Schema Inference for Massive JSON Datasets

Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani

▶ **To cite this version:**

Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani. Schema Inference for Massive JSON Datasets. Extending Database Technology (EDBT), Mar 2017, Venise, Italy. 10.5441/002/edbt.2017.21 . hal-01491765

HAL Id: hal-01491765

https://hal.sorbonne-universite.fr/hal-01491765

Submitted on 17 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Schema Inference for Massive JSON Datasets

Mohamed-Amine Baazizi
LIP6
Université Pierre et Marie Curie
Mohamed-Amine.Baazizi@lip6.fr

Houssem Ben Lahmar
IPVS
University of Stuttgart
Houssem.Ben-Lahmar@ipvs.uni-stuttgart.de

Dario Colazzo
Université Paris-Dauphine,
PSL Research University,
CNRS, LAMSADE
75016 PARIS, FRANCE
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica
Università di Pisa
ghelli@di.unipi.it

Carlo Sartiani
DIMIE
Università della Basilicata
sartiani@gmail.com

ABSTRACT

Recent years have seen the widespread use of JSON as a data format to represent massive data collections. JSON data collections are usually schemaless. While this ensures several advantages, the absence of schema information has important negative consequences: the correctness of complex queries and programs cannot be statically checked, users cannot rely on schema information to quickly figure out structural properties that could speed up the formulation of correct queries, and many schema-based optimizations are not possible.

In this paper we deal with the problem of inferring a schema from massive JSON data sets. We first identify a JSON type language which is simple and, at the same time, expressive enough to capture irregularities and to give complete structural information about input data. We then present our main contribution, which is the design of a schema inference algorithm, its theoretical study and its implementation based on Spark, enabling reasonable schema inference time for massive collections. Finally, we report about an experimental analysis showing the effectiveness of our approach in terms of execution time, precision and conciseness of inferred schemas, and scalability.

CCS Concepts

•**Information systems** → **Semi-structured data; Data model extensions;** •**Theory of computation** → *Type theory; Logic;*

Keywords

JSON, schema inference

1. INTRODUCTION

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Big Data applications typically process and analyze very large structured and semi-structured datasets. In many of these applications, and in those relying on NoSQL document stores in particular, data are represented in JSON, a data format that is widely used thanks to its flexibility and simplicity.

JSON data collections are usually schemaless. This ensures several advantages: in particular it enables modern applications to quickly consume huge amounts of semi-structured data without waiting for a schema to be specified. Unfortunately, the lack of a schema makes it impossible to statically detect unexpected or unwanted behaviors of complex queries and programs (i.e., lack of correctness), users cannot rely on schema information to quickly figure out structural properties that could speed up the formulation of correct queries, and many schema-based optimizations are not possible.

In this paper we deal with the problem of inferring a schema from massive JSON datasets. Our main goal in this work is to infer structural properties of JSON data, that is, a description of the structure of JSON objects and arrays that takes into account nested values and the presence of optional values. These are the main properties that characterize semi-structured data, and having a tool that ensures *fast*, *precise*, and *concise* inference is crucial in modern applications characterized by agile consumption of huge amounts of data coming from multiple and disparate sources.

The approach we propose here is based on a JSON schema language able to capture structural irregularities and complete structural information about input data. This language resembles and borrows mechanisms from existing proposals [20], but it has the advantage to be simple yet very expressive.

The proposed technique infers a schema that provides a *global* description of the whole input JSON dataset, while having a size that is small enough to enable a user to consult it in a reasonable amount of time, in order to get a global knowledge of the structural and type properties of the JSON collection. The description of the input JSON collection is global in the sense that each path that can be traversed in the tree-structure of each input JSON value can be traversed in the inferred schema as well. This property is crucial to enable a series of query optimization tasks. For instance, thanks to this property JSON queries [1, 10] can be optimized at compile-time by means of schema-based

path rewriting and wildcard expansion [16] or projection [9]. These optimizations are not possible if the schema hides some of the structural properties of the data, as happens in related approaches [22].

At the same time, our inferred schemas precisely capture the presence of optional and mandatory fields in collection of JSON records. Thanks to our approach, the user has a precise knowledge about i) all possible fields of records, ii) optional ones, and iii) mandatory ones. Property i) is crucial, as thanks to it the user can avoid time consuming, error-prone (approximated) data explorations to realize what fields can be really selected, while property ii) guides the user towards the adoption of code to handle the optional presence of certain fields; property iii), finally, indicates fields that can be always selected for each record in the collection.

A precise schema, like the one that can be inferred by our approach, can be very useful when very large datasets must be analyzed or queried with main-memory tools: indeed, by identifying the data requirements of a query or a program through a simple static analysis technique, it is possible to match these requirements with the schema in order to load in main memory only those fragments of the input dataset that are actually needed, hence improving both scalability and performance.

It is worth stressing that, even if in some cases JSON data feature a rather regular structure, the only alternative way for the user to be sure that all possible (optional) fields are identified is to explore the entire dataset either manually or by means of scripts that must be manually adapted to each particular JSON source, with weak guarantees of efficiency and soundness. Our approach instead applies to any JSON data collection, and is shown to be sound and effective on massive datasets. In addition, it is worth observing that, while in many cases processed JSON data come from remote, uncontrolled sources, in other particular cases JSON data are generated by applications whose code is known. In these cases more knowledge is available about the structure of the program output, but again schema inference is important as it can highlight subtle structural properties that can arise only in outputs of some particular program runs; also, when the code starts being complex, it is difficult to precisely figure out the structure of output JSON data. In some other cases, remote JSON sources can be accessed by APIs (e.g., Twitter APIs) that sometimes are provided with some schema descriptions. Unfortunately, these descriptions are often incomplete, often some fields are ignored, and the distinction between optional and mandatory fields is often omitted.

Our Contribution. Our main contribution is the design of a schema inference algorithm and its implementation based on Spark, in order to ensure reasonable schema inference time for massive collections. Our schema inference approach consists of two main steps. In the first one, an input collection of JSON values is processed by a Map transformation in order to infer a simple type for each value. The resulting output is processed by a Reduce action, which *fuses* inferred types that are not necessarily identical, but that share similar structure. This step relies on a binary function that takes two JSON types as input and fuses them. This function inspects the two input types identifying parts that are mandatory, optional, or repeated in the types, in order to obtain a type which is a super type of the two input types (it includes them), but that is potentially much more succinct than their simple union. A theoretical study shows that the fusion function is correct and, very importantly, associative.

Associativity is crucial as it allows Spark to safely distribute and parallelize the fusion of a massive collection of values. Associativity is also important to enable incremental evolution of the inferred schema under updates. In many applications the JSON sources are dynamic, and new values can be added at any time, with a structure that can differ from that already inferred for previous records. In this situation, in the case of insertion of a new record in an existing record collection, thanks to associativity, we simply need to fuse the existing schema with the schema of the new record. For incremental maintenance under other forms of updates, in the usual case that a massive data set is kept partitioned and the updated parts are known, it just suffices to re-infer the schema for the updated parts and to fuse them with previously inferred schemas for unchanged parts.

Our last contribution consists of an implementation of the proposed approach based on Spark, and experimental evaluation validating our claims of succinctness, precision, and efficiency. We based our tests on 4 real JSON datasets. Our experiments confirm that our schema inference algorithm returns very succinct yet precise schemas, even in the presence of poorly organized data (i.e., Wikipedia dataset). Also, a scalability analysis reveals that our approach ensures reasonable execution times, and that a simple partitioning strategy allows the performance to be improved.

Paper Outline. The paper is organized as follows. In Section 2 we illustrate some scenarios that motivate our work. In Section 3, then, we survey existing works. In Section 4, we describe the data model and the schema language we use here, while in Section 5 we present our schema inference approach. In Sections 6 and 7, finally, we show the results of our experimental evaluation and draw our conclusions.

2. MOTIVATION AND OVERVIEW

This section overviews the two steps of our schema fusion approach: type inference and type fusion. To this end, we first briefly recall the general syntax and semantics of JSON values. As in most semi-structured models, JSON distinguishes between basic values, which range over numbers (e.g., 123), strings (e.g., "abc"), and booleans (i.e., true/-false), and complex values which can be either (unordered) sets of key/value pairs called *records* or (ordered) lists of values called *arrays*. The only constraint that JSON values must obey is key uniqueness within each record. Arrays can mix both basic and complex types. In the following, we will use the term *mixed-content arrays* for arrays mixing atomic and complex values.

A sample JSON record is illustrated in Figure 1. Syntactically, records use the conventional curly braces symbols whereas arrays use square brackets; finally, string values and keys are wrapped inside quotes in JSON (but we will avoid quotes around keys in our formal syntax).

Type inference.

Type inference, during the Map phase, is dedicated to inferring individual types for the input JSON values and yields a set of distinct types to be fused during the Reduce

```
{"A": 123
 "B": "The ..."
 "C": false
 "D": ["abc", "cde", "fr12"]
}
```

Figure 1: A JSON record r_1.

phase. Some proposals of JSON schemas exist in the literature. With one exception [20], none of them uses regular expressions which, as we shall illustrate, are important for concisely representing types for array values. Moreover, a clean formal semantics specification of types is often missing in these works, hence making it difficult to understand their precise meaning.

The type language we adopt is meant to capture the core features of the JSON data model with an emphasis on succinctness. Intuitively, basic values are captured using standard data types (String, Number, Boolean), complex values are captured by introducing record and array type constructors, and a union type constructor is used to add flexibility and expressive power. To illustrate the type language, observe the following type that is inferred for the record r_1 given in Figure 1:

$$\{A : \texttt{Num}, B : \texttt{Str}, C : \texttt{Bool}, D : [\texttt{Str}, \texttt{Str}, \texttt{Str}]\}$$

As we will show, the initial type inference is a quite simple and fast operation: it is consists in a simple traversal of the input values that produces a type that is isomorphic to the value.

Type fusion.

Type fusion is the second step of our approach and consists in iteratively merging the types produced during the Map phase. Because it is performed during the Reduce phase in a distributed fashion, type fusion relies on a fusion operator which enjoys the commutativity and associativity properties. This fusion operator is invoked over two types T_1 and T_2 and produces a supertype of the inputs. To do so, the fusion collapses the parts of T_1 and T_2 that are identical and preserves the parts that are distinct in both types. To this end, T_1 and T_2 are processed in a synchronised top-down manner in order to identify common parts. The main idea is to only represent once what is common, and, at the same time, to preserve all the parts that differ.

Fusion treats atomic types, record types and array types differently, as follows.

- Atomic types: the fusion of atomic types is obvious: identical types are collapsed while different types are combined using the union operator.
- Record types: recall that valid record types enjoy key uniqueness. Therefore, the fusion of T_1 and T_2 is led by two rules:

  (R_1) matching keys from both types are collapsed and their respective types are recursively fused;

  (R_2) keys without a match are deemed optional in the resulting type and decorated with a question mark ?.

  To illustrate those cases, assume that T_1 and T_2 are, respectively, $\{A{:}\texttt{Str}, B{:}\texttt{Num}\}$ and $\{B{:}\texttt{Bool}, C{:}\texttt{Str}\}$. The only matching key is "B" and hence its two atomic types Num and Bool are fused, which yields $\texttt{Num}+\texttt{Bool}$. The other keys will be optional according to rule R_2. Hence, fusion yields the type

  $$T_{12} = \{(A{:}\texttt{Str})?, B{:}\texttt{Num} + \texttt{Bool}, (C{:}\texttt{Str})?\}$$

  Assume now that T_{12} is fused with

  $$T_3 = \{A{:}\texttt{Null}, B{:}\texttt{Num}\}$$

  Rules R_1 and R_2 need to be slightly adapted to deal with optional types. Intuitively, we should simply consider that optionality '?' prevails over the implicit *total* cardinality '1'. The resulting type is thus

  $$T_{123} = \{(A{:}\texttt{Str} + \texttt{Null})?, B{:}\texttt{Num} + \texttt{Bool}, (C{:}\texttt{Str})?\}.$$

  Fusion of nested records eventually associates keys with types that may be unions of atomic types, record types, and array types. We will see that, when such types are merged, we separately merge the atomic types, the record types, and the array types, and return the union of the result. For instance, the fusion of types

  $$\{l{:}(\texttt{Bool} + \texttt{Str} + \{A{:}\texttt{Num}\})\}$$

  and

  $$\{l{:}(\texttt{Bool} + \{A{:}\texttt{Str}, B{:}\texttt{Num}\})\}$$

  yields

  $$\{l{:}(\texttt{Bool} + \texttt{Str} + \{A{:}(\texttt{Num} + \texttt{Str}), (B{:}\texttt{Num})?\})\}.$$

- Array types: array fusion deserves special attention. A particular aspect to consider is that an array type obtained in the first phase may contain several repeated types, and may feature mixed-content. To deal with this, before fusing types we perform a kind of simplification on bodies by using regular expression types, and, in particular, union + and repetition types $*$. To illustrate this point, consider the array value

  $$[''abc'', '' cde'', \{''E'' : ''fr'', \ ''F'' : 12\}],$$

  containing two strings followed by a record (mixed-content). The first phase infers for this value the type

  $$[\texttt{Str}, \texttt{Str}, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}].$$

  This type can be actually simplified. For instance, one can think of a *partition*-based approach which collapses adjacent identical types into a star-guarded type, thus transforming

  $$[\texttt{Str}, \texttt{Str}, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}]$$

  into

  $$[(\texttt{Str})*, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}]$$

  by collapsing the string types. The resulting schema is indeed succinct and precise. However, succinctness cannot be guaranteed after fusion. For instance, if that type was to be merged with

  $$[\{E{:}\texttt{Str}, F{:}\texttt{Num}\}, \texttt{Str}, \texttt{Str}],$$

  where strings and record swapped positions, succinctness would be lost because we need to duplicate at least one sub-expression, $(\texttt{Str})*$ or $\{E{:}\texttt{Str}, F{:}\texttt{Num}\}$. As we are mainly concerned with generating types that can

be human-readable, we trade some precision for succinctness and do not account for position anymore. To achieve this, in our simplification process (made before fusing array types) we generalize the above partition-based solution by returning the star-guarded union of all distinct types expressed in an array. So, simplification for either

$$[\texttt{Str}, \texttt{Str}, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}]$$

or

$$[\{E{:}\texttt{Str}, F{:}\texttt{Num}\}, \texttt{Str}, \texttt{Str}]$$

yields the same type

$$S = [(\texttt{Str} + \{E{:}\texttt{Str}, F{:}\texttt{Num}\})*].$$

After the array types have been so simplified, they are fused by simply recursively fusing their content types, applying the same technique described for record types: when the body type is a union type, we separately merge the atomic components, the array components, and the record components, and take the union of the results.

3. RELATED WORK

The problem of inferring structural information from JSON data collections has recently attracted attention in the database research community. The closest work to ours is the very preliminary investigation that we presented in [12]. While [12] only provides a sketch of a MapReduce approach for schema inference, in this paper we present results about a much deeper study. In particular, while in [12] a declarative specification of only a few cases of the fusion process is presented, in this paper we fully detail this process, provide a formal specification as well as a fusion algorithm. Furthermore, differently from [12], we present here an experimental evaluation of our approach validating our claims of parallelizability and succinctness.

In [22] Wang et al. present a framework for efficiently managing a schema repository for JSON document stores. The proposed approach relies on a notion of JSON schema called *skeleton*. In a nutshell, a skeleton is a collection of trees describing structures that frequently appear in the objects of JSON data collection. In particular, the skeleton may totally miss information about paths that can be traversed in some of the JSON objects. In contrast, our approach enables the creation of a complete yet succinct schema description of the input JSON dataset. As already said, having such a complete structural description is of vital importance for many tasks, like query optimisation, defining and enforcing access-control security policies, and, importantly, giving the user a global structural vision of the database that can help her in querying and exploring the data in an effective way. Another important application of complete schema information is query type checking: as illustrated in [12] our inferred schemas can be used to make type checking of Pig Latin scripts much stronger.

In a very recent work [20], motivated by the need of laying the formal foundations for the JSON Schema language [3], Pezoa et al. present the formal semantics of that language, as well as a theoretical study of its related expressive power and validation problem. While that work does not deal with the schema inference problem, our schema language can be seen as a core part of the JSON Schema language studied therein, and shares union types and repetition types with that one. These constructors are at the basis of our technique to collapse several schemas into a more succinct one. An alternative proposal for typing JSON data is JSound [2]. That language is quite restrictive wrt ours and JSON Schemas: for instance it lacks union types.

In a very recent work [13] Abadi and Discala deal with the problem of automatic transforming denormalised, nested JSON data into normalised relational data that can be stored into a RDBMS; this is achieved by means of a schema generation algorithm that learns the *normalised, relational* schema from data. Differently from that work, we deal with schemas that are far from being relational, and are closer to tree regular grammars [17]. Furthermore, the approach proposed in [13] ignores the original structure of the JSON input dataset and, instead, depends on patterns in the attribute data values (functional dependencies) to guide its schema generation. So, that approach is complementary to ours.

In [15] Liu et al. propose storage, querying, and indexing principles enabling RDBMSs to manage JSON. The paper does not deal with schema inference, but indicates a possible optimisation of their framework based on the identification of common attributes in JSON objects that can be captured by a relational schema for optimization purposes. In [21] Scherzinger et al. propose a plugin to track changes in object-NoSQL mappings. The technique is currently limited to only detect mismatches between base types (e.g., Boolean, Integer, String), and the authors claim that a wider knowledge of schema information is needed to enable the detection of other kinds of changes, like, for instance, the removal or renaming of attributes.

It is important to state that the problem of schema inference has already been addressed in the past in the context of semi-structured and XML data models. In [18] and [19], Nestorov et al. describe an approach to extract a schema from semistructured data. They propose an object-oriented type system where nodes are captured by classes built starting from nodes sharing the same incoming and outcoming edges and where data edges are generalized to relations between the classes. In [19], the problem of building a type out a of a collection of semistructured documents is studied. The emphasis is put on minimizing the size of the resulting type while maximizing its precision. Although that work considers a very general data model captured by graphs, it does not suit our context. Firstly, we consider the JSON model that is tree-shaped by nature and that features specific constructs such as arrays that are not captured by the semi-structured data model. Secondly, we aim at processing potentially large datasets efficiently, a problem that is not directly addressed in [18] and [19].

More recent effort on XML schema inference (see [14] and works cited therein) is also worth mentioning since it is somewhat related to our approach. The aim of these approaches is to infer restricted, yet expressive enough forms of regular expressions starting from a positive set of strings representing element contexts of XML documents. While XML and JSON both allow one to represent tree-shaped data, they have radical differences that make existing XML related approaches difficult to apply to the JSON setting. Similar remarks hold for related approaches for schema inference for RDF [11]. Also, none of these approaches is designed to deal with massive datasets.

4. DATA MODEL AND TYPE LANGUAGE

This section is devoted to formalizing the JSON data model and the schema language we adopt.

We represent JSON values as records and arrays, whose abstract syntax is given in Figure 2. Basic values B comprise *null* values, booleans, numbers n and strings s. As outlined in Section 2, records are sets of fields, each field being an association of a value V to a key l whereas arrays are sequences of values. The abstract syntax is practical for the formal treatment, but we will typically use the more readable notation introduced at the bottom of Figure 2, where records as represented as $\{l_1 : V_1, \ldots, l_n : V_n\}$ and arrays are represented as $[V_1, \ldots, V_n]$.

$$
\begin{array}{lll}
V ::= & B \mid R \mid A & \text{Top-level values} \\
B ::= & \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid n \mid s & \text{Basic values} \\
R ::= & \mathit{ERec} \mid \mathit{Rec}(l, V, R) & \text{Records} \\
A ::= & \mathit{EArr} \mid \mathit{Arr}(V, A) & \text{Arrays}
\end{array}
$$

Semantics:

Records

$$
\begin{array}{lll}
\mathit{Domain} : \mathit{FS}(\mathit{Keys} \times \mathit{Values}) & & \\
[\![\mathit{ERec}]\!] & \triangleq & \emptyset \\
[\![\mathit{Rec}(l, V, R)]\!] & \triangleq & \{(l, V)\} \cup [\![R]\!]
\end{array}
$$

Arrays

$$
\begin{array}{lll}
\mathit{Domain} : \mathit{Lists}(\mathit{Values}) & & \\
[\![\mathit{EArr}]\!] & \triangleq & [\,] \\
[\![\mathit{Arr}(V, A)]\!] & \triangleq & [\![V]\!] :: A
\end{array}
$$

Notation:

$$
\begin{array}{lll}
\{l_1 : V_1, \ldots, l_n : V_n\} & \triangleq & \mathit{Rec}(l_1, V_1, \ldots \mathit{Rec}(l_n, V_n, \mathit{ERec})) \\
[V_1, \ldots, V_n] & \triangleq & \mathit{Arr}(V_1, \ldots \mathit{Arr}(V_n, \mathit{EArr}))
\end{array}
$$

Figure 2: Syntax of JSON data.

In JSON, a record is well-formed only if all its top-level keys are mutually different. In the sequel, we only consider well-formed JSON records, and we use Keys(R) to denote the set of the top-level keys of R.

Since a record is a set of fields, we identify two records that only differ in the order of their fields.

The syntax of the JSON schema language we adopt is depicted in Figure 3. The core of this language is captured by the non-terminals BT, RT, and AT which are a straightforward generalization of their B, R and A counterparts from the data model syntax.

As previously illustrated in Section 2, we adopt a very specific form of regular types in order to prepare an array type for fusion. Before fusion, an array type $[T_1, \ldots, T_n]$ is simplified as $[(T_1 + \ldots + T_n)*]$, or, more precisely, as $[\mathit{LFuse}(T_1, \ldots, T_n)*]$: instead of giving the content type element by element as in $[T_1, \ldots, T_n]$, we just say that it contains a sequence of values all belonging to $\mathit{LFuse}(T_1, \ldots, T_n)$ that will be defined as a compact upper bound of $T_1 + \ldots + T_n$. This simplification is allowed by the fact that, besides the basic array types $AT = [T_1, \ldots, T_n]$ we also have the simplified array type $SAT = [T*]$, where T may be any type, including a union type.

A field $\mathit{OptRecT}(l, T, \ldots)$, represented as $l : T?$ in the simplified notation, represents an optional field, that is, a field that may be either present or absent in a record of the corresponding type. For example, a type $\{l : \texttt{Num}?, m : (\texttt{Str} + \texttt{Null})\}$ describes records where l is optional and, if present, contains a number, while the m field is mandatory and may contain either *null* or a string.

A union type $T + U$ contains the union of the values from T and those for U. The empty type ϵ denotes the empty set.[1]

We define now schema semantics by means of the function $[\![_]\!]$, defined as the minimal function mapping types to sets of values that satisfies the following equations. For the sake of simplicity we omit the case of basic types.

Auxiliary functions

$$
\begin{array}{lll}
S^0 & \triangleq & \{[\,]\} \\
S^{n+1} & \triangleq & \{[V] :: a \mid V \in S, a \in S^n[\,]\} \\
S^* & \triangleq & \bigcup_{i \in N} S^i
\end{array}
$$

Records

$$
\begin{array}{lll}
\mathit{Domain} : \mathit{Sets}(\mathit{FS}(\mathit{Keys} \times \mathit{Values})) & & \\
[\![\mathit{ERecT}]\!] & \triangleq & \{\emptyset\} \\
[\![\mathit{RecT}(l, T, RT)]\!] & \triangleq & \{\{(l, V)\} \cup R \mid V \in [\![T]\!],\ R \in [\![RT]\!]\} \\
[\![\mathit{OptRecT}(l, T, RT)]\!] & \triangleq & [\![\mathit{RecT}(l, T, RT)]\!] \cup [\![RT]\!]
\end{array}
$$

Arrays and **Simplified Arrays**

$$
\begin{array}{lll}
\mathit{Domain} : \mathit{Sets}(\mathit{Lists}(\mathit{Values})) & & \\
[\![\mathit{EArrT}]\!] & \triangleq & \{[\,]\} \\
[\![\mathit{ArrT}(T, AT)]\!] & \triangleq & \{[V] :: A \mid V \in [\![T]\!],\ A \in [\![AT]\!]\} \\
[\![[T*]]\!] & \triangleq & [\![T]\!]^*
\end{array}
$$

Union types

$$
\begin{array}{lll}
[\![\epsilon]\!] & \triangleq & \emptyset \\
[\![T + U]\!] & \triangleq & [\![T]\!] \cup [\![U]\!]
\end{array}
$$

The basic idea behind our type fusion mechanism is that we always generalize the union of two record types to one record type containing the keys of both, and similarly for the union of two array types. We express this idea as 'merging types that have the same kind'. The following *kind*() function that maps each type to an integer ranging over $\{0, \ldots, 5\}$ is used to implement this approach.

In the sequel, generic types are indicated by the metavariables T, U, W, while BT, RT and AT are reserved for basic types, record types and array types.

Later on, in order to express correctness of the fusion process we rely on the usual notion of sub-typing (type inclusion).

Definition 4.1 (Sub-typing) *Let T and U be two types. Then T is a sub-type of U, denoted with $T <: U$, if and only if $[\![T]\!] \subseteq [\![U]\!]$.*

The sub-typing relation is a partial order among types. We don't use any subtype checking algorithm in this work, but we exploit this notion to state properties of our schema inference approach.

[1]The type ϵ is never used during type inference, since no value belongs to it. In greater detail, ϵ is actually a technical device that is only useful when an empty array type EArrT is simplified, before fusion, into a simplified array type: EArrT (that is, the type $[\,]$) is simplified as $[\epsilon*]$, which has the same semantics as EArrT, and our algorithms never insert ϵ in any other position.

$$
\begin{array}{rcll}
T & ::= & BT \mid RT \mid AT \mid SAT \mid \epsilon \mid T + T & \text{Top-level types} \\
BT & ::= & \texttt{Null} \mid \texttt{Bool} \mid \texttt{Num} \mid \texttt{Str} & \text{Basic types} \\
RT & ::= & ERecT \mid RecT(l, T, RT) \mid OptRecT(l, T, RT) & \text{Record types} \\
AT & ::= & EArrT \mid ArrT(T, AT) & \text{Array types} \\
SAT & ::= & [T*] & \text{Simplified array types}
\end{array}
$$

Notation:

$$
\begin{array}{lcl}
\{l_1 : T_1[?], \ldots, l_n : T_n[?]\} & \triangleq & [Opt]RecT(l_1, T_1, \ldots [Opt]RecT(l_n, T_n, ERecT)) \quad \textit{'?' is translated as 'Opt'} \\
[\,] & \triangleq & EArrT \\
[T_1, \ldots, T_n] & \triangleq & ArrT(T_1, \ldots ArrT(T_n, EArrT))
\end{array}
$$

Figure 3: Syntax of the JSON type language.

$$
\begin{array}{lcllcl}
kind(\texttt{Null}) & = & 0 & kind(\texttt{Str}) & = & 3 \\
kind(\texttt{Bool}) & = & 1 & kind(RT) & = & 4 \\
kind(\texttt{Num}) & = & 2 & kind(AT) = kind(SAT) & = & 5
\end{array}
$$

5. SCHEMA INFERENCE

As already said, our approach is based on two steps: i) type inference for each single value in the input JSON data collection, and ii) fusion of types generated by the first step. We present these steps in the following two sections.

5.1 Initial Schema Inference

The first phase of our approach consists of a Map phase that performs schema inference for each single value of the input collection. Type inference for single values is done according to the inference rules in Figure 4. Each rule allows one to infer the type of a value indicated in the conclusion (part below the line) in terms of types recursively determined in the premises (part above the line). Rules with no premises deal with the terminal cases of the recursive typing process, which infers the type of a value by simply reflecting the structure of the value itself. Note the particular case of record values where uniqueness of attribute keys l_i is checked. Also notice that these rules are deterministic: each possible value matches at most the conclusion of one rule. These rules, hence, directly define a recursive typing algorithm. The following lemma states soundness of value typing, and it can be proved by a simple induction.

Lemma 5.1 *For any JSON value V, $\vdash V \rightsquigarrow T$ implies $V \in [\![T]\!]$.*

It is worth noticing that schema inference done in this phase does not exploit the full expressivity of the schema language. Union types, optional fields and repetition types (the Simplified Array Types) are never inferred, while these types will be produced by the schema fusion phase described next.

5.2 Schema Fusion

The second phase of our approach is meant to fuse all the types inferred in the first Map phase. The main mechanism of this phase is a binary fusion function, that is commutative and transitive. These properties are crucial as they ensure that the function can be iteratively applied over n types in a distributed and parallel fashion.

When fusion is applied over two types T and U, it outputs either a single type obtained by recursively merging T and U if they have the same kind, or the simple union $T + U$ otherwise. Since fusion may result in a union type, and since this is in turn fused with other types, possibly obtained by fusion itself, the fusion function has to deal with the case when union types $T = T_1 + \ldots + T_n$ and $U = U_1 + \ldots + U_m$ need to be fused. In this case, our fusion function identifies and fuses types T_j and U_h with matching kinds, while types of non-matching kinds are just moved unchanged into the output union type. As we will see later, the fusion process ensures the invariant property that in each output union type a given kind may occur at most once in each union; hence, in the two union types above, $n \leq 6$ and $m \leq 6$, since we only have six different kinds.

The auxiliary functions *KMatch* and *KUnmatch*, defined in Figure 5, respectively have the purpose of collecting pairs of types of the same kind in two union-types T_1 and T_2, and of collecting non-matching types. In Figure 5, two similar functions *FMatch* and *FUnmatch* are defined. They identify and collect fields having matching/unmatched keys in two input body record types RT_1 and RT_2.

These two functions are based on the auxiliary functions $\circ(T)$ and $\diamond(RT)$. The functions $\circ(T)$ transforms a union type $T_1 + \ldots + T_n$ into the list of its non-union addends $[T_1, \ldots, T_n]$. The function $\diamond(RT)$ transforms a record type $\{(l_1{:}T_1)^{\mathtt{m}_1} + \ldots + (l_n{:}T_n)^{\mathtt{m}_n}\}$ into the set of its fields $[(l_1{:}T_1)^{\mathtt{m}_1} + \ldots + (l_n{:}T_n)^{\mathtt{m}_n}]$ — in this case we can use a set since no repetition is possible. Here we use $(l{:}T)^1$ do denote a mandatory field, $(l{:}T)^?$ do denote an optional field, and the symbols $\mathtt{m}$ and $\mathtt{n}$ for metavariables that range over $\{1, ?\}$.

We are now ready to present the fusion function. Its formal specification is given in Figure 6. We use function $\oplus L$ that is a right inverse of $\circ(T)$ and rebuilds a union type from a list of non-union addends, and function $\bigcirc S$ that is a right inverse of $\diamond(RT)$ and rebuilds a record type from a set of fields. We also use $min(\mathtt{m}, \mathtt{n})$ which is a partial function that picks the "smallest" cardinality, by assuming $? < 1$.

The general case where types T_1 and T_2 that may be union types have to be fused is dealt with by the $Fuse(T_1, T_2)$ function. According to what said before, it recursively applies *LFuse* to pairs of types coming from T_1 and T_2 and having the same kind, while unmatched types are simply returned in the output union type.

The specification of *LFuse* is captured by lines 2 to 7. Line 2 deals with the case where the input types are two identical basic types. In this case, the fusion yields the input basic type. Line 3 deals with the case where the input types are records. In this case, pairs of fields whose keys match are recursively fused by calling *LFuse*, the lowest cardinality

$$\frac{}{\vdash null \rightsquigarrow \texttt{Null}}\ (\textsc{TypeNull}) \qquad \frac{}{\vdash \textbf{true} \rightsquigarrow \texttt{Bool}}\ (\textsc{TypeTrueBool}) \qquad \frac{}{\vdash n \rightsquigarrow \texttt{Num}}\ (\textsc{TypeNumber}) \qquad \frac{}{\vdash s \rightsquigarrow \texttt{Str}}\ (\textsc{TypeString})$$

$$\frac{}{\vdash ERec \rightsquigarrow ERecT}\ (\textsc{TypeEmptyRec}) \qquad \frac{}{\vdash EArr \rightsquigarrow EArrT}\ (\textsc{TypeEmptyArray})$$

$$\frac{\vdash V \rightsquigarrow T \quad \vdash W \rightsquigarrow RT \quad l \notin \text{Keys}(RT)}{\vdash Rec(l,V,W) \rightsquigarrow RecT(l,V,RT)}\ (\textsc{TypeRec}) \qquad \frac{\vdash V \rightsquigarrow T \quad \vdash W \rightsquigarrow AT}{\vdash Arr(V,W) \rightsquigarrow ArrT(T,AT)}\ (\textsc{TypeArray})$$

Figure 4: Type inference rules.

$\circ(T)$: *transforms a type into a list of non-union types, where* $\cdot$ *is list concatenation and* [ ] *is the list constructor*

$$\begin{array}{lcl}
\circ(T_1 + T_2) & := & \circ(T_1) \cdot \circ(T_2) \\
\circ(\epsilon) & := & [\,] \\
\circ(T) & := & [T] \qquad \text{when } T \neq T_1 + T_2 \text{ and } T \neq \epsilon \\
KMatch(T_1, T_2) & := & \{(U_1, U_2) \mid U_1 \in \circ(T_1), U_2 \in \circ(T_2), kind(U_1) = kind(U_2)\} \\
KUnmatch(T_1, T_2) & := & \{U_1 \in \circ(T_1) \mid \forall U_2 \in T_2.\ kind(U_1) \neq kind(U_2)\} \\
 & & \cup \{U_2 \in \circ(T_2) \mid \forall U_1 \in \circ(T_1).\ kind(U_2) \neq kind(U_1)\}
\end{array}$$

$\diamond(RT)$: *transforms a record type into a set of fields*

$$\begin{array}{lcl}
\diamond(ERecT) & := & \emptyset \\
\diamond(RecT(l,T,RT)) & := & \{(l{:}T)^{1}\} \cup \diamond(RT) \\
\diamond(OptRecT(l,T,RT)) & := & \{(l{:}T)^{?}\} \cup \diamond(RT) \\
FMatch(RT_1, RT_2) & := & \{((l{:}T)^{\mathtt{n}}, (k{:}U)^{\mathtt{m}}) \mid (l{:}T)^{\mathtt{n}} \in \diamond(RT_1) \text{ and } (m{:}U)^{\mathtt{m}} \in \diamond(RT_2) \text{ and } l = k\} \\
FUnmatch(RT_1, RT_2) & := & \{(l{:}T)^{\mathtt{n}} \in \diamond(RT_1) \mid \forall (k{:}U)^{\mathtt{m}} \in \diamond(RT_2).\ l \neq k\} \cup \{(l{:}T)^{\mathtt{n}} \in \diamond(RT_2) \mid \forall (k{:}U)^{\mathtt{m}} \in \diamond(RT_1).\ l \neq k\}
\end{array}$$

Figure 5: Auxiliary functions.

is chosen for each, so that a field is mandatory only if is mandatory in both record types, whereas the unmatching fields are copied in the result type as optional fields.

The remaining lines of *LFuse* deal with the case where the input types are arrays. Each of these lines deals with a combination among original and simplified arrays by ensuring that *Fuse* is called over the body types of arrays that have been simplified through the call of *collapse*. While line 4 deals with the case that the two types have not been subject to fusion yet, lines 5-7 deal with the case that one of the input is the result of previous fusion operations, and therefore it has a *-expression as a body (recall the discussion in Section 2). Lines 8 and 9 are dedicated to the array simplification function *collapse*. This function simply rely on *Fuse* in order to generate an over-approximation of all the different types that are found in the original array type, in order to prepare the array type for the fusion process.

To illustrate both body array type simplification and record fusion, consider the following type T:

$$T = [\texttt{Num}, \texttt{Bool}, \texttt{Num},\ \{l_1 : \texttt{Num}, l_2 : \texttt{Str}\},\ \{l_1 : \texttt{Num}\},\ \{l_2 : \texttt{Bool}, l_3 : \texttt{Str}\}]$$

We have that $collapse(T)$ is equal to:

$$(\texttt{Num} + \texttt{Bool} + \{l_1 : \texttt{Num}, l_2 : \texttt{Str} + \texttt{Bool}, (l3 : \texttt{Str})?\})$$

Note that only one record type is created, by iterating fusion over the three record types. Also note that there is a good level of size reduction entailed by simplification. This happens in the most frequent cases (where elements of an array share most of their structure), while size reduction becomes weaker when very heterogeneous records appear in the array body type (in the particular case where no field key is shared among records, the unique record type given by simplification contains all keys, with their associated types, as optional fields).

To conclude this section the following theorems prove the main theoretical properties of the fusion process: correctness, commutativity and associativity. The crucial role played by these properties has already been discussed in the previous sections.

All these properties hold for types that respect the invariant that types of a given kind can occur at most once in each union. We use the term "normal types" to refer to such types. All of our algorithms respect this invariant, that is, they only generate normal types.

We first deal with correctness.

Theorem 5.2 (Correctness of *Fuse*) *Given two normal types* T_1 *and* T_2*, if* $T_3 = Fuse(T_1, T_2)$*, then* $T_1 <: T_3$ *and* $T_2 <: T_3$*.*

The proof of the above theorem relies on the following lemma.

Lemma 5.3 (Correctness of *LFuse*) *Given two non-union normal types* T_1 *and* T_2 *with the same kind, if* $T_3 = LFuse(T_1, T_2)$*, then it holds that* $T_1 <: T_3$ *and* $T_2 <: T_3$*.*

Another important property of fusion is commutativity.

$\oplus\ L$: *transforms a list of types into a union type, right inverse for* $\circ(T)$

$$
\begin{array}{lcll}
\oplus\ ([\,]) & := & \epsilon & \\
\oplus\ ([T]) & := & T & \\
\oplus\ ([T_1, T_2, \ldots, T_n]) & := & T_1 + \oplus\ ([T_2, \ldots, T_n]) & \text{when } n \geq 2
\end{array}
$$

$\bigcirc\ S$: *transforms a set of fields into a record type, right inverse for* $\diamond(RT)$

$$
\begin{array}{lcl}
\bigcirc\ (\emptyset) & := & ERecT \\
\bigcirc\ (\{(l{:}T)^1\} \cup S) & := & RecT(l, T, \bigcirc\ (S)) \\
\bigcirc\ (\{(l{:}T)^? \} \cup S) & := & OptRecT(l, T, \bigcirc\ (S))
\end{array}
$$

1. $Fuse(T_1, T_2) := \oplus\ (\{LFuse(U_1, U_2) \mid (U_1, U_2) \in KM\} \cup \{U_3 \mid U_3 \in KU\})$
   with $KM = KMatch(T_1, T_2),\ KU = KUnmatch(T_1, T_2)$
2. $LFuse(B, B) := B$ with $kind(B) < 4$
3. $LFuse(RT_1, RT_2) := \bigcirc\ (\{l{:}Fuse(T_1, T_2)^{min(\mathtt{m},\mathtt{n})} \mid ((l{:}T_1)^{\mathtt{m}}, (l{:}T_2)^{\mathtt{n}}) \in FM\} \cup \{(l{:}T)^? \mid (l{:}T)^{\mathtt{m}} \in FU\})$
   with $FM = FMatch(RT_1, RT_2),\ FU = FUnmatch(RT_1, RT_2)$
4. $LFuse(AT_1, AT_2) := [\,Fuse(collapse(AT_1), collapse(AT_2))*\,]$
5. $LFuse([T*], AT) := [\,Fuse(T, collapse(AT))*\,]$
6. $LFuse(AT, [T*]) := [\,Fuse(collapse(AT), T)*\,]$
7. $LFuse([T_1*], [T_2*]) := [\,Fuse(T_1, T_2)*\,]$
8. $collapse(EArrT) := \epsilon$
9. $collapse(ArrT(T, AT)) := Fuse(T, collapse(AT))$

Figure 6: The formal specification of the type fusion.

Theorem 5.4 (Commutativity) *The following two properties hold.*

1. *Given two normal types* T_1, T_2, *we have* $Fuse(T_1, T_2) = Fuse(T_2, T_1)$.
2. *Given two non-union normal types* T *and* U *having the same kind, we have* $LFuse(T, U) = LFuse(U, T)$.

Associativity of binary type fusion is stated by the following theorem.

Theorem 5.5 (Associativity) *The following two properties hold.*

1. *Given three normal types* T_1, T_2, *and* T_3, *we have*
$$Fuse(Fuse(T_1, T_2), T_3) = Fuse(T_1, Fuse(T_2, T_3))$$
2. *Given three non-union normal types* T, U *and* V *of the same kind, we have*
$$LFuse(LFuse(T, U), V) = LFuse(T, LFuse(U, V))$$

6. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of our approach whose main goal is to validate our precision and succinctness claims. We also incorporate a preliminary study on using our approach in a cluster-based environment for the sake of dealing with complex large datasets.

6.1 Experimental Setup and Datasets

For our experiments, we used Spark 1.6.1 [7] installed on two kinds of hardware. The first configuration consists in a single Mac mini machine equipped with an Intel dual core 2.6 Ghz processor, 16GB of RAM, and a SATA hard-drive. This machine is mainly used for verifying the precision and succinctness claims. In order to assess the scalability of our approach and its ability to deal with large datasets, we also exploited a small size cluster of six nodes connected using a Gigabit link with 1Gb speed. Each node is equipped with two 10-core Intel 2.2 Ghz CPUs, 64GB of RAM, and a standard RAID hard-drive.

The choice of using Spark is intuitively motivated by its widespread use as a platform for processing large datasets of different kinds (e.g., relational, semi-structured, and graph data). Its main characteristic lies in its ability to *persist* large datasets into main-memory in order to process them in a fast and efficient manner. Spark offers APIs for major programming languages like Java, Scala, and Python. In particular, Scala serves our case well since it makes the encoding of pattern matching and inductive definitions very easy. Using Scala has, for instance, allowed us to implement both the type inference and the type fusion algorithms in a rather straightforward manner starting from their respective formal specifications.

The type inference implementation extends the Json4s library [4] for parsing the input JSON documents. This library yields a specific Scala object for each JSON construct

(array, record, string, etc), and this object is used by our implementation to generate the corresponding type construct. The type fusion implementation follows a standard functional programming approach and does not need to be commented.

It is important to mention that the Spark API offers a feature for extracting a schema from a JSON document. However, this schema inference suffers from two main drawbacks. First, the inferred schemas do not contain regular expressions, which prevents one from concisely representing repeated types, while our type system uses the Kleene-Star to encode the repetition of types. Second, the Spark schema extraction is imprecise when it comes to deal with arrays containing *mixed content*, such as, for instance, an array of the form:

$$[\texttt{Num}, \texttt{Str}, \{l : \texttt{Str}\}]$$

In such a case, the Spark API uses *type coercion* yielding an array of type String only. In our case, we can exploit union types to generate a much more precise type:

$$[(\texttt{Num} + \texttt{Str} + \{l : \texttt{Str}\})*]$$

For our experiments we used four datasets. The first two datasets are borrowed from an existing work [13] and correspond to data crawled from GitHub and from Twitter. The third dataset consists in a snapshot of Wikidata [6], a large repository of facts feeding the Wikipedia portal. The last dataset consists in a crawl of NYTimes articles using the NYTimes API [5]. A detailed description of each dataset is provided in the sequel.

GitHub.

This dataset corresponds to metadata generated upon pull requests issued by users willing to commit a new version of code. It comprises 1 million JSON objects sharing the same top-level schema and only varying in their lower-level schema. All objects of this dataset consist exclusively of records, sometimes nested, with a nesting depth never greater than four. Arrays are not used at all.

Twitter.

Our second dataset corresponds to metadata that are attached to the tweets shared by Twitter users. It comprises nearly 10 million records corresponding, in majority, to tweet entities. A tiny fraction of these records corresponds to a specific API call meant to delete tweets using their ids. This dataset is interesting for our experiment for many reasons. First, it uses both records and arrays of records, although the maximum level of nesting is 3. Second, it contains five different top-level schemas sharing common parts. Finally, it mixes two kinds of JSON records (tweets and deletes). This dataset is useful to assess the effectiveness of our typing approach when dealing with arrays.

Wikidata.

The largest dataset comprises 21 million records reaching a size of 75GB and corresponding to Wikipedia *facts*. These facts are structured following a fixed schema, but suffer from a poor design compared to the previous datasets. For instance, an important portion of Wikidata objects corresponds to *claims* issued by users. These users identifiers are directly encoded as keys, whereas a clean design would suggest encoding this information as a value of a specific key called *id*, for example. This dataset can be of interest to our experiments since several records reach a nesting level of 6.

NYTimes.

The last dataset we are considering here is probably the most interesting one and comprises approximately 1.2 million records and reaches the size of 22GB. Its records feature both nested records and arrays and are nested up to 7 levels. Most of the fields in records are associated to text data, which explains the large size of this dataset compared to the previous ones. These records encode metadata about news articles, such as the headline, the most prominent keywords, the lead paragraph as well as a snippet of the article itself. The interest of this dataset lies in the fact that the content of fields is not fixed and varies from one record to another. A quick examination of an excerpt of this dataset has revealed that the content of the *headline* field is associated, in some records, to subfields labeled *main*, *content_kicker*, *kicker*, while in other records it is associated to subfields labeled *main* and *print_headlines*. Another common pattern in this dataset is the use of *Num* and *Str* types for the same field.

In order to compare the results of our experiments using the four datasets, we decided to limit the size of every dataset to the first million records (the size of the smallest one). We also created, starting from each dataset, subdatasets by restricting the original ones to respectively thousand (1K), ten thousands (10K) and one hundred thousands (100K) records chosen in a random fashion. Table 1 reports the size of each of these sub-datasets.

	1K	10K	100K	1M
GitHub	14MB	137MB	1.3GB	14GB
Twitter	2.2MB	22 MB	216MB	2.1GB
Wikidata	23MB	155MB	1.1GB	5.4GB
NYTimes	10MB	189MB	2GB	22GB

Table 1: (Sub-)datasets sizes.

6.2 Testing Scenario and Results

The main goal of our experiments is to assess the effectiveness of our approach and, in particular, to understand if it is able to return succinct yet precise fused types. To do so we report in Tables 2 to 5, for each dataset, the number of distinct types, the min, max and average size of these types as well as the size of the fused type. The notion of size of a type is standard, and corresponds to the size (number of nodes) of its Abstract Syntax Tree. For fairness, one can consider the average size as a baseline wrt which we compare the size of the fused type. This helps us judge the effectiveness of our fusion at collapsing common parts of the input types.

From Tables 2, 3 and 4, it is easy to observe that our primary goal of succinctness is achieved for the GitHub and the Twitter datasets. Indeed, the ratio between the size of the fused type and that of the average size of the input types is not bigger than 1.4 for GitHub whereas it is bounded by 4 for Twitter, which are relatively good factors. These results are not surprising: GitHub objects are homogeneous. Twitter has a more varying structure and, in addition, it mixes two different kinds of objects that are deletes and tweets, as outlined in the description of this dataset. This explains the slight difference in terms of compaction wrt GitHub.

As expected, the results for Wikidata are worse than the results for the previous datasets, due to the particularity of this dataset concerning the encoding of user-ids as keys. This has an impact on our fusion technique, which relies on keys to merge the underlying records. Still, our fusion algorithm manages to collapse the common parts of the input types as testified by the fact that the size of the fused types is smaller than the sum of the input types.[2] Finally, the results for NYtimes dataset, which features many irregularities, are promising and even better than the rest. This can be explained by the fact that the fields in the first level are fixed while the lower level fields may vary. This does not happen in the previous datasets, where the variations occur on the first level.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	29	147	305	233	321
10K	66	147	305	239	322
100K	261	147	305	246	330
1M	3,043	147	319	257	354

Table 2: Results for GitHub.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	167	7	218	74	221
10K	677	7	276	75	273
100K	2,320	7	308	75	277
1M	8,117	7	390	77	299

Table 3: Results for Twitter.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	999	27	36,748	1,215	37,258
10K	9,886	21	36,748	866	82,191
100K	95,298	11	39,292	607	87,290
1M	640,010	11	39,292	310	117,010

Table 4: Results for Wikidata.

Execution times for the type inference and the type fusion for GitHub, Twitter, and Wikidata datasets are reported in Table 6. As it can be observed, processing the Wikidata dataset is more time-consuming than processing the two other datasets. This is explained, once again, by the nature of the Wikidata dataset. Observe also that the processing time of GitHub is larger than that of Twitter due to the size of the former dataset that is larger than the latter one.

6.3 Scalability

To assess the scalability of our approach, we have deployed the typing and the fusion implementations on our cluster. We set the parameters so to exploit the full capacity of the cluster in terms of number of cores. To do so, we set the number of cores to 120, that is, 20 cores per node. We

[2]The total size of input types can be roughly estimated by multiplying either the minimum, maximum or average size with the number of types.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	555	299	887	597.25	88
10K	2,891	6	943	640	331
100K	15,959	6	997	755	481
1M	312,458	6	1,046	674	760

Table 5: Results for NYTimes

	1K	10K	100K	1M
GitHub	1s	4s	32s	297s
Twitter	0	1s	7s	73s
Wikidata	7s	15s	121s	925s

Table 6: Typing execution times.

also assign to our job 300GB of main memory, hence leaving 72GB for the task manager and other runtime monitoring processes. We used the NYTimes full dataset (22GB) stored on HDFS. Because our approach requires two steps (type inference and type fusion), we adopted a strategy where the results of the type inference step are persisted into main-memory to be directly available to the fusion step. We ran the experiments on datasets of varying size obtained by restricting the full one to the first fifty, two hundred-fifty and five hundred thousands records, respectively. The results for these experiments are reported in Table 7 together with some statistics on these datasets (number of records and cardinality of the distinct types). It can be observed that execution increases linearly with the dataset size.

size	# records	# distinct types	time
1GB	50,000	5,679	2 min
4.5GB	250,000	54,868	4.4 min
9GB	500,000	128,943	8.5 min
22GB	1,184,943	312,458	12.5 min

Table 7: Scalability - NYTimes dataset.

In an attempt to optimize the execution time on the cluster, we started by analyzing the execution and realized that the full capacity of the cluster was not exploited. Indeed, the HDFS uses only one node to store the entire dataset, which does not allow the parallelism to be exploited. We also observed that the intermediate results produced by the type inference step were split on only two nodes. The overall effect is that the computation was performed on two nodes while the remaining four nodes were idle.

To overcome this problem, we considered a strategy based on partitioning the input data that would force Spark to take full advantage of the cluster. In order to avoid the overhead of data shuffling, the ideal solution would be to force computation to be local until the end of the processing. Because Spark 1.6 does not explicitly allow such an option we had to opt for a manual strategy where each partition of data is processed in isolation, and each of the inferred schema is finally fused with the others (this is a fast operation as each schema to fuse has a very small size). The purpose is to simulate the realistic situation where Spark processes data exclusively locally thus avoiding the overhead of synchronization. The times for processing each partition are reported in Table 8. The average time is 2.85 minutes, which is a rather reasonable time for processing a dataset of 22

GB.

	# objects	# types	time
partition 1	284,943	67,632	2.4 min
partition 2	300,000	83,226	3.8 min
partition 3	300,000	89,929	1.9 min
partition 4	300,000	84,333	3.3 min

Table 8: Parition-based processing of NYTimes.

Note that this simple yet effective optimization is possible thanks to the associativity of our fusion process.

7. CONCLUSIONS AND FUTURE WORK

The approach described in this paper is a first step towards the definition of a schema-based mechanism for exploring massive JSON datasets. This issue is of great importance due to the overwhelming quantity of JSON data manipulated on the web and due to the flexibility offered by the systems managing these data.

The main idea of our approach is to infer schemas for the input datasets in order to get insights about the structure of the underlying data; these schemas are succinct yet precise, and faithfully capture the structure of the input data. To this end, we started by identifying a schema language with the operators needed to ensure succinctness and precision of our inferred schemas. We, then, proposed a fusion mechanism able to detect and collapse common parts of the input types. An experimental evaluation on several datasets validated our claims and showed that our type fusion approach actually achieves the goals of succinctness, precision, and efficiency.

Another benefit of our approach is its ability to perform type inference in an incremental fashion. This is possible because the core of our technique, fusion, is incremental by essence. One possible and interesting application would be to process a subset of a large dataset to get a first insight on the structure of the data before deciding whether to refine this partial schema by processing additional data.

In the near future we plan to enrich schemas with statistical and provenance information about the input data. Furthermore, we want to improve the precision of the inference process for arrays and study the relationship between precision and efficiency.

8. ACKNOWLEDGMENTS

Houssem Ben Lahmar has been partially supported by the project D03 of SFB/Transregio 161 [3].

9. REFERENCES

[1] The JSON Query Language. `http://www.jsoniq.org`.
[2] Json schema definition language. `http://jsoniq.org/docs/JSound/html-single/`.
[3] Json schema language. `http://json-schema.org`.
[4] Json4s library. `http://json4s.org`.
[5] Nytimes api. `https://developer.nytimes.com/`.
[6] Wikidata. `https://dumps.wikimedia.org/wikidatawiki/entities/`.
[7] Apache Spark, 2016. `http://spark.apache.org`.
[8] M.-A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani. Schema inference for massive JSON datasets. Technical report available at `http://webia.lip6.fr/baazizi/research/json/full.pdf`.
[9] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyên. Type-based xml projection. VLDB '06, pages 271–282, 2006.
[10] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.
[11] S. Cebiric, F. Goasdoué, and I. Manolescu. Query-oriented summarization of RDF graphs. *PVLDB*, 8(12):2012–2015, 2015.
[12] D. Colazzo, G. Ghelli, and C. Sartiani. Typing massive json datasets. In *XLDI '12, Affiliated with ICFP*, 2012.
[13] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. SIGMOD '16, pages 295–310, 2016.
[14] D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and dtds. *Theory Comput. Syst.*, 57(4):1114–1158, 2015.
[15] Z. H. Liu, B. Hammerschmidt, and D. McMahon. Json data management: Supporting schema-less development in rdbms. SIGMOD '14, pages 1247–1258, 2014.
[16] J. McHugh and J. Widom. Query optimization for xml. VLDB '99, pages 315–326. Morgan Kaufmann Publishers Inc., 1999.
[17] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, Nov. 2005.
[18] S. Nestorov, S. Abiteboul, and R. Motwani. Infering structure in semistructured data. *SIGMOD Record*, 26(4):39–43, 1997.
[19] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 295–306. ACM Press, 1998.
[20] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of json schema. WWW '16, pages 263–273, 2016.
[21] S. Scherzinger, E. C. de Almeida, T. Cerqueus, L. B. de Almeida, and P. Holanda. Finding and fixing type mismatches in the evolution of object-nosql mappings. In T. Palpanas and K. Stefanidis, editors, *Proceedings of the Workshops of the EDBT/ICDT 2016*, volume 1558 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.
[22] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz. Schema management for document stores. *Proc. VLDB Endow.*, 8(9):922–933, May 2015.

[3] http://www.trr161.de/interfak/forschergruppen/sfbtrr161