Schemas And Types For JSON Data

Mohamed-Amine Baazizi[1] Dario Colazzo[2] Giorgio Ghelli[3] Carlo Sartiani[4]

22nd International Conference on Extending Database Technology, March 26-29, 2019

[1]LIP6 - Sorbonne Université

[2]LAMSADE - Université Paris-Dauphine, PSL Research University

[3]Dipartimento di Informatica - Università di Pisa

[4]DIMIE - Università della Basilicata

Outline

- JSON Primer ($\sim$ 10 min)
- Schema Languages ($\sim$ 20 min)
- Types in Programming Languages ($\sim$ 15 min)
- Schema Tools ($\sim$ 30 min)
  - Schema Inference Tools
  - Parsing Tools
- Future Opportunities ($\sim$ 10 min)

JSON Primer

Introduction

JavaScript Object Notation

- JSON is a data format mixing the flexibility of semistructured models and traditional data structures like records and ordered sequences (*arrays*)
- Born as a subset of the JavaScript object language [2]
  - Now fully independent
  - No support for JavaScript complex data structures like Maps, Sets, and Typed Arrays
  - U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR are legal in JSON but not in JavaScript
- It offers a syntax for booleans, numbers, strings, records, and arrays

JSON Grammar

$$
\begin{array}{rlrl}
J ::= & B \mid R \mid A & & \text{JSON expressions} \\
B ::= & \mathrm{null} \mid \mathrm{true} \mid \mathrm{false} \mid n \mid s & n \in \text{Number},\ s \in \text{String} & \text{Basic values} \\
R ::= & \{l_1 : J_1, \ldots, l_n : J_n\} & n \geq 0 & \text{Records} \\
A ::= & [J_1, \ldots, J_n] \quad n \geq 0 & & \text{Arrays}
\end{array}
$$

Basic Values

- A string is a UTF-8 string surrounded by quotation marks
    - `"Cat"`
- A number is represented in base 10 using decimal digits
    - It comprises an integer part prefixed by an optional minus sign, and followed by an optional fractional part and/or an optional exponent part
    - 90210, -3.141, 17.17E4
- *null*, *true*, and *false* are just predefined literals

Records

- A JSON record is a sequence of *zero* or more name/value pairs (*members*) surrounded by curly braces
    - A name is just a string
    - A value can be anything
- A record can be empty: {}
- A record can contain multiple members with the same name
    - Member labels are not required to be unique [3]
    - Very bad practice [20]

Arrays

- An array is a sequence of *zero* or more comma-separated elements, surrounded by square brackets
- Array elements can be any JSON value
    - [162, 185]
    - [{"*id*" : 1039608069599240194, . . .

Constraints

- There are almost no constraints on JSON data
- Member labels are not required to be unique [3]
    - Very bad practice [20]
- Records and arrays can be empty
- Numbers can be almost everything
- The only real requirement is the use of UTF-8

Uses

JSON is prominently used for data interchange

- Communication between web apps and remote servers
- Publishing open data
    - The U.S. Government's open data platform: `https://www.data.gov`
- Publishing scientific data
    - `https://minorplanetcenter.net/data`
- Web API
    - `https://developer.nytimes.com`, `https://twitter.com`, etc.

A Sample Dataset

New York Times

- A dataset where each line contains a JSON object representing the metadata of an article
- Obtained by invoking the web API of `https://developer.nytimes.com`
  - Objects may be nested
  - The same field in different instances may have a very different structure

Schema Languages

Schemas for JSON

- When working with any data format an important aspect is being able to:
    - specify the structure of valid documents via a schema
    - efficiently checking that a document is valid wrt the schema
- Main desiderata for a schema language:
    - schemas should be easy to define/read/understand
    - high expressivity
    - efficient checking of main properties: non-emptiness, schema inclusion, document validity, query correctness.
- Proposals of schema languages in these directions exist, we focus on JSON Schema and Joi.
- By relying of several examples.

JSON Schema

Records are described by JSON object values of the form

```
{
"type" : "object",
"properties"  : { ......  }
}
```

Open record assumption - for instance the type of records possibly having ”a” and/or ”b” fields of type string

```
{
"type": "object",
"properties" : { "a : { "type" : "string" }, "b" : { "type" : "string" }
    }
}
```

JSON Schema

Records are typically described by JSON object values of the form

```
{
"type" : "object",
"properties"  : { ......  }
}
```

The type of records *only* having ”a” and ”b” fields of type string

```
{
"type" : "object",
"properties" : { "a" : { "type" : "string" }, "b" : { "type" : "string" }
          }
"additionalProperties"  : false
"required"  :  [  "a" ,  "b"  ]
        }
}
```

JSON Schema

A more complex example now, related to a JSON data fragment coming from New York Times.

- The byline field can either
    - have value Null, or
    - have an object as value, where ”person” field of the is an empty array if the ”organisation” field is present,
    - otherwise ”person” is a non empty array of records (with fields ”fn”, ”sn”, etc.)

A JSON Schema for NYT byline information

```
{
"definitions" : {
   "S1": ....case with organisation field...
   "S2":  ....case without organisation field...
}
.......
{
"type": "object",
"properties"  :  { "byline":  {
                           "anyOF" : [
                                "enum": [null],
                                "$ref" : "#/definitions/S1",
                                "$ref" : "#/definitions/S2" ]
                           }
                     }
        }
}
```

A JSON Schema for NYT fragment - S1

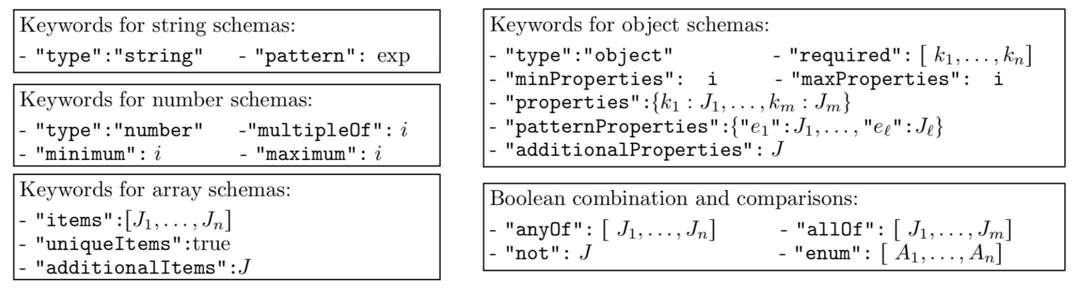
```
{
"type" : "object",
"properties"  :  { "contributor" : { "type" : "string" },
                   "organization" : { "type" : "string" },
                   "original" : { "type" : "string" },
                   "person" : { "type" : "array",
                              "maxItems" : 0 } ,
"additionalProperties"  :  false
"required"  :  [ "contributor", "organization", "original", "person" ]
}
```

A JSON Schema for NYT fragment - S2

```
{ "type" : "object",
  "properties"  :  {
        "contributor" : { "type" : "string" },
        "original" : { "type" : "string" },
        "person" : { "type" : "array",
                     "minItems": 1,
                     "items"  : [ { "type" : "object",
                                    "properties"  :  {
                                        "fn" : { "type" : "string" },
                                        "ln" : { "type" : "string" },
                                        "mn" : { "type" : "string" },
                                        "org" : { "type" : "string" } },
                                    "additionalProperties" : false } ] }
                                        }
  "additionalProperties"  : false ,
  "required"   : ["contributor", "original", "person"]
}
```

JSON schema

- Main schema language for JSON, standardisation efforts are in progress [11].
- Formal semantics and study done in [29, 19], from which we borrow subsequent examples.
- Main properties in a nutshell [19]:



Keywords for string schemas:

- "type":"string" - "pattern": exp

Keywords for number schemas:

- "type":"number" -"multipleOf": i
- "minimum": i - "maximum": i

Keywords for array schemas:

- "items":$[J_1, \ldots, J_n]$
- "uniqueItems":true
- "additionalItems":J

Keywords for object schemas:

- "type":"object" - "required": $[\, k_1, \ldots, k_n]$
- "minProperties": i - "maxProperties": i
- "properties":$\{k_1 : J_1, \ldots, k_m : J_m\}$
- "patternProperties":{"e_1":$J_1, \ldots,$ "e_ℓ":J_ℓ}
- "additionalProperties": J

Boolean combination and comparisons:

- "anyOf": $[\, J_1, \ldots, J_n]$ - "allOf": $[\, J_1, \ldots, J_m]$
- "not": J - "enum": $[\, A_1, \ldots, A_n]$

Object schemas

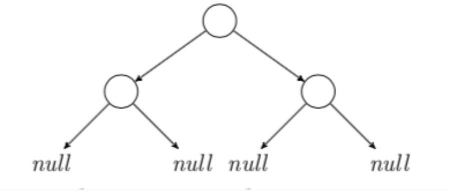
```
{ "type" : "object",
  "properties"  :  { "name" : { "type":"string"} },
  "patternProperties" : {
                "a(b|c)a" : { "type" : "number", "multipleOf" : 2} },
  "additionalProperties" : {
               "type": "number",
               "minimum" : 1,
               "maximum" : 1 }
            }
}
```

Arrays

```
{
"type" : "array",
"items"  : [ { "type" : "string" }, { "type" : "string" } ],
"additionalItems"  : { "type" : "number" }, "uniqueItems" : true
}
```

Boolean operators, recursion and path expressions

```
{
"definitions"  : {
   "S": {
    "anyOf"  : [
      {"enum": [null]},
      {"allOf" : [
          {"type": "array",
           "minItems" : 2,
           "maxItems" : 2,
           "items" : [
             {"$ref" : "#/definitions/S"},
             {"$ref" : "#/definitions/S"}]
          },
          {"not" : {"type": "array",
                    "uniqueItems" : true}}
       ]}
]},}}
```



.....

Complexity of validation

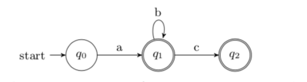
- Validation is the problem of checking whether a given JSON document J conforms to a given JSON schema S, noted as:

$$J \models S$$

- A simple validation algorithm can be devised with complexity bound by $O(|S| * |J|)$, provided that `uniqueItems` is not used.
- Otherwise validation can be performed in $O(|S| * log(|J|) * |J|)$ time
- So validation is in PTIME, and proved to be PTIME-hard actually [29].

Expressivity: JSON Schema is inherently as expressive as NFAs

- JSON string encoding, e.g., ”abbc” → `{"a":{"b":{:{"b":{"c": Null}}}}`.



```
{
   "definitions": {
      "q0": {
         "type":"object",
         "properties": {
            "a": {"$ref": "#/definitions/q1"}
            },
          "additionalProperties": false},
      "q1": {
         "anyOf": [
            {"enum": [null]},
            {"type": "object",
             "properties": {
                "b": {"$ref": "#/definitions/q1"},
                "c": {"$ref": "#/definitions/q2"}
            },
            "additionalProperties": false}
         ]},
      "q2": {"enum": [null]}
  },
  "$ref": "#/definitions/q0"
}
```

- As stated in [29], this construction can be generalised to *tree automata*
- Negative consequence: checking consistency is EXPTIME-hard.
- Future research: finding meaningful fragments with better complexity.

Joi

Main features

- Joi is a powerful schema language to describe and check at run-time properties of JSON objects exchanged over the Web and that Web applications expect, especially server-side ones.
- Large intersection with JSON Schema
- But more fluent and readable code

Joi

```
Joi = require('joi');
const schema = Joi.string().min(6).max(10);
const updatePassword = function (password) {
                                    Joi.assert(password, schema);
                                    console.log('Validation success!');
};

updatePassword('password');
```

Joi in action

Important: *closed record assumption*

```
const Joi = require('joi');

const schema = Joi.object().keys({
    username: Joi.string().alphanum().min(3).max(30).required(),
    password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}\$/),
    access_token: [Joi.string(), Joi.number()],
    birthyear: Joi.number().integer().min(1900).max(2013),
    email: Joi.string().email({ minDomainAtoms: 2 })
}).with('username', 'birthyear').without('password', 'access_token');
```

Joi in action

Important: *closed record assumption*

```
const Joi = require('joi');

const schema = Joi.object().keys({
    username: Joi.string().alphanum().min(3).max(30).required(),
    password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}\$/),
    access_token: [Joi.string(), Joi.number()],
    birthyear: Joi.number().integer().min(1900).max(2013),
    email: Joi.string().email({ minDomainAtoms: 2 })
}).with('username', 'birthyear').without('password', 'access_token');
```

Add .unknown() for enabling open record semantics.

Back to our NYT schema fragment

```
const Joi = require('joi');
const byline-with-organisation = Joi.object().keys(.......)
const byline-wo-organisation = Joi.object().keys(.......)
const docSchema = Joi.alternative().try(
                                      Joi.any().valid(null),
                                      byline-with-organisation,
                                      byline-wo-organisation
                                    )
```

JSON Schema vs Joi

JSON Schema	Joi
open record types	closed record types
better documented	many use cases available on the web, but poor documentation
language independent	bound to Java Script (but translators exists)
more verbose, expressed in JSON	more fluent to write/read
full support for union, disjunction, negation	limited support (works needs to be done to fix boundaries)
limited expressive power for expressing properties of base values	much more expressive

Conclusive remarks on schemas

- We focused on JSON Schema and Joi
- other proposals exists, like JSound, but with much less impact
- work still needed in the standardisation, documentation and specification of formal semantics
- we are currently focusing on a deep and formal comparison between JSON Schema and Joi

Types in Programming Languages

Typing JSON Data in a Programming Language

- JSON is just nesting of objects and arrays, supported by any type system
- We consider Typescript as an example

Types for JSON Data in Typescript

- Basic types:

Types for JSON Data in Typescript

- Basic types:
  - boolean, number, string, null

Types for JSON Data in Typescript

- Basic types:
    - **boolean**, **number**, **string**, **null**
    - **enum**
        - **enum** Color Red = 1, Green, Blue;
        - type *Color* is the set {1, 2, 3}

Types for JSON Data in Typescript

- Basic types:
  - **boolean**, **number**, **string**, **null**
  - **enum**
    - **enum** Color Red = 1, Green, Blue;
    - type *Color* is the set {1, 2, 3}
  - **symbol**

Types for JSON Data in Typescript

- Basic types:
  - **boolean**, **number**, **string**, **null**
  - **enum**
    - **enum** Color Red = 1, Green, Blue;
    - type *Color* is the set {1, 2, 3}
  - **symbol**
  - Trivial types, apart from **null**, : **any**, **void**, **undefined**, **never**

Types for JSON Data in Typescript

- Basic types:
    - **boolean**, **number**, **string**, **null**
    - **enum**
        - **enum** Color Red = 1, Green, Blue;
        - type *Color* is the set {1, 2, 3}
    - **symbol**
    - Trivial types, apart from **null**, : **any**, **void**, **undefined**, **never**
- Array types:

Types for JSON Data in Typescript

- Basic types:
  - **boolean**, **number**, **string**, **null**
  - **enum**
    - **enum** Color Red = 1, Green, Blue;
    - type *Color* is the set {1, 2, 3}
  - **symbol**
  - Trivial types, apart from **null**, : **any**, **void**, **undefined**, **never**
- Array types:
  - Repetition array types: elemtype**[ ]** (or: **Array**<elemtype>)

Types for JSON Data in Typescript

- Basic types:
  - **boolean**, **number**, **string**, **null**
  - **enum**
    - **enum** Color Red = 1, Green, Blue;
    - type *Color* is the set {1, 2, 3}
  - **symbol**
  - Trivial types, apart from **null**, : **any**, **void**, **undefined**, **never**
- Array types:
  - Repetition array types: elemtype**[ ]** (or: **Array**<elemtype>)
  - Tuple array types: **[** elemtype$_1$, ..., elemtype$_n$ **]**

Types for JSON Data in Typescript

- Basic types:
    - **boolean**, **number**, **string**, **null**
    - **enum**
        - **enum** Color Red = 1, Green, Blue;
        - type *Color* is the set {1, 2, 3}
    - **symbol**
    - Trivial types, apart from **null**, : **any**, **void**, **undefined**, **never**
- Array types:
    - Repetition array types: elemtype**[ ]** (or: **Array**<elemtype>)
    - Tuple array types: **[** elemtype$_1$, ..., elemtype$_n$ **]**
        - A coordinate pair: **[number, number]**
        - A list of coordinate pairs: **Array<[number, number]>** (i.e. **[number, number] [ ]**)

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
    - $\{\text{key}_1 : \text{type}_1, \ldots, \text{key}_n : \text{type}_n\}$: describes any object that has at least those fields.
        - e.g.:{ name: **string** }

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - $\{key_1 : type_1, \ldots, key_n : type_n\}$: describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - $\{\text{key}_1 : \text{type}_1, \ldots, \text{key}_n : \text{type}_n\}$: describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }
- Optional fields:

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - $\{key_1 : type_1, \ldots, key_n : type_n\}$: describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }
- Optional fields:
  - **interface** SquareConfig { color: **string**, width**?**: **number** }

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - $\{key_1 : type_1, \ldots, key_n : type_n\}$: describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }
- Optional fields:
  - **interface** SquareConfig { color: **string**, width**?**: **number** }
  - If a *width* is present, its type is **number**

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - {$\text{key}_1 : \text{type}_1, \ldots, \text{key}_n : \text{type}_n$} : describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }
- Optional fields:
  - **interface** SquareConfig { color: **string**, width**?**: **number** }
  - If a *width* is present, its type is **number**
  - The extraction of a *width* field from a *SquareConfig* object is legal

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - $\{key_1 : type_1, \ldots, key_n : type_n\}$: describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }
- Optional fields:
  - **interface** SquareConfig { color: **string**, width**?**: **number** }
  - If a *width* is present, its type is **number**
  - The extraction of a *width* field from a *SquareConfig* object is legal
- Interfaces can be defined by inheritance

JSON object types in Typescript

- *Interface* object types - structural, transparent, open-ended:
  - {key_1 : $type_1$,..., key_n : $type_n$} : describes any object that has at least those fields.
    - e.g.:{ name: **string** }
  - Interface declaration is just a shorthand (structural typing)
    - e.g.: **interface** NamedValue { name: **string** }
- Optional fields:
  - **interface** SquareConfig { color: **string**, width**?**: **number** }
  - If a *width* is present, its type is **number**
  - The extraction of a *width* field from a *SquareConfig* object is legal
- Interfaces can be defined by inheritance
- **readonly** properties, **ReadonlyArray**

Advanced types in Typescript

- Intersection types T & U
  - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }

Advanced types in Typescript

- Intersection types T & U
  - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }
- Union types T | U
  - { name: **string** } | { age: **number** }= ?

Advanced types in Typescript

- Intersection types T & U
    - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }
- Union types T | U
    - { name: **string** } | { age: **number** }= ?
- Union types with enumerations can simulate *discriminated union types*

Advanced types in Typescript

- Intersection types T & U
    - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }
- Union types T | U
    - { name: **string** } | { age: **number** }= ?
- Union types with enumerations can simulate *discriminated union types*
    - **enum** Role { Consultant, Employee };
    - { role: Role.Consultant, fee: **number**} | { role: Role.Employee, salary: **number**}

Advanced types in Typescript

- Intersection types T & U
    - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }
- Union types T | U
    - { name: **string** } | { age: **number** }= ?
- Union types with enumerations can simulate *discriminated union types*
    - **enum** Role { Consultant, Employee };
    - { role: Role.Consultant, fee: **number**} | { role: Role.Employee, salary: **number**}
- Recursive types

Advanced types in Typescript

- Intersection types T & U
    - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }
- Union types T | U
    - { name: **string** } | { age: **number** }= ?
- Union types with enumerations can simulate *discriminated union types*
    - **enum** Role { Consultant, Employee };
    - { role: Role.Consultant, fee: **number**} | { role: Role.Employee, salary: **number**}
- Recursive types
- Type-level computations:

Advanced types in Typescript

- Intersection types T & U
    - { name: **string** } & { age: **number** }= { name: **string**, age: **number** }
- Union types T | U
    - { name: **string** } | { age: **number** }= ?
- Union types with enumerations can simulate *discriminated union types*
    - **enum** Role { Consultant, Employee };
    - { role: Role.Consultant, fee: **number**} | { role: Role.Employee, salary: **number**}
- Recursive types
- Type-level computations:
    - Generics: <T> (arg: T): T
    - **keyof** Person : enumeration type with all keys of *Person*
    - Person[“name”] : the type of p[“name”] when *p* is a *Person*
    - Iterations or conditions on types:
        - **type** Partial<T> = { [P in **keyof** T]?: T[P]; }
        - T **extends** U ? X<T> : Y<T>

NYTimes JSON data in Typescript

{ docs: { byline: **null**

NYTimes JSON data in Typescript

```
{ docs: { byline: null
                | { contributor: string,
                    organization: string,
                    original: string,
                    person: [ ]
                  }
                |
```

NYTimes JSON data in Typescript

```
{ docs: { byline: null
                | { contributor: string,
                    organization: string,
                    original: string,
                    person: [ ]
                  }
                | { contributor: string,
                    original: string,
                    person: Array< {fn?: string, ln?: string, mn?: string, org?: string} >
                  }
        }
}
```

NYTimes JSON data in Typescript

```
{ docs:
    { byline: null
```

NYTimes JSON data in Typescript

```
{ docs:
    { byline:  null
               | { contributor: string, original: string }
                 &
```

NYTimes JSON data in Typescript

```
{ docs:
   { byline:  null
              | { contributor: string, original: string }
                &
                ( { organization: string, person: [ ] }
                  |
```

NYTimes JSON data in Typescript

```
{ docs:
    { byline:  null
               | { contributor: string, original: string }
                 &
                 ( { organization: string, person: [ ] }
                   |
                   { person: Array< {fn?: string, ln?: string, mn?: string, org?: string} > }
                 )
    }
}
```

JSON types in Typescript

- Arrays and interfaces model the essential JSON features

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data
- Typescript has a rich type algebra, mostly used to type functions

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data
- Typescript has a rich type algebra, mostly used to type functions
- We miss:
    - Closed object types

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data
- Typescript has a rich type algebra, mostly used to type functions
- We miss:
    - Closed object types
    - Negation

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data
- Typescript has a rich type algebra, mostly used to type functions
- We miss:
    - Closed object types
    - Negation
    - Patterns for strings and keys, facets for numbers

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data
- Typescript has a rich type algebra, mostly used to type functions
- We miss:
  - Closed object types
  - Negation
  - Patterns for strings and keys, facets for numbers
  - min/maxProperties for objects and arrays

JSON types in Typescript

- Arrays and interfaces model the essential JSON features
- Union types and optional fields allow one to express semi-structured data
- Typescript has a rich type algebra, mostly used to type functions
- We miss:
    - Closed object types
    - Negation
    - Patterns for strings and keys, facets for numbers
    - min/maxProperties for objects and arrays
    - ...

Schema Tools

Schema Tools

Schema Inference Tools

Overview

- Inferring *descriptive* schemas for JSON
- Prior work on semi-structured data [25, 28] and XML [24, 18]
- Summarization of the structure [32], outlier detection [30], generation of a normalized relational schema [22], distributed schema inference [15, 16, 17, 21], schema-based classification [23]
- System-related techniques: Spark [1], Flink [8], MongoDB [12], Couchbase [10], PostgreSQL [13], Apache Drill [7]

Overview

- Inferring *descriptive* schemas for JSON
- Prior work on semi-structured data [25, 28] and XML [24, 18]
- Summarization of the structure [32], outlier detection [30], generation of a normalized relational schema [22], **distributed schema inference [15, 16, 17, 21]**, schema-based classification [23]
- System-related techniques: **Spark [1]**, Flink [8], **MongoDB [12], Couchbase [10]**, PostgreSQL [13], Apache Drill [7]

Distributed schema inference approaches

- Main goal: infer a schema describing massive JSON datasets
- Many variants
    - schemas reflecting *structural* information only [15] (EDBT’2017)
    - schemas with *cardinality* information [16] (DBPL’2017)
    - schema with a controlled level of precision [17] (VLDBJ’2019)

Inferring schemas reflecting structural information (EDBT'2017)

- Infer information about:
  - fields in records, indicate whether optional or mandatory
  - content of arrays
  - structural variety
- Designed in Map-Reduce to process large datasets efficiently
  - Input: a collection $J_1, \ldots, J_n$
  - Map phase: infer the schema S_i for each J_i
  - Reduce phase: combine the S_is into a single schema S describing the entire collection
    commutative and *associative* operation

Illustration of EDBT'2017

```
 {byline:
  {contributor:"..",
   organization:"..",
   original:"..",
   person:[ ]
  }
 }

 {byline:null}

{byline:
 {contributor:"..",
  original:"..",
  person:[
   {fn:"..",ln:".."},
   {mn:"..",org:"..."}
  ]
 }
}
```

Input collection

Illustration of EDBT'2017

```
 {byline:
  {contributor:"..",
   organization:"..",
   original:"..",
   person:[ ]
  }
 }

  {byline:null}

{byline:
 {contributor:"..",
  original:"..",
  person:[
   {fn:"..",ln:".."},
   {mn:"..",org:"..."}
  ]
 }
}
```

`Input collection`

Map

```
{byline:
 {contributor:Str,
  organization:Str,
  original:Str,
  person:[ ]
 }
}

 {byline:Null}

 {byline:
  {contributor:Str,
   original:Str,
   person:[
    {fn?:Str,ln?:Str,
    mn?:Str,org?:Str}
   ]
  }
 }
```

Illustration of EDBT'2017

```
{byline:
 {contributor:"..",
  organization:"..",
  original:"..",
  person:[ ]
 }
}

 {byline:null}

{byline:
 {contributor:"..",
  original:"..",
  person:[
   {fn:"..",ln:".."},
   {mn:"..",org:"..."}
  ]
 }
}
```

Map

```
{byline:
 {contributor:Str,
  organization:Str,
  original:Str,
  person:[ ]
 }
}

 {byline:Null}

 {byline:
  {contributor:Str,
   original:Str,
   person:[
    {fn?:Str,ln?:Str,
    mn?:Str,org?:Str}
   ]
  }
 }
```

Reduce

```
{byline:
 Null+
 {contributor:Str,
  organization?:Str,
  original:Str,
  person:[{fn?:Str,ln?:Str,
     mn?:Str,org?:Str}]
 }
}
```

Input collection

Inferred schema

Inferring schemas with cardinality information (DBPL’2017)

- Enrich schema with statistical information
  - how often a field appears
  - how many items in each branch of a union
  - how many items in an array
- Extend [15] with a counting mechanism

`{byline:`
`Null`^{10}`+`
`{contributor:Str`^{90}`,`
`organization:Str`^{80}`,`
`original:Str`^{90}`,`
`person:[{..}`^{20}`]`^{10}
`}`^{90}
`}`^{100}

Choosing the level of precision (VLDBJ'2019)

- Conciseness-precision trade off
  - concise schemas may lose cardinality information
  - precise schema may be too large
- Control the level of precision with an equivalence relation
- Interactive inference (ongoing work)

```
{byline:
 Null+
 {contributor:Str,
  organization:Str,
  original:Str,
  person:[ ]
 } +
 {contributor:Str,
  original:Str,
  person:[{..}]
 }
}
```

System-related schema inference approaches

- Selected systems: SparkSQL [1], MongoDB [12], Couchbase [10]
- Investigate the expressivity of the inferred schema
    - field optionality
    - union types
    - cardinality information
- No formal specification, testing and source code examination (partly)

Schema inference in SparkSQL [14]

- JSON data is mapped into relational tables with complex types (lists and objects)
- Built-in schema inference (Dataframe API, Catalyst query optimizer)
- Schema specified by the user or automatically inferred when loading data
- Infer structural properties only, all fields are optional (nullable), no union type

Illustration of SparkSQL schema inference

```
{first:"al",
 last:"jr",
 coord: null,
 email:".."
}
```

```
{first:Str?,
 last:Str?,
 coord:Str?,
 email:Str?
}
```

```
{first:"li",
 last:"ban",
 coord:{lat:45,
 long:12}
```

first	last	coord	email
"al"	"jr"	"null"	".."
"li"	"ban"	"{"lat":45,.."	
"jo"	"do"	"[45,12]"	

```
{first:"jo",
 last:"do",
 coord:[45,12]
}
```

`Re-parsing coord required!`

Schema inference in Mongodb [4]

- JSON data is stored natively (BSON)
- No schema inference, but possibility to validate data against a user-fed JSON-Schema
- Some external tools for schema inference (eg. mongodb-schema [31], [26])
    - Infer both structural and cardinality information, express union-type

Illustration of mongodb-schema inference [31]

```
{first:"al",
 last:"jr",
 coord: null,
 email:".."
}

{first:"li",
 last:"ban",
 coord:{lat:45,
 long:12}

{first:"jo",
 last:"do",
 coord:[45,12]
}
```

```
{count:3,
 fields: [
    {name:"first", count:3, proba:1,
     types:[{name:"string", count:1, proba:1,..}]
    },
   {name:"coord",
     types:[
           {name:"null", count:1, proba:0.33},
            {name:"document", count:1, proba:0.33,
              fields:[...] } ]
           {name:"array", count:1, proba:0.33,
            lengths:[2], average_length:2,
            types: [{name:"number", count:2, proba:1,..}]
           }
   },
   {name:"email", count:1, proba:0.33
    types:[{name:"string", count:1, proba:0.33..},
            {name:"undefined", count:2, proba:0.66..}]
    }
   {name:"last",...}
   ]
}
```

Schema inference in Couchbase [10]

- Native JSON storage, hence, data can have a flexible structure
- No schema validation but a built-in schema inference
- Infer both structural and cardinality information, no union-type, non-deterministic behavior when data have a varying structure

Illustration of the Couchbase schema inference

```
{first:"al",
 last:"jr",
 coord: null,
 email:".."
}
```

```
{first:"li",
 last:"ban",
 coord:{lat:45,
 long:12}
```

```
{first:"jo",
 last:"do",
 coord:[45,12]
}
```

```
[
 [
  {#docs:3,
   properties:
   {
    first: {#docs:3, %docs:100, type:"string"},
    coord: {#docs:1, %docs:33.33, type:"object",
           properties:
            {lat: {#docs:1, %docs:100, type:"number"},
             long: {#docs:1, %docs:100, type:"number"}
            }
    email: {#docs:1, %docs:33.33, type:"string"},
    last: {#docs:3, %docs:100, type:"string"}
   },
   type: "object"
 ]
]
```

Comparison of schema inference techniques

Features	Distributed inference	Spark SQL	Mongodb	Couchbase
optional fields	yes	no	yes	yes
structural variation	yes	no	yes	no
cardinality information	yes	no	yes	yes
precision tuning	yes	no	no	no

Comparison of schema inference techniques

Features	Distributed inference	Spark SQL	Mongodb	Couchbase
optional fields	yes	no	yes	yes
structural variation	yes	no	yes	no
cardinality information	yes	no	yes	yes
precision tuning	yes	no	no	no

NoSQL realm

- Manage JSON data in document-databases to account for variety
- Feed data into analytical systems like Spark using connectors

Schema Tools

Parsing Tools

Overview

- In the previous parts of this tutorial we outlined
  - The most important schema languages
  - How JSON data can be manipulated inside typed programming languages
  - How JSON schema information can be derived from a collection of JSON values
- In all these cases, we talked about explicit schema information
  - Designed by hand
  - Inferred
- There are however tools that exploit implicit schema information
  - Computed on the fly and destroyed after its use
  - Derived from applications or user queries

Mison Overview

- Mison [27] is a library for evaluating projection queries while parsing data
- Many times data analytics applications process data just once and access only a limited subset of object fields
- Since data must be parsed before data processing, Mison aims at anticipating query processing at parsing time

Mison key ideas

- Skip not required fields as much as possible
- Find a very quick way to locate fields in a JSON text

Mison Parsing Process

- Mison takes as input
    - A collection of JSON objects in textual form
    - A set of queried fields, possibly nested

```
{"id":"id:\"a\"", "reviews":50,
 "attributes":{"breakfast":false, "lunch":true,
               "dinner":true, "latenight":true},
 "categories":["Restaurant", "Bars"], "state":"WA", "city":"seattle"}
```

Queries
{“reviews”, “city”, “attributes.breakfast”, “attributes.lunch”, “attributes.dinner”, “attributes.latenight”, “categories”}

Mison Parsing Process

- Mison builds for each object a *structural index* that pinpoints field separators (“:”) in the object as well as element separators (“,”) in arrays
    - One bitmap per nesting level
    - One bit per character of the input string
- Mison uses this index to quickly locate fields
- Index construction time + index use time < parsing time with FSM parsers
    - Heavy use of SIMD vectorization + bitwise parallelism

Structural Index Example

Word: `{"id":"id:\"a\"","reviews":50,"a`

Structural ‘:’: `00000100000000000000000000100000`

L1 ‘:’ bitmap: `00000100000000000000000000100000`

L2 ‘:’ bitmap: `00000000000000000000000000000000`

The Structural Index is not Enough

- The structural index is great, but by relying on it the parser has still to analyze all fields

Another Mison key idea

- Speculative parsing
  - Making guesses about the position of required fields
- Another data structure
  - The Pattern Tree

Sample Dataset

```
{"id":"id:\"a\"", "reviews":50,
  "attributes":{"breakfast":false, "lunch":true,
               "dinner":true, "latenight":true},
 "categories":["Restaurant", "Bars"], "state":"WA", "city":"seattle"}

{"id":"id:\"b\"", "reviews":80,
 "attributes":{"breakfast":false, "lunch":true,
               "latenight":false, "dinner":true},
 "categories":["Restaurant"], "state":"CA", "city":"SF"}

{"id":"id:\"c\"", "reviews":120,
 "attributes":{"delivery":true, "lunch":true,
               "dessert": true, "dinner":true},
 "categories":["Restaurant"], "state":"NY", "city":"NY"}
```

Sample Dataset (Again)

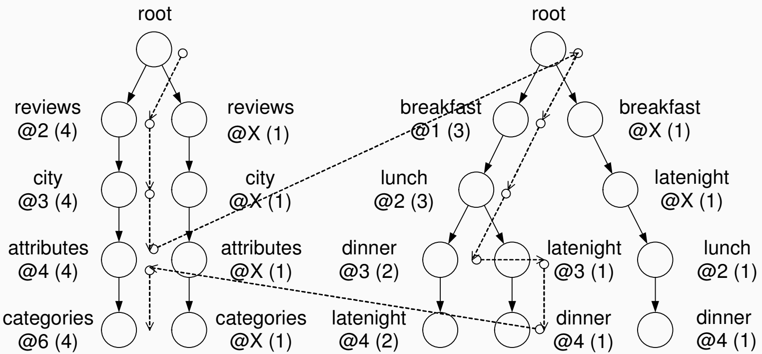
```
{"id":"id:\"d\"", "name":"Alice", "age":40, "favorites":30}

{"id":"id:\"e\"", "reviews":70,
 "attributes":{"breakfast":true, "lunch":true,
               "dinner":true, "latenight":false},
 "categories":["Restaurant", "Brunch"], "state":"CA", "city":"LA"}
```

Sample Queries and Pattern Trees

Queries

{“reviews”, “city”, “attributes.breakfast”, “attributes.lunch”, “attributes.dinner”, “attributes.latenight”, “categories”}

Pattern Trees



Pattern trees for the root field and for the “attributes” field

Parsing Example

New Object

```
{"id":"id:\"f\"", "reviews":20,
 "attributes":{"breakfast":true, "lunch":true,
               "latenight":true, "dinner":true},
 "categories":["Restaurant", "Brunch", "Bars"],
 "state":"IL", "city":"chicago"}
```

Queries

Searching for: “`attributes.breakfast`”, “`attributes.lunch`”, “`attributes.dinner`”, “`attributes.latenight`”

Guesses

The guesses for `attributes.breakfast` and `attributes.lunch` are trivial

Parsing Example

New Object

```
{"id":"id:\"f\"", "reviews":20,
 "attributes":{"breakfast":true, "lunch":true,
               "latenight":true, "dinner":true},
 "categories":["Restaurant", "Brunch", "Bars"],
 "state":"IL", "city":"chicago"}
```

Queries
Searching for: “`attributes.breakfast`”, “`attributes.lunch`”, “`attributes.dinner`”, “`attributes.latenight`”

Guesses
The first guesses for `attributes.dinner` and `attributes.latenight` are wrong: @3 and @4

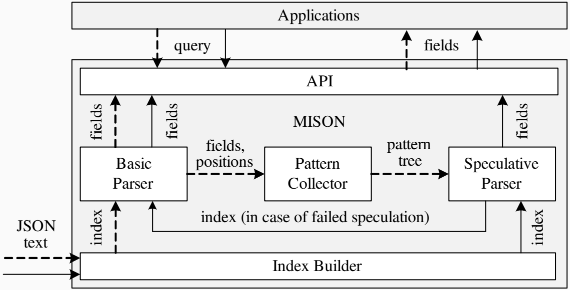
Mison has to inspect the second pattern to find a correct guess: @4 and @3

Parsing Process Architecture

- A two-step process

Training

- Mison starts parsing JSON objects through the *Basic Parser*
- The *Index Builder* creates a structural index per object and the *Basic Parser* answers user queries
- Objects are used for creating the pattern tree

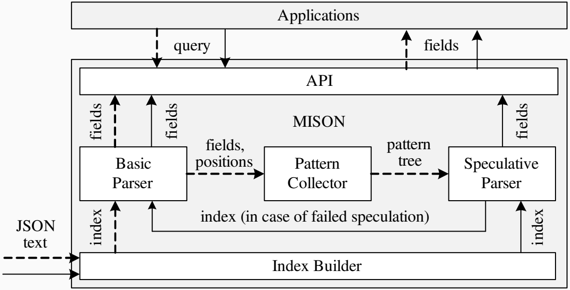


Parsing process architecture

Speculative Parsing

Speculative parsing

- After a given number of objects, the Index Builder is still used for creating the structural index
- The *Speculative Parser* answers user queries by making guesses about the position of queried fields
- Only if all the guesses are wrong, Mison resorts to the Basic Parser



Parsing process architecture

Structural Index

- One index per object
- Each index has one bitmap per nesting level and records the position of “:”
  - Quickest way to spot the location of a field
  - Built by using SIMD and bitwise parallelism
- Since bitmaps are leveled, no need to parse a nested object if one is interested in top level fields only
  - Just a way to parse JSON objects in a BFS style

Pattern Tree

- Each object is analyzed for creating a structural index.
- The first *n* objects are used to train the *speculative* parser
- During the training phase, common object patterns are summarized in the pattern tree
- An “horizontal” DataGuide
  - Nodes correspond to queried fields
  - Each node is endowed with its frequency as well as positional information
- One pattern tree for the root level
- One pattern tree for each object field
  - A field traversed by a path query and containing a nested object

Pattern Tree Remarks

- Field order is relevant
  - Mison guesses about the logical position of a field: the 3rd subfield of the 2nd root level field
- Fields that are not requested by the queries are just skipped
- To avoid size blow up, unfrequent patterns are pruned

Speculative Parsing

- Mison looks for queried fields inside each top level object
- For each queried field, Mison makes a guess by inspecting the corresponding pattern tree
    - Patterns are inspected from left to right (from the most frequent to the least frequent one)
- The guess is just the logical position of the field
    - Translated into a physical position by using the structural index
- If the guess is wrong, Mison resumes inspecting the pattern tree
- When no correct guesses can be done, Mison resorts to basic parsing
    - Structural index only

Mison Pros and Cons

- On-the-fly parsing
    - One time data analytics applications
    - Not the best choice for datasets that undergo multiple and/or iterative analysis
- Availability of SIMD instructions and access to SIMD registers
    - Virtualized environments
- It only supports collections of records

Future Opportunities

Overview

- In this tutorial we described so far
    - Several schema languages for JSON data
    - Tools for inferring schemas
    - Tools able to exploit implicit or explicit schema information
- In this very last section of the tutorial we discuss novel research opportunities that arise at the cross of different areas
    - Schema inference and ML
    - Schema-aware data cooking

Schema Inference and ML

- When inferring a schema for a JSON data collection, there is always a trade-off between precision and conciseness
    - Implicit if it is hard-wired in the inference algorithm
    - Explicit if the algorithm can infer different kinds of schemas
- It is very hard to find such a good trade-off
    - Human-in-the-loop approach
    - Entropy-based approach [23]
- None of these solutions is really satisfactory

Schema Inference and ML

- A human-in-the-loop approach
  - An initial schema is inferred
    - The most detailed or
    - The most concise
  - The user decides what parts should be collapsed/expanded
  - Tedious and time-consuming process
- Entropy-based approach
  - Unable to capture application access patterns
  - It relies on users’ interviews
    - “Everybody lies” (Gregory House, M.D.)
- Can ML help us?

Schema Inference and ML

- Learn how data are used inside user applications
  - Create more detailed schemas for frequently accessed data
  - Create more compact schemas for data rarely accessed
- Learn what are the value conditions that are mostly used
  - `Age >= 40`
  - Introduce value dependent types (see Joi, for instance) that capture these conditions
- A step further
  - Avoid schema fusion (or schema inference at all) for data never accessed by the user workload
    - Data collections can be greatly heterogeneous

Schema-aware Data Cooking

- JSON is awesome for exchanging data between applications
  - Self-describing
  - Flexible without the hassles of XML
  - Relatively close to nested relational
- JSON is terrible for data processing
  - Textual representation
  - No fast access to fields
- JSON data are usually “cooked”
  - Transformed in more efficient formats like
    - Parquet [9], Avro [6], Arrow [5], and many others

Schema-aware Data Cooking

- Data cooking has an upfront cost that may be significant
    - Affordable when data are to be processed multiple times
- Data formats like Parquet also store basic schema information
    - Column names and basic type information
- Several opportunities

Schema-aware Data Cooking

- Data cooking often requires one
  - To load JSON data in a system like Spark (and many others) and export them back in Parquet
    - Spark does the job of creating a schema
  - Or to design your own schema for the data
    - Apache NiFi
  - Or to exploit another format/schema language as a man in the middle

Streamline the process

- Scan the JSON file for inferring a schema **and** creating the Parquet representation
  - Need to find to right trade-off between different schema abstraction levels

References i

[1] Apache Spark.
`http://spark.apache.org`.

[2] ECMAScript Language Specification, dec 1999.
Third Edition.

[3] The JSON Data Interchange Syntax, dec 2017.

[4] Mongodb schema validation, 2019.
Available at
`https://docs.mongodb.com/manual/core/schema-validation/`.

[5] Apache Arrow.
`https://arrow.apache.org`.

References ii

[6] Apache Avro.
`https://avro.apache.org`.

[7] Apache Drill.
`https://drill.apache.org`.

[8] Apache Flink.
`https://flink.apache.org`.

[9] Apache Parquet.
`https://parquet.apache.org`.

[10] Couchbase auto-schema discovery.
https://blog.couchbase.com/auto-schema-discovery/.

References iii

[11] JSON Schema language.
`http://json-schema.org`.

[12] Mongo DB.
`https://www.mongodb.com`.

[13] PostgreSQL.
`https://www.postgresql.org`.

[14] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al.
Spark sql: Relational data processing in spark.
In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

References iv

[15] M. A. Baazizi, H. Ben Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani.
Schema inference for massive JSON datasets.
In *EDBT '17*, 2017.

[16] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani.
Counting types for massive JSON datasets.
In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, pages 9:1–9:12, 2017.

[17] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani.
Parametric schema inference for massive json datasets.
The VLDB Journal, pages 1–25, 2019.

References v

[18] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls.
Inference of concise dtds from XML data.
In *VLDB ‘06*, pages 115–126, 2006.

[19] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoc.
JSON: data model, query languages and schema specification.
In *PODS ’17*, pages 123–135, 2017.

[20] T. Bray.
The JavaScript Object Notation (JSON) Data Interchange Format.
Technical report, Internet Engineering Task Force (IETF), Dec 20017.
Standards Track.

References vi

[21] D. Colazzo, G. Ghelli, and C. Sartiani.
Typing massive json datasets.
In *XLDI '12, Affiliated with ICFP*, 2012.

[22] M. DiScala and D. J. Abadi.
Automatic generation of normalized relational schemas from nested key-value data.
In *SIGMOD '16*, pages 295–310, 2016.

[23] E. Gallinucci, M. Golfarelli, and S. Rizzi.
Schema profiling of document-oriented databases.
Inf. Syst., 75:13–25, 2018.

References vii

[24] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim.
XTRACT: A system for extracting document type descriptors from XML documents.
In *SIGMOD '00*, pages 165–176, 2000.

[25] R. Goldman and J. Widom.
Dataguides: Enabling query formulation and optimization in semistructured databases.
In *VLDB'97*, pages 436–445, 1997.

[26] T. S. Labs.
Studio 3T, 2017.
Available at `https://studio3t.com`.

References viii

[27] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann.
Mison: A fast JSON parser for data analytics.
PVLDB, 10(10):1118–1129, 2017.

[28] S. Nestorov, S. Abiteboul, and R. Motwani.
Extracting schema from semistructured data.
In *SIGMOD '98*, pages 295–306, 1998.

[29] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč.
Foundations of json schema.
In *WWW '16*, pages 263–273, 2016.

References ix

[30] 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 *Proceedings of the Workshops of the EDBT/ICDT 2016*, 2016.

[31] P. Schmidt.
mongodb-schema, 2017.
Available at `https://github.com/mongodb-js/mongodb-schema`.

[32] 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.