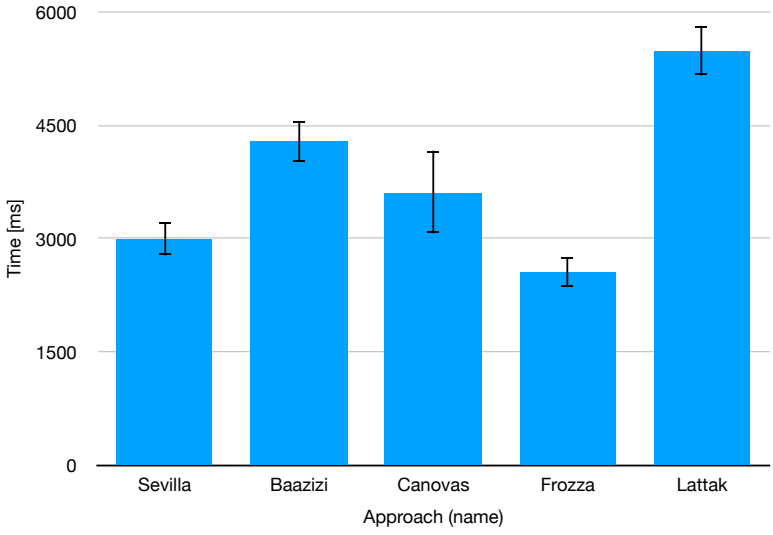


Figure 5.5: Average runtimes of measured inference approaches on data samples of 16k documents

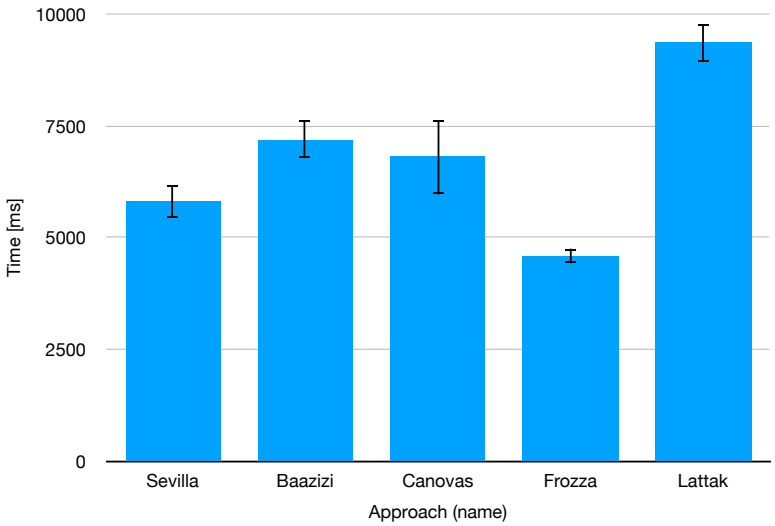


Figure 5.6: Average runtimes of measured inference approaches on data samples of 32k documents

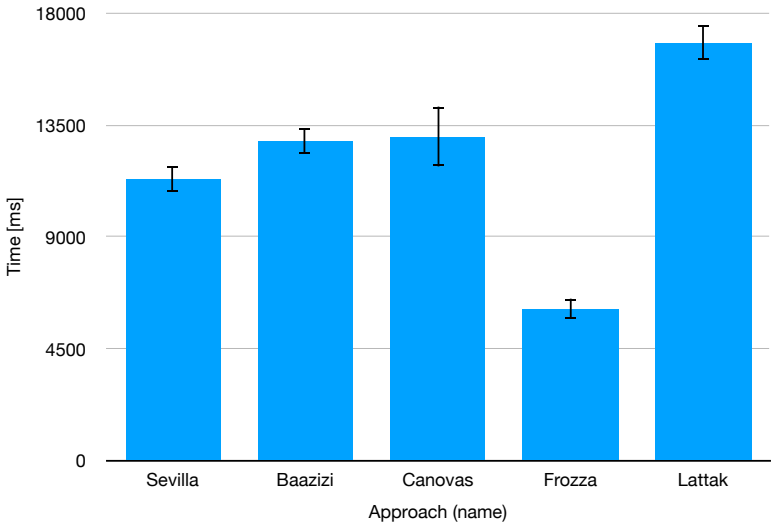


Figure 5.7: Average runtimes of measured inference approaches on data samples of 64k documents

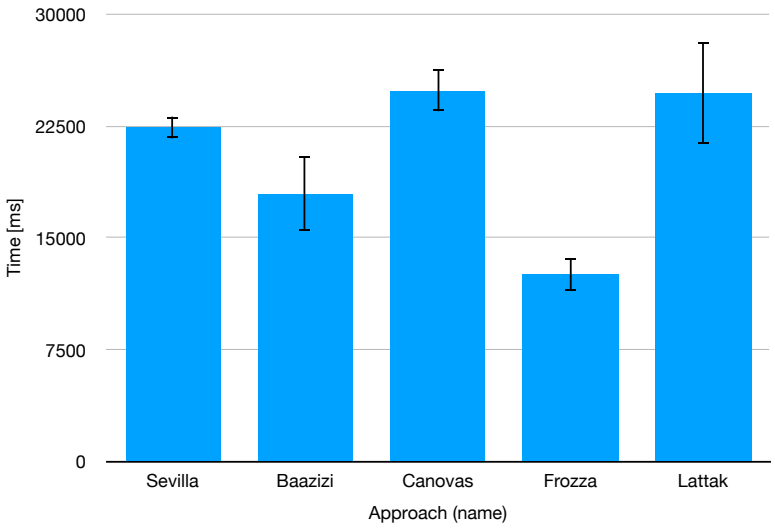


Figure 5.8: Average runtimes of measured inference approaches on data samples of 128k documents

In the smaller-sized experiments, we generally see larger standard deviations than in larger sizes. This is probably caused by external influences of the runtime environment having a greater relative impact on each individual measurement. Sevilla et al. runtime measurements seem to have a consistently low standard deviation, even, and especially, in smaller-sized experiments across experiment sizes.

The next two charts contain all the data from all of these eight partial charts combined, except for the standard deviations. They provide a clearer visualization of the dependence of runtime performance of each inference approach on the size of the input dataset.

Figure 5.9 contains a line chart describing the behavior of each of the approaches in all experiments. The x-axis represents the different experiment sizes. The y-axis is linear and represents the average runtime of each approach. The legend above provides meaning to the different colors of the lines.

The linear scale of this chart shows well the differences in average runtimes for large datasets. Frozza et al. performed best in the larger datasets, twice as fast as the slowest approaches. In the 16k, 32k, and 64k experiments Sevilla et. al., Baazizi et al., and Canovas et al. kept roughly identical performance. However, in the 128k experiment, Baazizi had significantly better performance compared to the other two. The new approach performs the worst consistently across experiment sizes except for the 1k experiment, where Baazizi et al. performed worst.

Attachment 5.10 contains a line chart almost identical to the previous, except the y-axis is logarithmic. This chart can better express the performance behavior for smaller-sized experiments since the logarithmic scale exaggerates relative differences in small numbers and shrinks them in large numbers.

More importantly, though, this chart demonstrates the linear scalability of each of the measured approaches. All five approaches exhibit their performance as more-or-less straight lines on the chart. Since both axes are logarithmic, this expresses a linear relationship between the size of the input and the average

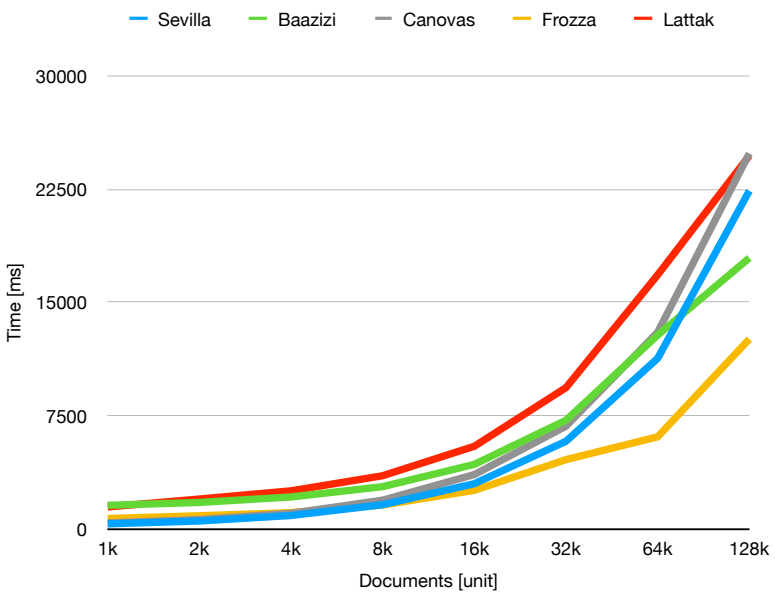


Figure 5.9: Average runtimes of measured inference approaches across all experiment sizes, linear scale

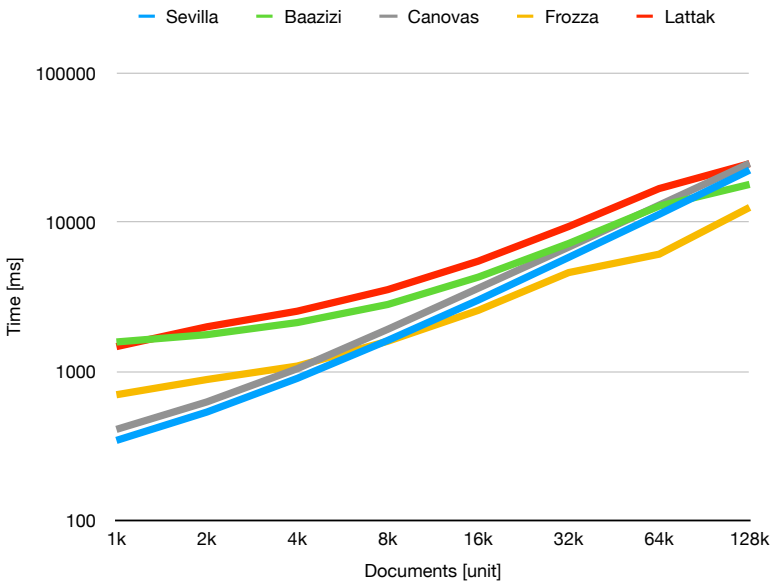


Figure 5.10: Average runtimes of measured inference approaches across all experiment sizes, logarithmic scale

runtime.

Looking at the logarithmic line chart, we can see that the Frozza et al. and especially Sevilla et al. and Canovas et al. approaches performed significantly better for smaller sample sizes. This can be attributed mainly to the high overhead cost of using Apache Spark in Baazizi et al. and the new approach.

This difference between (1) Sevilla et al. and Canovas et al. and (2) Baazizi et al. and our new approach becomes less pronounced in larger sample sizes. Baazizi et al. even started performing better in the largest experiment.

Performance of Baazizi et al. and our approach follows a very similar profile. This is expected as both approaches use Apache Spark in a very similar way. The performance advantage of the former compared to the latter is possibly caused by more simple schema inference process not involving model transformations.

5.2.3 Evaluation

The results of the experiment provide useful insight to the performance of each approach and the dependence of the performance on the size of the input data.

Different approaches are preferable to use performance-wise, depending on the size of the input data. For small datasets, Sevilla et al., Canovas et al., even Frozza et al. approaches are advisable, due to the fact that they do not incur high flat overhead costs of the Apache Spark framework unlike Baazizi et al. and our approach. Out of these, Sevilla et al. had the best performance. Combining that with an impressive feature set makes it the best option for small datasets.

As far as large datasets are considered, Frozza et al. is the best option if inference is to be run on a single machine. However, as MongoDB datasets sometimes span multiple database nodes and can contain upwards of millions of documents, horizontal scaling of schema inference may be desired. In that case, Frozza et al. is unsuitable, as it cannot be horizontally scaled.

Sevilla et al. and Canovas et al. can scale horizontally by decreasing the input size using MapReduce functionality built into MongoDB, in which case the number of MongoDB cluster nodes is the scaling factor. However in the worst-case scenario where every (or almost every) document has a unique raw schema this may not significantly decrease the input size, in which case the algorithm will run very slowly.

Baazizi et al. and our approach can linearly scale even for the describe worst-case scenario simply by adding nodes to the Apache Spark cluster the approach is being run with.

Between Baazizi et al. and our approach, the former runs faster so is the better option if inference speed is paramount. Of course, there is also the functional aspect to consider. Our approach provides more functionality: it can infer entity references, can be easily extended to work with database systems other than MongoDB, and provides more granular control over using versioned entities versus optional properties and union types in the resulting schema.

The lacking performance of our approach can be improved by further work. The performance of individual algorithm stages can be analyzed using a Java runtime profiler, such as *JProfiler*[4]. This can provide useful insight into the algorithm's performance behavior and highlight places which need optimization

[4] `https://www.ej-technologies.com/products/jprofiler/overview.html`

the most. For profiling the runtime of Apache Spark, one can use either a native tool, like the default Resource Manager of an Apache Spark node, or an external tool like *sparkMeasure*[5].

[5] `https://github.com/LucaCanali/sparkMeasure`

6. Future work

The newly designed NoSQL Schema metamodel and the related NoSQL data schema inference approach opens new horizons as it gives us another, in certain aspects superior, way how to model NoSQL data and infer its schema. As with everything, this allows us to see new possibilities of evolution for this metamodel and inference approach.

We believe the proposal of associativity of the schema folding operator (theorem 3.2) made in section 3.5.4 to be true, since it is extensively tested by unit tests. Nevertheless, it is not formally proven. A rigorous proof of the statement would confirm the general soundness of the approach design.

During development and testing, it became apparent, that the granularity of selectively certain entities and leaving others versioned is not sufficient. In some use cases, a user may want to selectively merge some entity versions of a given entity with others, while leaving other versions of that same entity untouched.

Since the performance analysis of the available approaches revealed that the performance of our approach is suboptimal in comparison, further work should be invested into improving this aspect, e.g. using profiling solutions described in section 5.2.3.

There are some complex types which can not be modeled by the NoSQL Schema metamodel, and whose integration into the metamodel and approach could increase the approach's usability: *Maps*, sometimes also called dictionaries, are similar in structure to JSON objects but are semantically different because the key set is not part of the schematic information (metadata), but is part of actual data. *Tuples* are special cases of arrays—they have a fixed size and the types of their positional elements must be modeled separately, not as a union. Tuples in this sense are supported by the metamodel by Sevilla et al. Finally, *sets*—unordered arrays—should be modeled distinctly from standard ordered arrays.

Another area of the approach that could be improved is the modeling of entity references. Currently, to confirm that a property is an entity reference, the inferred entities are searched to find one with matching name. To make this heuristic stronger, the reference itself could be checked against existing objects of the given entity type. Additionally, support for other than primitive-typed entity references, such as references with composite keys, could be beneficial.

Finally, the NoSQL Schema metamodel could serve as a basis for a possible future research into a proper multi-model approach, capable of modeling data from NoSQL, relational, and other logical models within one single schema.

Conclusion

Working in the context of *big data*, designing, implementing, and using any data processing algorithm only makes sense if its execution can be scaled horizontally, similar to how the data is often stored using horizontally scalable technologies.

Despite there having been numerous attempts in the recent past to devise an approach for schema inference from NoSQL data, there are still many areas in which these need to be improved. From the ability to model even deeply nested JSON structures, to various issues with the quite necessary horizontal scalability, there is a significant number of aspects in which the existing solutions are lacking. For instance, as far as we know, there still does not exist a schema inference approach for NoSQL or even just JSON data able to infer and detect other than very basic integrity constraints, i.e. inter-entity references, and even those are inferred only using very simple heuristics.

Our work takes the best ideas and principles from some of these solutions and applies them in a way suitable for the specific properties of NoSQL data, namely massive size of datasets and high heterogeneity. Although there is much room for improvement, we believe that our contribution can be used as a base for future research on the intricacies of NoSQL schema inference.

The work does not stop there. As we have mentioned earlier, today it often makes sense for a company to store different areas of their data using different storage technologies and, consequently, in different logical models. This *multi-model context* is especially difficult to develop a schema inference approach for and correctly model data in.

The JSON document model is quite complex, as are schema inference approaches based on it. Due to this complexity they are fitting as a starting point for the development of a true multi-model schema inference solution. A JSON schema inference approach could be expanded to infer schemas spanning different models, e.g. graph (Neo4j, RDF), key-value, or XML document model, and take into account specifics of each individual model.

Even in the presence of such multi-model inference approach, detection and modeling of inter-model entity references and other integrity constraints proves to be a non-trivial task.

Bibliography

[1] Irena Holubová, Jiří Kosek, Karel Minařík, and David Novák. *Big Data a NoSQL databáze*. Grada Publishing a.s., Prague, Czech Republic, 2015.

[2] MongoDB Inc. Data Models – MongoDB Documentation. `https://docs.mongodb.com/v5.0/data-modeling/`, 2021. Accessed: 2021-07-14.

[3] Neo4j Inc. Modeling Designs – Neo4j Documentation. `https://neo4j.com/developer/modeling-designs/`, 2021. Accessed: 2021-07-14.

[4] Elasticsearch B.V. Data in: Documents and Indices – Elasticsearch Documentation. `https://www.elastic.co/guide/en/elasticsearch/reference/7.x/documents-indices.html`, 2021. Accessed: 2021-07-14.

[5] Irena Holubová, Martin Svoboda, and Jiaheng Lu. Unified Management of Multi-model Data. In *Conceptual Modeling*, Lecture Notes in Computer Science, pages 439–447, Cham, Switzerland, 2019. Springer International Publishing Switzerland.

[6] Object Management Group. OMG Unified Modeling Language, Version 2.5.1. `https://www.omg.org/spec/UML/2.5.1/PDF`, 2017. Accessed: 2021-07-14.

[7] Pavel Čontoš, Irena Holubová, and Martin Svoboda. Multi-Model Data Modeling and Representation: State of the Art and Research Challenges. In *Proceedings 25th International Database Engineering & Applications Symposium*, IDEAS, 2021. In press.

[8] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesus Garcia Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Conceptual Modeling*, Lecture Notes in Computer Science, pages 467–480, Cham, Switzerland, 2015. Springer International Publishing Switzerland.

[9] Severino Feliciano Morales. *Inferring NoSQL Data Schemas with Model-Driven Engineering Techniques*. PhD thesis, University of Murcia, Murcia, Spain, March 2017.

[10] Alberto Hernandez Chillon, Severino Feliciano Morales, Diego Sevilla Ruiz, and Jesus Garcia Molina. Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases. 2017.

[11] Carlos Fernandez Candel, Diego Sevilla Ruiz, and Jesus Garcia-Molina. A Unified Metamodel for NoSQL and Relational Databases. *Computing Research Repository*, 2105.06494, 2021.

[12] Stefanie Scherzinger, Meike Klettke, and Uta Storl. Managing Schema Evolution in NoSQL Data Stores. *Computing Research Repository*, 1308.0514, 2013.

[13] Meike Klettke, Uta Storl, and Stefanie Scherzinger. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, pages 425–444, Bonn, Germany, 2015. Gesellschaft für Informatik e.V.

[14] Meike Klettke, Hannes Awolin, Uta Storl, Daniel Muller, and Stefanie Scherzinger. Uncovering the Evolution History of Data Lakes. In *2017 IEEE International Conference on Big Data*, pages 2380–2389, New York, United States, 2017. Institute of Electrical and Electronics Engineers Inc.

[15] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. *The VLDB Journal*, 28:497–521, 2019.

[16] Javier Luis Canovas Izquierdo and Jordi Cabot. Discovering Implicit Schemas in JSON Data. In *Web Engineering*, Lecture Notes in Computer Science, pages 68–83, Heidelberg, Germany, 2013. Springer-Verlag Berlin Heidelberg.

[17] Javier Luis Canovas Izquierdo and Jordi Cabot. JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowledge-Based Systems*, 103:52–55, 2016.

[18] 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 *Proceedings from the 2018 IEEE International Conference on Information Reuse and Integration for Data Science*, IRI, pages 356–363, 2018.

[19] Angelo Augusto Frozza, Eduardo Dias Defreyn, and Ronaldo dos Santos Mello. A Process for Inference of Columnar NoSQL Database Schemas. In *Anais do XXXV Simpósio Brasileiro de Bancos de Dados*, pages 175–180, Porto Alegre, RS, Brasil, 2020. SBC.

[20] Lanjun Wang, Oktie Hassanzadeh, Shuo Zhang, Juwei Shi, Limei Jiao, Jia Zou, and Chen Wang. Schema Management for Document Stores. In *Proceedings of the VLDB Endowment*, pages 922–933, New York, United States, 2015. Association for Computing Machinery.

[21] Mark Lukas Moller, Nicolas Berton, Meike Klettke, Stefanie Scherzinger, and Uta Storl. jHound: Large-Scale Profiling of Open JSON Data. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019)*, pages 555–558, Bonn, Germany, 2019. Gesellschaft für Informatik e.V.

[22] Michael Fruth, Kai Dauberschmidt, and Stefanie Scherzinger. Josch: Managing Schemas for NoSQL Document Stores. In *2021 IEEE 37th International Conference on Data Engineering*, ICDE, pages 2693–2696, 2021.

[23] Michael DiScala and Daniel J. Abadi. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD, pages 295—-310, New York, United States, 2016. Association for Computing Machinery.

[24] Irena Mlýnková and Martin Nečaský. Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues. *Informatica*, 24(4):577–602, 2013.

[25] Pavel Čontoš and Martin Svoboda. JSON Schema Inference Approaches. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling*, pages 173–183, Cham, Switzerland, 2020. Springer International Publishing.

[26] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of Concise Regular Expressions and DTDs. *ACM Transactions on Database Systems*, 35(2), 2010. Article no. 11, 47 p.

[27] Enrico Gallinucci, Matteo Golfarelli, Stefano Rizzi, Alberto Abello, and Oscar Romero. Interactive multidimensional modeling of linked data for exploratory OLAP. *Information Systems*, 77:86–104, 2018.

[28] Redouane Bouhamoum, Kenza Kellou-Menouer, Stephane Lopes, and Zoubida Kedad. Scaling Up Schema Discovery for RDF Datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops*, ICDEW, pages 84–89, 2018.

[29] The PostgreSQL Global Development Group. PostgreSQL 9.2.24 Documentation. `https://www.postgresql.org/docs/9.2/`, 2017. Accessed: 2021-04-11.

[30] International Organization for Standardization. ISO 21778:2017. `https://www.iso.org/standard/71616.html`, 2017. Accessed: 2021-04-05.

[31] Internet Engineering Task Force. The JavaScript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/html/rfc8259`, 2017. Accessed: 2021-04-05.

[32] Luke Lovett and David Golden. SPEC-587. `https://github.com/mongodb/specifications/blob/master/source/extended-json.rst`, 2020. Accessed: 2021-04-05.

[33] BSON (Binary JSON): Specification. `https://bsonspec.org/spec.html`. Accessed: 2021-04-05.

[34] Greg Dennis. JSON Schema: A Media Type for Describing JSON Documents, Draft 2020-12. `https://json-schema.org/draft/2020-12/json-schema-core.html`, 2020. Accessed: 2021-04-05.

[35] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[36] Internet Engineering Task Force. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. `https://datatracker.ietf.org/doc/html/rfc2046`, 1996. Accessed: 2021-07-17.

[37] International Organization for Standardization. ISO 8601-1:2019. `https://www.iso.org/standard/70907.html`, 2019. Accessed: 2021-03-15.

[38] MongoDB Inc. Database References – MongoDB Documentation. https://docs.mongodb.com/v5.0/reference/database-references/, 2021. Accessed: 2021-07-16.

[39] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. DTD-Miner, a tool for mining DTD from XML documents. In *Proceedings from the Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, WECWIS, pages 144–151, Washington, DC, United States, 2000. IEEE Computer Society.

[40] Oracle Corporation. Serializable, Java Development Kit Standard Edition 8 Documentation. `https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html`, 2014. Accessed: 2021-06-26.

List of Figures

Acronyms

API Application Programming Interface. 70, 72, 74, 77, 78

BSON Binary JSON. 14, 40

CSV Comma-Separated Values. 4

DTD Document Type Definition. 7

EMF Eclipse Modeling Framework. 11, 15, 36, 42, 72, 74, 76

HDFS Hadoop Distributed File System. 53, 73

IETF Internet Engineering Task Force. 34

JDK Java Development Kit. 73, 75

JVM Java Virtual Machine. 70, 73–75, 78

LOD Linked Open Data. 7

MDE Model-Driven Engineering. 6, 13, 19, 21, 23, 25, 34, 40, 42, 50, 52

OOP Object-Oriented Programming. 76

RDD Resilient Distributed Dataset. 50, 53, 58, 59, 74, 75

RDF Resource Description Framework. 6, 7, 91

REST Representational State Transfer. 77

RFC Request for Comments. 34

RSUS Raw Schema Unified Structure. 40, 41, 46

SG Structure Identification Graph. 28, 29, 46

SQL Structured Query Language. 4

UML Unified Modeling Language. 4, 5

XML Extensible Markup Language. 4, 6, 7, 26, 27, 38, 79, 91, 99

Glossary

aggregate-oriented A type of NoSQL databases that work with aggregated data, i.e. data which is organized so as to group together related data chunks. This is in contrast with data in relational database normal forms. 6, 11, 12, 21, 23, 24, 28, 37, 42, 46, 50, 53, 57

Extended JSON See section 2.1.4.2. 6, 14, 19, 26, 38, 40–42, 44, 55, 58, 60, 81

JSON Schema See section 2.1.4.4. 4, 11, 15, 25, 27–30, 34, 38, 40–46, 52, 67–69, 80, 81

MapReduce See section 2.1.4.6. 13, 16, 17, 21, 23, 25, 31, 45, 52, 53, 88

NoSQL A new generation of database technologies related to the big data movement. They favor scalability, speed of data access, and ease of new application development over the transactional correctness guarantees that traditional relational databases provide. They are usually schema-less. 4–6, 8, 11, 21, 23–28, 37, 40, 45, 50, 52–56, 59, 61–64, 66, 67, 70–72, 74, 76, 79, 90, 91

semi-structured Kind of data formats which use tags or other markers to define named fields and other structural elements of the data. Unlike with fully structured data, like that in relational databases, semi-structured data does not follow an upfront strictly defined schema. Examples include XML and JSON. 4–6, 10, 13, 44

A. Schemas inferred from the running example

A.1 Sevilla et al.

```
<?xml version="1.0" encoding="UTF-8"?>
<NoSQLSchema:NoSQLSchema xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns
    :NoSQLSchema="http://www.modelum.es/NoSQLSchema" xsi:schemaLocation
    ="http://www.modelum.es/NoSQLSchema platform:/resource/es.um.nosql.
    s13e/model/nosqlschema.ecore" name="inference">
  <entities name="Articles" root="true">
    <variations variationId="1" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="_id">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="Number"/>
      </properties>
      <properties xsi:type="NoSQLSchema:Aggregate" name="attachments"
          optional="true" upperBound="1" aggregates="//@entities.2/
          @variations.0"/>
      <properties xsi:type="NoSQLSchema:Aggregate" name="author"
          lowerBound="1" upperBound="1" aggregates="//@entities.1/
          @variations.0"/>
      <properties xsi:type="NoSQLSchema:Aggregate" name="body" optional
          ="true" lowerBound="1" upperBound="1" aggregates="//@entities
          .3/@variations.0"/>
      <properties xsi:type="NoSQLSchema:Attribute" name="comments"
          optional="true">
        <type xsi:type="NoSQLSchema:PList">
          <elementType xsi:type="NoSQLSchema:PrimitiveType" name="String
              "/>
        </type>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="published">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="Boolean"/>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="ratings">
        <type xsi:type="NoSQLSchema:PList">
          <elementType xsi:type="NoSQLSchema:PrimitiveType" name="Number
              "/>
        </type>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="timestamp">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
    </variations>
    <variations variationId="2" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="_id">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="Number"/>
```

```
    </properties>
    <properties xsi:type="NoSQLSchema:Reference" name="article_id"
       optional="true" lowerBound="1" upperBound="1" refsTo="//
       @entities.0" originalType="Number"/>
    <properties xsi:type="NoSQLSchema:Aggregate" name="author"
       lowerBound="1" upperBound="1" aggregates="//@entities.1/
       @variations.1"/>
    <properties xsi:type="NoSQLSchema:Attribute" name="body" optional
       ="true">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
    <properties xsi:type="NoSQLSchema:Attribute" name="comments"
       optional="true">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
    <properties xsi:type="NoSQLSchema:Attribute" name="published">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="Boolean"/>
    </properties>
    <properties xsi:type="NoSQLSchema:Attribute" name="ratings">
      <type xsi:type="NoSQLSchema:PList">
        <elementType xsi:type="NoSQLSchema:PrimitiveType" name="Number
           "/>
      </type>
    </properties>
    <properties xsi:type="NoSQLSchema:Attribute" name="timestamp">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
  </variations>
</entities>
<entities name="Author">
  <variations variationId="1" count="1">
    <properties xsi:type="NoSQLSchema:Attribute" name="first_name">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
    <properties xsi:type="NoSQLSchema:Attribute" name="last_name">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
    <properties xsi:type="NoSQLSchema:Aggregate" name="location"
       lowerBound="1" upperBound="1" aggregates="//@entities.4/
       @variations.0"/>
    <properties xsi:type="NoSQLSchema:Attribute" name="phone_number"
       optional="true">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
  </variations>
  <variations variationId="2" count="1">
    <properties xsi:type="NoSQLSchema:Attribute" name="first_name">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
    <properties xsi:type="NoSQLSchema:Attribute" name="last_name">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
```

```
      </properties>
      <properties xsi:type="NoSQLSchema:Aggregate" name="location"
          lowerBound="1" upperBound="1" aggregates="//@entities.4/
          @variations.1"/>
      <properties xsi:type="NoSQLSchema:Attribute" name="phone_number"
          optional="true">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="Number"/>
      </properties>
    </variations>
  </entities>
  <entities name="Attachment">
    <variations variationId="1" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="url">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
    </variations>
  </entities>
  <entities name="Body">
    <variations variationId="1" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="content">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="mime_type">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
    </variations>
  </entities>
  <entities name="Location">
    <variations variationId="1" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="latitude"
          optional="true">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="longitude"
          optional="true">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
    </variations>
    <variations variationId="2" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="address"
          optional="true">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
    </variations>
  </entities>
</NoSQLSchema:NoSQLSchema>
```

A.2 Klettke et al.

```
{ "$schema": "http://json-schema.org/draft-06/schema#",
  "properties": { "_id": { "type": "number" },
```

```
    "timestamp": { "type": "string" },
    "author": { "type": "object",
      "properties": { "first_name": { "type": "string" },
        "last_name": { "type": "string" },
        "phone_number": {
          "oneOf": [ { "type": "string" }, { "type": "number" } ] },
        "location": { "type": "object",
          "properties": { "latitude": { "type": "string" },
            "longitude": { "type": "string" },
            "address": { "type": "string" } },
          "additionalProperties": false, "required": [] } },
      "additionalProperties": false,
      "required": [ "first_name", "last_name",
        "phone_number", "location" ] },
    "ratings": { "type": "array", "items": { "type": "number" } },
    "comments": {
      "oneOf": [
        { "type": "array", "items": { "type": "string" } },
        { "type": "string" } ] },
    "attachments": { "type": "array",
      "items": { "type": "object",
        "properties": { "url": { "type": "string" } },
        "additionalProperties": false, "required": [ "url" ] } },
    "body": {
      "oneOf": [
        { "type": "object",
          "properties": { "content": { "type": "string" },
            "mime_type": { "type": "string" } },
          "additionalProperties": false,
          "required": [ "content", "mime_type" ] },
        { "type": "string" } ] },
    "published": { "type": "boolean" },
    "article_id": { "type": "number" } },
  "additionalProperties": false,
  "required": [ "_id", "timestamp", "author", "ratings", "comments",
    "body", "published" ] }
```

A.3 Baazizi et al., kind-equivalence

```
{ _id: Num, article_id: Num,
  attachments: [ { url: Str } ],
  author: { first_name: Str, last_name: Str,
    location: { address: Str, latitude: Str, longitude: Str },
    phone_number: Num + Str },
  body: Str + { content: Str, mime_type: Str },
  comments: Str + [ Str ],
  published: Bool,
  ratings: [ Num ],
  timestamp: Str }
```

A.4 Baazizi et al., label-equivalence

```
{ _id: Num, article_id: Num,
  author: { first_name: Str, last_name: Str,
    location: { address: Str }, phone_number: Num },
  body: Str, comments: Str,
  published: Bool, ratings: [ Num ], timestamp: Str } +
{ _id: Num, attachments: [ { url: Str } ],
  author: { first_name: Str, last_name: Str,
    location: { latitude: Str, longitude: Str },
    phone_number: Str },
  body: { content: Str, mime_type: Str },
  comments: [ Str ], published: Bool,
  ratings: [ Num ], timestamp: Str }
```

A.5 Canovas et al.

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="
    http://www.eclipse.org/emf/2002/Ecore" name="articles" nsURI="http
    ://jsonDiscoverer/discovered/articles" nsPrefix="discoa">
  <eClassifiers xsi:type="ecore:EClass" name="Article">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="_id"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EInt"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="timestamp"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EString"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference" name="author"
        lowerBound="1" eType="//Author" containment="true">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
    </eStructuralFeatures>
```

```
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="ratings"
      lowerBound="1" upperBound="-1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EInt"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="comments"
      lowerBound="1" upperBound="-1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference" name="attachments"
      lowerBound="1" upperBound="-1" eType="//Attachment" containment
      ="true">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
      <details key="ratioTotalFound" value="0.5"/>
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference" name="body"
      lowerBound="1" eType="//Body" containment="true">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="published"
      lowerBound="1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EBoolean"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="article_id">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
      <details key="ratioTotalFound" value="0.5"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EInt"/>
  </eStructuralFeatures>
</eClassifiers>
```

```
<eClassifiers xsi:type="ecore:EClass" name="Author">
  <eAnnotations source="coverage">
    <details key="totalFound" value="2"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="first_name"
      lowerBound="1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="last_name"
      lowerBound="1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="phone_number"
       lowerBound="1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference" name="location"
      lowerBound="1" eType="//Location" containment="true">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Attachment">
  <eAnnotations source="coverage">
    <details key="totalFound" value="4"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="url"
      lowerBound="1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="4"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
```

```
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Body">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="content"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EString"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mime_type"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EString"/>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Location">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="latitude"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EString"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="longitude"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="2"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EString"/>
    </eStructuralFeatures>
  </eClassifiers>
</ecore:EPackage>
```

A.6 Frozza et al.

```
{ "$schema": "http://json-schema.org/draft-06/schema#",
  "definitions": {
    "ObjectID": { "title": "ObjectID", "type": "object",
      "properties": { "$oid": { "type": "string" } },
      "required": [ "$oid" ] },
    "Date": { "title": "Date", "type": "object",
      "properties": { "$date": { "type": "string" } },
      "required": [ "$date" ] },
    "DBRef": { "title": "DBRef", "type": "object",
      "properties": { "$id": { "type": "string" },
        "$ref": { "type": "string" } },
      "required": [ "$id", "$ref" ] } },
  "properties": { "_id": { "$ref": "#/definitions/ObjectID" },
    "attachments": { "name": "attachments", "type": "array",
      "items": { "type": "object",
        "properties": { "url": { "name": "url", "type": "string" } },
        "additionalProperties": false, "required": [ "url" ] },
      "minItems": 1, "additionalItems": true },
    "author": { "type": "object",
      "properties": {
        "first_name": { "name": "first_name", "type": "string" },
        "last_name": { "name": "last_name", "type": "string" },
        "location": { "type": "object",
          "properties": {
            "latitude": { "name": "latitude", "type": "string" },
            "longitude": { "name": "longitude", "type": "string" },
            "address": { "name": "address", "type": "string" } },
          "additionalProperties": false, "required": [],
          "name": "author.location" },
        "phone_number": { "name": "phone_number",
          "anyOf": [ { "type": "string" }, { "type": "number" } ] }},
      "additionalProperties": false,
      "required": [ "first_name", "last_name",
        "location", "phone_number" ],
      "name": "author" },
    "body": { "name": "body",
      "anyOf": [
        { "type": "object",
          "properties": {
            "content": { "name": "content", "type": "string" },
            "mime_type": { "name": "mime_type", "type": "string" } },
          "additionalProperties": false,
          "required": [ "content", "mime_type" ] },
        { "type": "string" } ] },
    "comments": { "name": "comments",
      "anyOf": [
        { "name": "comments", "type": "array",
          "items": { "type": "string" }, "minItems": 1,
          "additionalItems": true },
        { "type": "string" } ] },
    "published": { "name": "published", "type": "boolean" },
```

```
  "ratings": { "name": "ratings", "type": "array",
    "items": { "type": "number" }, "minItems": 1,
    "additionalItems": true },
  "timestamp": { "$ref": "#/definitions/Date" },
  "article_id": { "$ref": "#/definitions/DBRef" } },
"additionalProperties": false,
"required": [ "_id", "author", "body", "comments", "published",
  "ratings", "timestamp" }
```

B. Schemas inferred from the edge-case example

B.1 Sevilla et al.

```
<?xml version="1.0" encoding="UTF-8"?>
<NoSQLSchema:NoSQLSchema xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns
    :NoSQLSchema="http://www.modelum.es/NoSQLSchema" xsi:schemaLocation
    ="http://www.modelum.es/NoSQLSchema platform:/resource/es.um.nosql.
    s13e/model/nosqlschema.ecore" name="inferenceEdgeCases">
  <entities name="Value">
    <variations variationId="1" count="1"/>
    <variations variationId="2" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="key" optional="
          true">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
      </properties>
    </variations>
  </entities>
  <entities name="Collection" root="true">
    <variations variationId="1" count="1">
      <properties xsi:type="NoSQLSchema:Attribute" name="_id">
        <type xsi:type="NoSQLSchema:PrimitiveType" name="ObjectId"/>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="nested_ints">
        <type xsi:type="NoSQLSchema:PList">
          <elementType xsi:type="NoSQLSchema:PList">
            <elementType xsi:type="NoSQLSchema:PrimitiveType" name="
                Number"/>
          </elementType>
        </type>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="nested_objects
          ">
        <type xsi:type="NoSQLSchema:PList">
          <elementType xsi:type="NoSQLSchema:PList">
            <elementType xsi:type="NoSQLSchema:PrimitiveType" name=""/>
          </elementType>
        </type>
      </properties>
      <properties xsi:type="NoSQLSchema:Attribute" name="values">
        <type xsi:type="NoSQLSchema:PTuple">
          <elements xsi:type="NoSQLSchema:PrimitiveType" name=""/>
          <elements xsi:type="NoSQLSchema:PrimitiveType" name="Boolean
              "/>
          <elements xsi:type="NoSQLSchema:PrimitiveType" name="Number"/>
          <elements xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
```

```
        <elements xsi:type="NoSQLSchema:PList"/>
        <elements xsi:type="NoSQLSchema:PList">
          <elementType xsi:type="NoSQLSchema:PrimitiveType" name="
              Number"/>
        </elements>
        <elements xsi:type="NoSQLSchema:PrimitiveType" name=""/>
        <elements xsi:type="NoSQLSchema:PrimitiveType" name=""/>
      </type>
    </properties>
  </variations>
 </entities>
 <entities name="Nested_object">
  <variations variationId="1">
    <properties xsi:type="NoSQLSchema:Attribute" name="key">
      <type xsi:type="NoSQLSchema:PrimitiveType" name="String"/>
    </properties>
  </variations>
 </entities>
</NoSQLSchema:NoSQLSchema>
```

B.2 Baazizi et al., kind-equivalence

```
{ _id: { $oid: Str }, empty_array: [ Empty() ],
  nested_ints: [ [ Num ] ], nested_objects: [ [ { key: Str } ] ],
  values: [ Null + Bool + Num + Str + { key: Str } + [ Num ] ] }
```

B.3 Baazizi et al., label-equivalence

```
{ _id: { $oid: Str }, empty_array: [ Empty() ],
  nested_ints: [ [ Num ] ], nested_objects: [ [ { key: Str } ] ],
  values: [ Null + Bool + Num + Str + {} + { key: Str } + [ Num ] ] }
```

B.4 Canovas et al.

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="
    http://www.eclipse.org/emf/2002/Ecore" name="collection" nsURI="http
    ://jsonDiscoverer/discovered/collection" nsPrefix="discoc">
  <eClassifiers xsi:type="ecore:EClass" name="Collection">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="_id"
        lowerBound="1" eType="//_id" containment="true">
      <eAnnotations source="coverage">
        <details key="totalFound" value="1"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
    </eStructuralFeatures>
```

```
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="empty_array"
      lowerBound="1" upperBound="-1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="nested_ints"
      lowerBound="1" upperBound="-1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EInt"/>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference" name="
      nested_objects" lowerBound="1" upperBound="-1" eType="//
      Nested_object" containment="true">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="values"
      lowerBound="1" upperBound="-1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="1"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="_id">
  <eAnnotations source="coverage">
    <details key="totalFound" value="2"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="$oid"
      lowerBound="1">
    <eAnnotations source="coverage">
      <details key="totalFound" value="2"/>
      <details key="ratioTotalFound" value="1.0"/>
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
        /2002/Ecore#//EString"/>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Nested_object">
```

```
    <eAnnotations source="coverage">
      <details key="totalFound" value="12"/>
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="key"
        lowerBound="1">
      <eAnnotations source="coverage">
        <details key="totalFound" value="12"/>
        <details key="ratioTotalFound" value="1.0"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="http://www.eclipse.org/emf
          /2002/Ecore#//EString"/>
    </eStructuralFeatures>
  </eClassifiers>
</ecore:EPackage>
```

B.5 Frozza et al.

```
{ "$schema": "http://json-schema.org/draft-06/schema#",
  "definitions": {
    "ObjectID": { "title": "ObjectID", "type": "object",
      "properties": { "$oid": { "type": "string" } },
      "required": [ "$oid" ] } },
  "properties": { "_id": { "$ref": "#/definitions/ObjectID" },
    "empty_array": { "name": "empty_array", "type": "array",
      "items": { "anyOf": [] }, "minItems": 0,
      "additionalItems": true },
    "nested_ints": { "name": "nested_ints", "type": "array",
      "items": { "name": "nested_ints", "type": "array",
        "items": { "type": "number" }, "minItems": 1,
        "additionalItems": true }, "minItems": 1,
      "additionalItems": true },
    "nested_objects": { "name": "nested_objects", "type": "array",
      "items": { "name": "nested_objects", "type": "array",
        "items": { "type": "object",
          "properties": {
            "key": { "name": "key", "type": "string" } },
          "additionalProperties": false, "required": [ "key" ] },
        "minItems": 1, "additionalItems": true },
      "minItems": 1, "additionalItems": true },
    "values": { "name": "values", "type": "array",
      "items": {
        "anyOf": [
          { "name": "values", "type": "array",
            "items": { "type": "number" },
            "minItems": 0, "additionalItems": true },
          { "type": "boolean" },
          { "type": "null" },
          { "type": "number" },
          { "type": "string" },
          { "type": "object", "properties": {
              "key": { "name": "key", "type": "string" } },
```

```
          "additionalProperties": false, "required": [] } ] },
    "minItems": 1, "additionalItems": true } },
"additionalProperties": false, "required": [ "_id", "empty_array",
  "nested_ints", "nested_objects", "values" ] }
```

C. New schema inference approach

C.1 NoSQL Schema model for the running example

```
<?xml version="1.0" encoding="UTF-8"?>
<NoSQLSchema:NoSQLSchema xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns
    :NoSQLSchema="http://www.ksi.mff.cuni.cz/NoSQLSchema" name="
    Inference DB">
  <entities name="article">
    <versions root="true" additionalCount="1">
      <properties name="_id">
        <type xsi:type="NoSQLSchema:Number"/>
      </properties>
      <properties name="article_id">
        <type xsi:type="NoSQLSchema:EntityReference" target="article">
          <originalType xsi:type="NoSQLSchema:Number"/>
        </type>
      </properties>
      <properties name="author">
        <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.2/
            @versions.0"/>
      </properties>
      <properties name="body">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="comments">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="published">
        <type xsi:type="NoSQLSchema:Boolean"/>
      </properties>
      <properties name="ratings">
        <type xsi:type="NoSQLSchema:Array">
          <elementType xsi:type="NoSQLSchema:Number"/>
        </type>
      </properties>
      <properties name="timestamp">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
    <versions root="true" additionalCount="1">
      <properties name="_id">
        <type xsi:type="NoSQLSchema:Number"/>
      </properties>
      <properties name="attachments">
```

```
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:Aggregate" target="//
            @entities.1/@versions.0"/>
      </type>
    </properties>
    <properties name="author">
      <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.2/
          @versions.1"/>
    </properties>
    <properties name="body">
      <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.3/
          @versions.0"/>
    </properties>
    <properties name="comments">
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:String"/>
      </type>
    </properties>
    <properties name="published">
      <type xsi:type="NoSQLSchema:Boolean"/>
    </properties>
    <properties name="ratings">
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:Number"/>
      </type>
    </properties>
    <properties name="timestamp">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
  </versions>
</entities>
<entities name="attachment">
  <versions additionalCount="1" aggregates="//@entities.0/@versions.1/
      @properties.1/@type/@elementType">
    <properties name="url">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
  </versions>
</entities>
<entities name="author">
  <versions aggregates="//@entities.0/@versions.0/@properties.2/@type
      ">
    <properties name="first_name">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
    <properties name="last_name">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
    <properties name="location">
      <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.4/
          @versions.0"/>
```

```
      </properties>
      <properties name="phone_number">
        <type xsi:type="NoSQLSchema:Number"/>
      </properties>
    </versions>
    <versions aggregates="//@entities.0/@versions.1/@properties.2/@type
        ">
      <properties name="first_name">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="last_name">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="location">
        <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.4/
            @versions.1"/>
      </properties>
      <properties name="phone_number">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
  </entities>
  <entities name="body">
    <versions aggregates="//@entities.0/@versions.1/@properties.3/@type
        ">
      <properties name="content">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="mime_type">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
  </entities>
  <entities name="location">
    <versions aggregates="//@entities.2/@versions.0/@properties.2/@type
        ">
      <properties name="address">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
    <versions aggregates="//@entities.2/@versions.1/@properties.2/@type
        ">
      <properties name="latitude">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="longitude">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
  </entities>
</NoSQLSchema:NoSQLSchema>
```

C.2 NoSQL Schema model for the running example, All entities flattened

```
<?xml version="1.0" encoding="UTF-8"?>
<NoSQLSchema:NoSQLSchema xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns
    :NoSQLSchema="http://www.ksi.mff.cuni.cz/NoSQLSchema" name="
    Inference DB">
  <entities name="article" flattened="true">
    <versions root="true" additionalCount="2">
      <properties name="_id">
        <type xsi:type="NoSQLSchema:Number"/>
      </properties>
      <properties name="article_id" optional="true">
        <type xsi:type="NoSQLSchema:EntityReference" target="article">
          <originalType xsi:type="NoSQLSchema:Number"/>
        </type>
      </properties>
      <properties name="attachments" optional="true">
        <type xsi:type="NoSQLSchema:Array">
          <elementType xsi:type="NoSQLSchema:Aggregate" target="//
              @entities.1/@versions.0"/>
        </type>
      </properties>
      <properties name="author">
        <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.2/
            @versions.0"/>
      </properties>
      <properties name="body">
        <type xsi:type="NoSQLSchema:UnionType">
          <types xsi:type="NoSQLSchema:String"/>
          <types xsi:type="NoSQLSchema:Aggregate" target="//@entities.3/
              @versions.0"/>
        </type>
      </properties>
      <properties name="comments">
        <type xsi:type="NoSQLSchema:UnionType">
          <types xsi:type="NoSQLSchema:String"/>
          <types xsi:type="NoSQLSchema:Array">
            <elementType xsi:type="NoSQLSchema:String"/>
          </types>
        </type>
      </properties>
      <properties name="published">
        <type xsi:type="NoSQLSchema:Boolean"/>
      </properties>
      <properties name="ratings">
        <type xsi:type="NoSQLSchema:Array">
          <elementType xsi:type="NoSQLSchema:Number"/>
        </type>
```

```
    </properties>
    <properties name="timestamp">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
  </versions>
</entities>
<entities name="attachment">
  <versions additionalCount="1" aggregates="//@entities.0/@versions.0/
      @properties.2/@type/@elementType">
    <properties name="url">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
  </versions>
</entities>
<entities name="author" flattened="true">
  <versions additionalCount="1" aggregates="//@entities.0/@versions.0/
      @properties.3/@type">
    <properties name="first_name">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
    <properties name="last_name">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
    <properties name="location">
      <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.4/
          @versions.0"/>
    </properties>
    <properties name="phone_number">
      <type xsi:type="NoSQLSchema:UnionType">
        <types xsi:type="NoSQLSchema:Number"/>
        <types xsi:type="NoSQLSchema:String"/>
      </type>
    </properties>
  </versions>
</entities>
<entities name="body">
  <versions aggregates="//@entities.0/@versions.0/@properties.4/@type/
      @types.1">
    <properties name="content">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
    <properties name="mime_type">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
  </versions>
</entities>
<entities name="location" flattened="true">
  <versions additionalCount="1" aggregates="//@entities.2/@versions.0/
      @properties.2/@type">
    <properties name="address" optional="true">
      <type xsi:type="NoSQLSchema:String"/>
```

```
      </properties>
      <properties name="latitude" optional="true">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
      <properties name="longitude" optional="true">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
  </entities>
</NoSQLSchema:NoSQLSchema>
```

C.3 JSON Schema for the running example

```
{ "$id": "http://ksi.mff.cuni.cz/schemas/inferred",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "description": "Inference DB",
  "definitions": {
    "#location.0": { "$id": "#location.0", "type": "object",
      "additionalProperties": false, "required": ["address"],
      "properties": { "address": { "type": "string" } } },
    "#location.1": { "$id": "#location.1", "type": "object",
      "additionalProperties": false,
      "required": ["latitude", "longitude"],
      "properties": {
        "latitude": { "type": "string" },
        "longitude": { "type": "string" } } },
    "#author.0": { "$id": "#author.0", "type": "object",
      "additionalProperties": false,
      "required": ["first_name", "last_name", "location",
        "phone_number"],
      "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" },
        "location": { "$ref": "#location.0" },
        "phone_number": { "type": "number" } } },
    "#attachment.0": { "$id": "#attachment.0", "type": "object",
      "additionalProperties": false, "required": ["url"],
      "properties": { "url": { "type": "string" } } },
    "#author.1": { "$id": "#author.1", "type": "object",
      "additionalProperties": false,
      "required": ["first_name", "last_name", "location",
        "phone_number"],
      "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" },
        "location": { "$ref": "#location.1" },
        "phone_number": { "type": "string" } } },
    "#body.0": { "$id": "#body.0", "type": "object",
      "additionalProperties": false,
      "required": ["content", "mime_type"],
      "properties": {
```

```
      "content": { "type": "string" },
      "mime_type": { "type": "string" } } },
  "#article.0": { "$id": "#article.0", "type": "object",
    "additionalProperties": false,
    "required": ["_id", "article_id", "author", "body", "comments",
      "published", "ratings", "timestamp"],
    "properties": {
      "_id": { "type": "number" },
      "article_id": {
        "description": "A reference to the 'article' entity",
        "type": "number" },
      "author": { "$ref": "#author.0" },
      "body": { "type": "string" },
      "comments": { "type": "string" },
      "published": { "type": "boolean" },
      "ratings": { "type": "array", "items": { "type": "number" } },
      "timestamp": { "type": "string" } } },
  "#article.1": { "$id": "#article.1", "type": "object",
    "additionalProperties": false,
    "required": ["_id", "attachments", "author", "body", "comments",
      "published", "ratings", "timestamp"],
    "properties": {
      "_id": { "type": "number" },
      "attachments": { "type": "array",
        "items": { "$ref": "#attachment.0" } },
      "author": { "$ref": "#author.1" },
      "body": { "$ref": "#body.0" },
      "comments": { "type": "array", "items": { "type": "string" } },
      "published": { "type": "boolean" },
      "ratings": { "type": "array", "items": { "type": "number" } },
      "timestamp": { "type": "string" } } } },
 "anyOf": [{ "$ref": "#article.0" }, { "$ref": "#article.1" }] }
```

C.4 NoSQL Schema model for the edge-case example

```
<?xml version="1.0" encoding="UTF-8"?>
<NoSQLSchema:NoSQLSchema xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns
    :NoSQLSchema="http://www.ksi.mff.cuni.cz/NoSQLSchema" name="
    Inference DB">
  <entities name="_id">
    <versions aggregates="//@entities.1/@versions.0/@properties.0/@type
        ">
      <properties name="$oid">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
  </entities>
  <entities name="collection">
    <versions root="true" additionalCount="1">
```

```
    <properties name="_id">
      <type xsi:type="NoSQLSchema:Aggregate" target="//@entities.0/
          @versions.0"/>
    </properties>
    <properties name="empty_array">
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:UnknownType"/>
      </type>
    </properties>
    <properties name="nested_ints">
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:Array">
          <elementType xsi:type="NoSQLSchema:Number"/>
        </elementType>
      </type>
    </properties>
    <properties name="nested_objects">
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:Array">
          <elementType xsi:type="NoSQLSchema:Aggregate" target="//
              @entities.2/@versions.0"/>
        </elementType>
      </type>
    </properties>
    <properties name="values">
      <type xsi:type="NoSQLSchema:Array">
        <elementType xsi:type="NoSQLSchema:UnionType">
          <types xsi:type="NoSQLSchema:Boolean"/>
          <types xsi:type="NoSQLSchema:Number"/>
          <types xsi:type="NoSQLSchema:String"/>
          <types xsi:type="NoSQLSchema:Array">
            <elementType xsi:type="NoSQLSchema:Number"/>
          </types>
          <types xsi:type="NoSQLSchema:Aggregate" target="//@entities
              .3/@versions.0"/>
          <types xsi:type="NoSQLSchema:Aggregate" target="//@entities
              .3/@versions.1"/>
        </elementType>
      </type>
    </properties>
  </versions>
</entities>
<entities name="nested_object">
  <versions additionalCount="5" aggregates="//@entities.1/@versions.0/
      @properties.3/@type/@elementType/@elementType">
    <properties name="key">
      <type xsi:type="NoSQLSchema:String"/>
    </properties>
  </versions>
</entities>
<entities name="value">
```

```
    <versions aggregates="//@entities.1/@versions.0/@properties.4/@type/
        @elementType/@types.4"/>
    <versions aggregates="//@entities.1/@versions.0/@properties.4/@type/
        @elementType/@types.5">
      <properties name="key">
        <type xsi:type="NoSQLSchema:String"/>
      </properties>
    </versions>
  </entities>
</NoSQLSchema:NoSQLSchema>
```

D. Proof of theorem 3.1

For the following proof let's shorten the names of functions for brevity's sake:

- foldSchemas function $\rightarrow f_S$
- foldEntities function $\rightarrow f_E$
- extractor of properties from entity version $\rightarrow p$.

f_S is only defined for two argument schemas with the same name. Therefore, the input arguments are each equivalent to the set of entities contained within and we can freely interchange between the two in our reasoning.

Lemma 1. *Property-dependent intersection of sets of entity versions is commutative.*

Proof. Given a set of versions V, let $\overline{p}$ be a function that applies the properties extractor to every version in a set:

$$\overline{p}(V) = \{p(v) | v \in V\}$$

Looking at the definition of property-dependent intersection for operands V_1 and V_2, we can see that the rule for deciding the result elements iterates over the set $\overline{p}(V_1) \cap \overline{p}(V_2)$. Since set intersection is commutative, the contents of this set do not depend on the order of operands.

For each element in that set, it constructs a new entity version. All parameters of those versions are independent on the order of operands for the following reasons:

- *properties* because the property set of v_1 is equal to that of v_2
- *root* because logical disjunction is commutative
- *aggregates* because set union is commutative
- *additionalCount* because integer addition is commutative

□

Lemma 2. *The foldEntities function is commutative.*

Proof.

$$\begin{aligned} f_E(e_1, e_2) &= (V_1 \setminus_V V_2) \cup (V_2 \setminus_V V_1) \cup (V_1 \cap_V V_2) && \text{(D.1)} \\ &= (V_2 \setminus_V V_1) \cup (V_1 \setminus_V V_2) \cup (V_1 \cap_V V_2) && \text{(D.2)} \\ &= (V_2 \setminus_V V_1) \cup (V_1 \setminus_V V_2) \cup (V_2 \cap_V V_1) && \text{(D.3)} \\ &= f_E(e_2, e_1) && \text{(D.4)} \end{aligned}$$

On line D.2, we swap the first two operands of the set union thanks to set union commutativity. On line D.3, we swap the operands of the property-dependent intersection within the last bracket thanks to commutativity of property-dependent intersection. □

Now we can proceed with the proof of theorem 3.1.

Proof. Given two input sets of entities M_1 and M_2. For each entity within M_1, either M_2 contains an entity with the same name or it doesn't. The same is true vice-versa: For each entity within M_2, either M_1 contains an entity with the same name or it doesn't. Let's introduce notation that will delimit these subsets. Each input entity set is composed of two disjoint subsets:

$$M_1 = M_{1E} \cup M_{1C}$$
$$M_2 = M_{2E} \cup M_{2C}$$

where M_{1E} is a set of entities which are **E**xclusive name-wise to M_1, M_{1C} is a set of entities which M_1 has in **C**ommon name-wise with M_2, and similarly for M_{2E} and M_{2C}.

f_S takes name-wise exclusive entities from both schemas and leaves them unchanged, but folds together matching pairs of name-wise common entities.

Let N_i be the set of names of entities in M_i. Let $M[s]$ be an entity within M with name s. Then:

$$f_S(M_1, M_2) = M_{1E} \cup M_{2E} \cup \left(\bigcup_{s \in N_1 \cap N_2} \{f_E(M_{1C}[s], M_{2C}[s])\} \right)$$

Therefore we have the following:

$$f_S(M_1, M_2) = M_{1E} \cup M_{2E} \cup \left(\bigcup_{s \in N_1 \cap N_2} \{f_E(M_{1C}[s], M_{2C}[s])\} \right) \tag{D.5}$$
$$= M_{2E} \cup M_{1E} \cup \left(\bigcup_{s \in N_1 \cap N_2} \{f_E(M_{1C}[s], M_{2C}[s])\} \right) \tag{D.6}$$
$$= M_{2E} \cup M_{1E} \cup \left(\bigcup_{s \in N_2 \cap N_1} \{f_E(M_{1C}[s], M_{2C}[s])\} \right) \tag{D.7}$$
$$= M_{2E} \cup M_{1E} \cup \left(\bigcup_{s \in N_2 \cap N_1} \{f_E(M_{2C}[s], M_{1C}[s])\} \right) \tag{D.8}$$
$$= f_S(M_2, M_1) \tag{D.9}$$

On line D.6, we swap the first two operands of the set union thanks to set union commutativity. On line D.7, we swap the operands of the set intersection within the big union limits thanks to commutativity of set intersection. On line D.8, we swap the operands of the f_E function because it is commutative as we have proven. □