Schemas And Types For JSON Data: From Theory to Practice

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

2019 ACM SIGMOD/PODS, June 30-July 5, 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 ($\sim$ 25 min)
- Schema Tools ($\sim$ 50 min)
    - Our contribution
    - Schema Inference Tools
    - Data Ingestion
- Closing Remarks ($\sim$ 1 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

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

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 any JSON value
    - A record can be empty: {}

```
{
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "organization" : "",
    "rank" : 1,
    "role" :  "reported"
}
```

- Member labels are not required to be unique [4], very bad practice, can lead to unpredictable behaviour of applications [12]

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]

  -
    ```
    "byline": {
                        "original": "By MELENA RYZIK",
                        "person": [
                            {
                                "firstname": "Melena",
                                "lastname": "RYZIK",
                                "organization": "",
                                "rank": 1,
                                "role": "reported"
                            }
                       ]
                    }
    ```

Uses

JSON is prominently used for data interchange and storage

- 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

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
  - Allows for efficient checking of non-emptiness, schema inclusion, document validity, query correctness.

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
  - Allows for efficient checking of non-emptiness, schema inclusion, document validity, query correctness.
- Proposals we focus on: JSON Schema and Joi.
- By relying on several examples.

JSON Schema

Records are described by JSON object values of the form

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

Open record assumption - i.e., 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

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 `byline` information of NYT JSON data.

- The `byline` field can either
  - Have value null, or
  - Have an object as value, where “person” subfield is an empty array if the “organization” 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

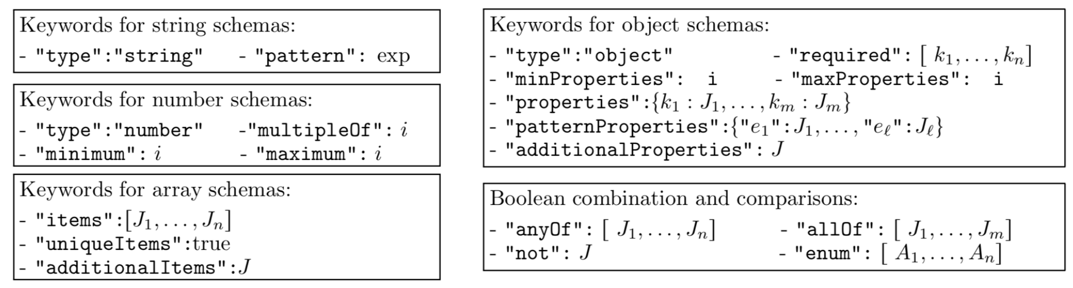
```
{
 "type" : "object",
 "properties"  :  { "contributor" : { "type" : "string" },
                    "organization" : { "type" : "string" },
                    "original" : { "type" : "string" },
                    "person" : { "type" : "array",
                                 "maxItems" : 0 } ,
 "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 [9].
- Formal semantics and study done in [16, 11], from which we borrow subsequent examples.
- Main properties in a nutshell [11]:



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

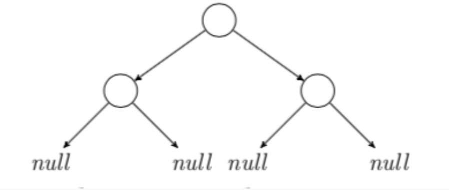
```
{ "type" : "object",
  "properties"  :  { "name" : { "type":"string"} },
  "patternProperties" : {
              "a(b|c)a" : { "type" : "number", "multipleOf" : 2} },
  "additionalProperties" : {
             "type": "number",
             "minimum" : 1,
             "maximum" : 1 }
          }
}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

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

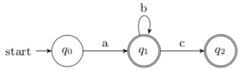
- Validation is the problem of checking whether a given JSON document J conforms to a given JSON schema S, noted as:

$$J \models S$$

- 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 [16].

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 [16], 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');
```

credit:https://github.com/hapijs/joi

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 JavaScript (but translators exist)
more verbose, expressed in JSON	more fluent to write/read
full support for union, disjunction, negation	limited support (work 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 and Mongoose, but with much less impact
- Work still needed in the standardisation, documentation, and specification of formal semantics

Schema Tools

Schema Tools

Our contribution

Schema Inference for Semistructured JSON Data

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system
- Structured and semistructured data

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system
- Structured and semistructured data
- Fully formalized

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system
- Structured and semistructured data
- Fully formalized
- Simple

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system
- Structured and semistructured data
- Fully formalized
- Simple
- Simple parallelizable algorithms

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system
- Structured and semistructured data
- Fully formalized
- Simple
- Simple parallelizable algorithms
- Parametric

Schema Inference for Semistructured JSON Data

- Schemas for the analyst and for the system
- Structured and semistructured data
- Fully formalized
- Simple
- Simple parallelizable algorithms
- Parametric
- Extensible

The type system

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

The type system

$\mathcal{B} ::= \quad \mathit{Null} \mid \mathit{Bool} \mid \mathit{Num} \mid \mathit{Str}$ *Basic types*

$R ::= \quad \{l_1 : J_1, \ldots, l_n : J_n\} \qquad n \geq 0$ **Records**

$A ::= \quad [J_1, \ldots, J_n] \quad n \geq 0$ **Arrays**

$J ::= \quad B \mid R \mid A$ **JSON expressions**

The type system

$$\begin{array}{llll}
\mathcal{B} ::= & Null \mid Bool \mid Num \mid Str & & \textit{Basic types} \\
\mathcal{R} ::= & \{l_1 : \mathcal{T}_1 q_1, \ldots, l_n : \mathcal{T}_n q_n\} & q_i \in \{'!', '?'\} \quad n \geq 0 & \textit{Record types} \\
A ::= & [J_1, \ldots, J_n] \quad n \geq 0 & & \text{Arrays} \\
J ::= & B \mid R \mid A & & \text{JSON expressions}
\end{array}$$

The type system

$\mathcal{B} ::=$	$Null \mid Bool \mid Num \mid Str$		*Basic types*
$\mathcal{R} ::=$	$\{l_1 : \mathcal{T}_1 q_1, \dots, l_n : \mathcal{T}_n q_n\}$	$q_i \in \{'!', '?'\} \quad n \geq 0$	*Record types*
$\mathcal{A} ::=$	$[\mathcal{T}]$		*Array types*
$J ::=$	$B \mid R \mid A$		**JSON expressions**

The type system

$$
\begin{array}{lllll}
\mathcal{B} ::= & \mathit{Null} \mid \mathit{Bool} \mid \mathit{Num} \mid \mathit{Str} & & & \textit{Basic types} \\
\mathcal{R} ::= & \{l_1 : \mathcal{T}_1 q_1, \ldots, l_n : \mathcal{T}_n q_n\} & q_i \in \{'!', '?'\} & n \geq 0 & \textit{Record types} \\
\mathcal{A} ::= & [\mathcal{T}] & & & \textit{Array types} \\
\mathcal{T} ::= & \mathcal{B} \mid \mathcal{R} \mid \mathcal{A} \mid +(\mathcal{T}_1, \ldots, \mathcal{T}_n) & & n \geq 0 & \textit{JSON types}
\end{array}
$$

Type flexibility

Type flexibility

- Assume a collection:

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...
- We can represent it as:

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...
- We can represent it as:
  - { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...
- We can represent it as:
  - { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- Or more precisely as:

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...
- We can represent it as:
  - { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- Or more precisely as:
  - { a: Int, b: (Int+Str), c: (Int+Str), d: Int}* + { a: Int, b: Int, e: Int, f: Int}*

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...
- We can represent it as:
  - { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- Or more precisely as:
  - { a: Int, b: (Int+Str), c: (Int+Str), d: Int}* + { a: Int, b: Int, e: Int, f: Int}*
- Or even:

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, ...
- We can represent it as:
  - { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- Or more precisely as:
  - { a: Int, b: (Int+Str), c: (Int+Str), d: Int}* + { a: Int, b: Int, e: Int, f: Int}*
- Or even:
  - { a: Int, b: Int, c: Int, d: Int}* + { a: Int, b: Str, c: Str, d: Int}* + { a: Int, b: Int, e: Int, f: Int}*

Type flexibility

- Assume a collection:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} , { a: , b: , e: , f: }, …
- We can represent it as:
  - { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- Or more precisely as:
  - { a: Int, b: (Int+Str), c: (Int+Str), d: Int}* + { a: Int, b: Int, e: Int, f: Int}*
- Or even:
  - { a: Int, b: Int, c: Int, d: Int}* + { a: Int, b: Str, c: Str, d: Int}* + { a: Int, b: Int, e: Int, f: Int}*
- No choice is “better”, it is even possible that I want to see more/less information in different moments

The equivalence parameter approach

The equivalence parameter approach

- The parameter: we let the analyst to decide size vs precision by fixing a parameter

The equivalence parameter approach

- The parameter: we let the analyst to decide size vs precision by fixing a parameter
- The *equivalence* parameter: the analyst choses a notion of similarity – two types are merged into one if they are “similar enough”

Useful equivalences

Useful equivalences

- K-equivalence: all records are similar:

Useful equivalences

- K-equivalence: all records are similar:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: 'a', c: 'b', d: 3} ,
    { a: , b: , e: , f: }, ...:
    { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*

Useful equivalences

- K-equivalence: all records are similar:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} ,
    { a: , b: , e: , f: }, ...:
    { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- L-equivalence: two records are equivalent if they have the same labels:

Useful equivalences

- K-equivalence: all records are similar:
    - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} ,
      { a: , b: , e: , f: }, ...:
      { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- L-equivalence: two records are equivalent if they have the same labels:
    - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} ,
      { a: , b: , e: , f: }, ...:
      { a: Int, b: (Int+Str), c: (Int+Str), d: Int}* + { a: Int, b: Int, e: Int, f: Int}*

Useful equivalences

- K-equivalence: all records are similar:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} ,
    { a: , b: , e: , f: }, ...:
    { a: Int, b: (Int+Str), c: (Int+Str)?, d: Int?, e: Int?, f: Int?}*
- L-equivalence: two records are equivalent if they have the same labels:
  - { a: 0, b: 1, c: 2, d: 0} , { a: 3, b: 1, e: 3, f: 2} , { a: 3, b: ‘a’, c: ‘b’, d: 3} ,
    { a: , b: , e: , f: }, ...:
    { a: Int, b: (Int+Str), c: (Int+Str), d: Int}* + { a: Int, b: Int, e: Int, f: Int}*
- Others

Our system

Our system

- We formalized that through a set of type rules

Our system

- We formalized that through a set of type rules
- We experimented a Spark map-reduce implementation

The equivalence approach

The equivalence approach

- Simple

The equivalence approach

- Simple
- Highly parallelizable

The equivalence approach

- Simple
- Highly parallelizable
- Parametric

The equivalence approach

- Simple
- Highly parallelizable
- Parametric
- But: too inflexible, in practice you would not employ the same equivalence everywhere

Interactive workbench

$$
\begin{array}{l}
+_K(\{\ \mathrm{docs} : \\
\qquad +_K(\{\ \mathrm{byline} : \\
\qquad\qquad +_K(\{\ \mathrm{organization} : +_K(\mathrm{Str})? \\
\qquad\qquad\qquad \mathrm{original} : +_K(\mathrm{Str}) \\
\qquad\qquad\qquad \mathrm{person} :[\ +_K(\{\mathrm{fn} : +_K(\mathrm{Str})?,\ \mathrm{ln} : +_K(\mathrm{Str})?,\ \mathrm{mn} : +_K(\mathrm{Str})?,\ \mathrm{org} : +_K(\mathrm{Str})?\})\ \ ] \\
\qquad\qquad\quad \}) \\
\qquad\quad \}) \\
\quad \})
\end{array}
$$

The byline

$$
\begin{array}{l}
+_K(\{\ \text{byline} : \\
\qquad +_K(\{\ \text{organization} : +_K(\text{Str})? \\
\qquad\qquad \text{original} : +_K(\text{Str}) \\
\qquad\qquad \text{person} :[+_K(\{\ \ \text{fn} : +_K(\text{Str})?, \text{ln} : +_K(\text{Str})?, \text{mn} : +_K(\text{Str})?, \text{org} : +_K(\text{Str})?\})\ \ ] \\
\qquad\qquad \}) \\
\qquad \})
\end{array}
$$

Expanding the byline

$$
\begin{array}{l}
+_K(\{\ \mathrm{byline} :\\
\qquad +_L(\{\ \mathrm{organization} : +_K(\mathrm{Str})\\
\qquad\qquad\quad \mathrm{original} : +_K(\mathrm{Str})\\
\qquad\qquad\quad \mathrm{person} :[+_K()\ ]\ \},\\
\qquad\qquad\ \ \{\mathrm{original} : +_K(\mathrm{Str})\\
\qquad\qquad\quad \mathrm{person} :[+_K(\{\ \ \mathrm{fn} : +_K(\mathrm{Str})?,\ \mathrm{ln} : +_K(\mathrm{Str})?,\ \mathrm{mn} : +_K(\mathrm{Str})?,\ \mathrm{org} : +_K(\mathrm{Str})?\})\ \ ]\ \}\\
\qquad\qquad\ \ )\\
\qquad \})
\end{array}
$$

Collapsing the byline

$$
\begin{array}{l}
+_K(\{\ \mathrm{byline} : \\
\qquad +_K(\{\ \mathrm{organization} : +_K(\mathrm{Str})? \\
\qquad\qquad \mathrm{original} : +_K(\mathrm{Str}) \\
\qquad\qquad \mathrm{person} : [+_K(\{\ \ \mathrm{fn} : +_K(\mathrm{Str})?, \mathrm{ln} : +_K(\mathrm{Str})?, \mathrm{mn} : +_K(\mathrm{Str})?, \mathrm{org} : +_K(\mathrm{Str})?\})\ ] \\
\qquad\qquad \}) \\
\qquad \})
\end{array}
$$

Expanding person

$$
\begin{array}{l}
+_K(\{\ \text{byline} : \\
\quad +_K(\{\ \text{organization} : +_K(\text{Str})? \\
\qquad \text{original} : +_K(\text{Str}) \\
\qquad \text{person} : [+_L(\{\ \text{fn} : +_K(\text{Str}),\ \text{ln} : +_K(\text{Str}),\ \text{mn} : +_K(\text{Str})\}, \\
\qquad\qquad\qquad\qquad \{\ \text{org} : +_K(\text{Str})\}\ )\ ] \\
\qquad \}) \\
\quad \})
\end{array}
$$

Expanding person - two

$+_K$({ byline :
$+_K$({ organization : $+_K$(Str)?
original : $+_K$(Str)
person : [$+_L$({ fn : ..., ln : ..., mn :..., org :...},
{ fn : ..., ln : ..., mn :...},
{ fn : ..., ln : ..., org :...},
{ fn : ..., ln : ...},
{ fn : ..., org :...},
{ ln : ..., org :...},
{ fn : ...},
{ fn : ..., mn :..., org :...},
{ fn : ..., mn :...},
{ ln : ...})]
})
})

Counting

Counting

- We infer this type

$$
\begin{array}{ll}
\{ & \text{title} : \text{Str} \,; \\
 & \text{text} : [\ \text{Str}\ ] + \text{Null} \,; \\
 & \text{author} : \{\ \text{address} : \text{T?} \,; \text{affiliation} : \text{T?} \,; \ldots\}? \,; \\
 & \text{abstract} : \text{Str?} \\
\} &
\end{array}
$$

Counting

- We infer this type

$$\begin{aligned}\{\quad & \text{title} : \text{Str} \,; \\ & \text{text} : [\ \text{Str}\ ] + \text{Null} \,; \\ & \text{author} : \{\ \text{address} : T? \,; \text{affiliation} : T? \,; \ldots\}? \,; \\ & \text{abstract} : \text{Str}? \\ \} & \end{aligned}$$

- How common is ‘optional’? How frequent is a branch? How big a collection?

Let us count

{ title : Str,
text : ([Str] + Null),
author : { address : T?, affiliation : T?, ...}?,
abstract : Str?
}

Let us count

$$\{\ \ \begin{array}{l} \text{title} : \text{Str}, \\ \text{text} : ([\ \text{Str}\ ] + \text{Null}), \\ \text{author} : \{\ \text{address} : \text{T?}, \text{affiliation} : \text{T?}, \ldots\}?, \\ \text{abstract} : \text{Str?} \end{array}$$
$$\}^{1000}$$

Let us count

$$\begin{array}{ll} \{ & \mathrm{title} : \mathrm{Str}^{1000}, \\ & \mathrm{text} : ([\ \mathrm{Str}\ ] + \mathrm{Null}), \\ & \mathrm{author} : \{\ \mathrm{address} : T?,\ \mathrm{affiliation} : T?,\ \ldots\}?, \\ & \mathrm{abstract} : \mathrm{Str}? \\ \}^{1000} & \end{array}$$

Let us count

$$
\begin{aligned}
\{\quad & \text{title} : \text{Str}^{1000}, \\
& \text{text} : ([\ \text{Str}\ ] + \text{Null})^{1000}, \\
& \text{author} : \{\ \text{address} : \text{T?},\ \text{affiliation} : \text{T?},\ \ldots\}?, \\
& \text{abstract} : \text{Str?} \\
\}^{1000} &
\end{aligned}
$$

Let us count

$$
\begin{aligned}
\{\quad & \text{title} : \text{Str}^{1000}, \\
& \text{text} : ([\ \text{Str}\ ]^{800} + \text{Null})^{1000}, \\
& \text{author} : \{\ \text{address} : \text{T?}, \text{affiliation} : \text{T?}, \ldots\}?, \\
& \text{abstract} : \text{Str?} \\
\}^{1000} &
\end{aligned}
$$

Let us count

$$
\begin{aligned}
\{\quad & \text{title} : \text{Str}^{1000}, \\
& \text{text} : ([\ \text{Str}\ ]^{800} + \text{Null}^{200})^{1000}, \\
& \text{author} : \{\ \text{address} : T?, \text{affiliation} : T?, \ldots\}?, \\
& \text{abstract} : \text{Str}? \\
\}^{1000} &
\end{aligned}
$$

Let us count

$$
\begin{array}{ll}
\{ & \text{title} : \text{Str}^{1000}, \\
 & \text{text} : ([\ \text{Str}^{8000}\ ]^{800} + \text{Null}^{200})^{1000}, \\
 & \text{author} : \{\ \text{address} : \text{T?},\ \text{affiliation} : \text{T?},\ \ldots\}\text{?}, \\
 & \text{abstract} : \text{Str?} \\
\}^{1000} &
\end{array}
$$

Let us count

$$\begin{aligned}
\{ \quad & \text{title} : \text{Str}^{1000}, \\
& \text{text} : ([\,\text{Str}^{8000}\,]^{800} + \text{Null}^{200})^{1000}, \\
& \text{author} : \{\, \text{address} : \text{T?}, \text{affiliation} : \text{T?}, \ldots\}^{800}, \\
& \text{abstract} : \text{Str}^{20} \\
\}^{1000} &
\end{aligned}$$

Let us count

$$
\begin{aligned}
\{\quad & \mathrm{title} : \mathrm{Str}^{1000}, \\
& \mathrm{text} : ([\ \mathrm{Str}^{8000}\ ]^{800} + \mathrm{Null}^{200})^{1000}, \\
& \mathrm{author} : \{\ \mathrm{address} : \mathrm{T}^{400}, \mathrm{affiliation} : \mathrm{T?}, \ldots\}^{800}, \\
& \mathrm{abstract} : \mathrm{Str}^{20} \\
\}^{1000} &
\end{aligned}
$$

Let us count

$$\begin{aligned}
\{\quad & \mathrm{title} : \mathrm{Str}^{1000}, \\
& \mathrm{text} : ([\ \mathrm{Str}^{8000}\ ]^{800} + \mathrm{Null}^{200})^{1000}, \\
& \mathrm{author} : \{\ \mathrm{address} : \mathrm{T}^{400}, \mathrm{affiliation} : \mathrm{T}^{200}, \ldots\}^{800}, \\
& \mathrm{abstract} : \mathrm{Str}^{20} \\
\}^{1000} &
\end{aligned}$$

Let us count

$$
\begin{aligned}
\{\quad & \text{title} : \text{Str}^{1000},\\
& \text{text} : ([\ \text{Str}^{8000}\ ]^{800} + \text{Null}^{200})^{1000},\\
& \text{author} : (\{\ \text{address} : \text{T}^{400}, \ldots\}^{400} + \{\ \text{affiliation} : \text{T}^{200}, \ldots\}^{400})^{800},\\
& \text{abstract} : \text{Str}^{20}\\
\}^{1000} &
\end{aligned}
$$

Conclusions

Conclusions

- A family of approaches

Conclusions

- A family of approaches
- Simple, fully formalized

Conclusions

- A family of approaches
- Simple, fully formalized
- Parametric

Conclusions

- A family of approaches
- Simple, fully formalized
- Parametric
- Parallelizable

Schema Tools

Schema Inference Tools

System-related schema inference approaches

- Selected systems: Spark SQL [1], MongoDB [10], Couchbase [8]
- Investigate the expressivity of the inferred schema
    - field optionality
    - union types
    - cardinality information
- No formal specification, testing and source code examination

Overview of Spark SQL

- A sub-system of Spark to process SQL over tables with complex values
- Built-in schema inference for data and for query results
- Schema used during data loading (CSV, JSON) and for logical query optimization

Spark SQL Data Model

- Basic values : Nulls, Booleans, Strings, many variants of numeric types (integer, long, ...), timestamps
- Objects: a list of (label, value) pairs
- Arrays: a list of values
- Maps: a collection of (key, value) pairs, a key can take any type

Spark SQL Schema language

$$
\begin{array}{lll}
T ::= & B \mid R \mid A \mid M & \textit{Types} \\
B ::= & \mathit{Null} \mid \mathit{Bool} \mid \mathit{Str} \mid \mathit{Num} \mid \mathit{Time} & \textit{Basic types} \\
R ::= & < (l_1, T_1, \mathit{Bool}), \ldots, (l_n, T_n, \mathit{Bool}) > & \textit{Record types} \\
A ::= & [\, T, \mathit{Bool} \,] & \textit{Array types} \\
M ::= & (T, T, \mathit{Bool}) & \textit{Map types}
\end{array}
$$

Bool indicates *nullability*

- No cardinality information
- No union type

Spark SQL Schema language: illustration

An (approximate) schema for byline

```
<("byline",
  < ("contributor", Str, true),
    ("original", Str, false),
    ("organization", Str, true),
    ("person",
     [ <("fn", Str, true),("ln", Str, true),("mn", Str, true)>, false ]
    )
  >, false
 )
```

Spark SQL Schema language: illustration

An (approximate) schema for byline

```
<("byline",
  < ("contributor", Str, true),
    ("original", Str, false),
    ("organization", Str, true),
    ("person",
     [ <("fn", Str, true),("ln", Str, true),("mn", Str, true)>, false ]
    )
  >, false
 )
```

when used during data loading

- Tolerate records with missing fields (nullable is *true* by default) ;
- Tolerate records not conforming to the schema (e.g. *person* is a string).

Spark SQL schema inference

- Distributed inference
- Infer the type of each object then combine the inferred types
- Combination rules
  - Fuse similar types
  - Coerce different types to:
    - the most common one when compatible, e.g. numeric types
    - *String* otherwise
- *Nullable* and ContainsNull set to true

Spark SQL schema inference: illustration

```
    { byline:                       { byline:
      {contributor: "...",            {contributor: "...",
       original: "...",                original: "...",
R1     organization: "...",     R2     person: [{fn: ".."},
       person: [ ]                        {mn: "..", ln: ".."} ]
     }                               }
    }                               }
```

Spark SQL schema inference: illustration

```
     { byline:
        {contributor: "...",
         original: "...",
R1       organization: "...",
         person: [ ]
       }
     }
```

```
     { byline:
        {contributor: "...",
         original: "...",
R2       person: [{fn: ".."},
            {mn: "..", ln: ".."} ]
       }
     }
```

```
<("byline",
  < ("contributor", Str, true),
    ("original", Str, true),
    ("organization", Str, true),
    ("person",
     [ <("fn", Str, true),("ln", Str, true),("mn", Str, true)>, true ]
    )
  >, true
 )
```

Schema for R1, R2

Spark SQL schema inference: illustration

```
     { byline:
       {contributor: "...",
        original: "...",
R1      organization: "...",
        person: [ ]
      }
     }
```

```
     { byline:
       {contributor: "...",
        original: "...",
R2      person: [{fn: ".."},
           {mn: "..", ln: ".."} ]
      }
     }
```

```
R3 { byline: [ ] }
```

Spark SQL schema inference: illustration

R1

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

R2

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

R3 `{ byline: [ ] }`

`<("byline", Str, true)>`

`Schema for R1, R2, R3`

`R1 and R2 encoded as a string, re-parsing is required`

System-related schema inference approaches

- Selected systems: Spark SQL [1], **MongoDB [10]**, Couchbase [8]
- Investigate the expressivity of the inferred schema
    - field optionality
    - union types
    - cardinality information
- No formal specification, testing and source code examination

Overview of Mongodb

- Native JSON support: binary storage (BSON), query and update capabilities
- No a priori schema required
- Built-in schema validation against a JSON-Schema specification
- Several external tools to infer schema from existing collection: Studio 3T [14], mongodb-schema [17]

The mongodb-schema inference

- A centralized, streaming-based JavaScript library
- Use a sample of the collection
- Infer both *structural* and *cardinality* schema information:
    - Field frequency, hence optionality
    - Union type
    - Array lengths
- Collect values

The mongodb-schema inference: illustration

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

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

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

The mongodb-schema inference: illustration

```
                    {count:3,

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


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


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



                    }
```

The mongodb-schema inference: illustration

```
{ 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", path:"first", count:3, proba:1,
     types:[{name:"string",..}]
    },




}
```

The mongodb-schema inference: illustration

```
{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", path:"first", count:3, proba:1,
     types:[{name:"string",..}]
    },
    {name:"coord", ...
    types:[
           {name:"null", count:1 ...},
            {name:"document", count:1...
               fields:[...] }
           {name:"array", count:1...
            types: [{name:"number"}]
           } ]
    },





}
```

The mongodb-schema inference: illustration

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


}
```

The mongodb-schema inference: illustration

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

System-related schema inference approaches

- Selected systems: Spark SQL [1], MongoDB [10], **Couchbase [8]**
- Investigate the expressivity of the inferred schema
    - field optionality
    - union types
    - cardinality information
- No formal specification, testing and source code examination

Overview of Couchbase

- Native JSON storage, 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	Spark SQL	Mongodb-schema	Couchbase
optional fields	no	yes	yes
structural variation	no	yes	no
cardinality information	no	yes	yes
precision tuning	no	no	no

Conclusion

- Data with high structural variety -> document databases
- Analytical pipelines combine document databases with general purpose processing systems (like Spark)

Schema Tools

Data Ingestion

Overview

- JSON is awesome for exchanging data between applications
- but it is terrible for data processing due to parsing overhead
- JSON data usually transformed into more efficient formats like Parquet [7], Avro [6], Arrow [5]
- Data transformation usually exploits an available schema for producing a compact representation, and to run efficiently

Parquet in a nutshell

- A binary, columnar, and compressed representation of nested records with possibly repeated fields
- Originally developed by Twitter and Cloudera, and based on Dremel [15]
- Records are shredded into columns, a column = a path to an atomic value
- Attach metadata to column to allow recovering original records

From Dremel to JSON

- A simple example loosely inspired by [3]
  - Mandatory fields
  - Optional fields
  - Repeated fields (0 or N occurrences), simulated with arrays in JSON

```
R1
{"owner":"Sherlock",
 "ownerNumbers": ["123", "456"],
 "contacts": [
          {"name": "John ", "number": "212"},
          {"name": "Greg"}
          ]}
```

```
R2
{"owner": "Mycroft"}
```

Parquet Shredding Mechanism

```
R1
{"owner":"Sherlock",
 "ownerNumbers": ["123", "456"],
 "contacts": [
          {"name": "John ", "number": "212"},
          {"name": "Greg"}
          ]}
```

```
R2
{"owner": "Mycroft"}
```

Parquet Shredding Mechanism

```
R1
{"owner":"Sherlock",
 "ownerNumbers": ["123", "456"],
 "contacts": [
          {"name": "John ", "number": "212"},
          {"name": "Greg"}
          ]}
```

```
R2
{"owner": "Mycroft"}
```

owner

R	D	Value
0	0	Sherlock
0	0	Mycroft

D=Number of *optional* or *repeated* fields **appearing** along the path

R=Number of *repeated* fields **repeating** along the path

Parquet Shredding Mechanism

```
R1
{"owner":"Sherlock",
 "ownerNumbers": ["123", "456"],
 "contacts": [
          {"name": "John ", "number": "212"},
          {"name": "Greg"}
          ]}
```

```
R2
{"owner": "Mycroft"}
```

owner

R	D	Value
0	0	Sherlock
0	0	Mycroft

ownerNumbers

R	D	Value
0	1	123
1	1	456
0	0	NULL

D=Number of *optional* or *repeated* fields **appearing** along the path

R=Number of *repeated* fields **repeating** along the path

Parquet Shredding Mechanism

```
R1
{"owner":"Sherlock",
 "ownerNumbers": ["123", "456"],
 "contacts": [
           {"name": "John ", "number": "212"},
           {"name": "Greg"}
           ]}
```

```
R2
{"owner": "Mycroft"}
```

owner

R	D	Value
0	0	Sherlock
0	0	Mycroft

ownerNumbers

R	D	Value
0	1	123
1	1	456
0	0	NULL

contacts.name

R	D	Value
0	1	John
1	1	Greg
0	0	NULL

D=Number of *optional* or *repeated* fields **appearing** along the path

R=Number of *repeated* fields **repeating** along the path

Parquet Shredding Mechanism

```
R1
{"owner":"Sherlock",
 "ownerNumbers": ["123", "456"],
 "contacts": [
          {"name": "John ", "number": "212"},
          {"name": "Greg"}
          ]}
```

```
R2
{"owner": "Mycroft"}
```

owner

R	D	Value
0	0	Sherlock
0	0	Mycroft

ownerNumbers

R	D	Value
0	1	123
1	1	456
0	0	NULL

contacts.name

R	D	Value
0	1	John
1	1	Greg
0	0	NULL

contacts.number

R	D	Value
0	2	212
1	1	NULL
0	0	NULL

D=Number of *optional* or *repeated* fields **appearing** along the path

R=Number of *repeated* fields **repeating** along the path

Schema Role

D=Number of *optional* or *repeated* fields **appearing** along the path

R=Number of *repeated* fields **repeating** along the path

owner

R	D	Value
0	0	Sherlock
0	0	Mycroft

ownerNumbers

R	D	Value
0	1	123
1	1	456
0	0	NULL

contacts.name

R	D	Value
0	1	John
1	1	Greg
0	0	NULL

contacts.number

R	D	Value
0	2	212
1	1	NULL
0	0	NULL

Schema Role

D=Number of *optional* or *repeated* fields **appearing** along the path

R=Number of *repeated* fields **repeating** along the path

owner

R	D	Value
0	0	Sherlock
0	0	Mycroft

ownerNumbers

R	D	Value
0	1	123
1	1	456
0	0	NULL

contacts.name

R	D	Value
0	1	John
1	1	Greg
0	0	NULL

contacts.number

R	D	Value
0	2	212
1	1	NULL
0	0	NULL

- Guide the data transformation: mandatory/optional/repeated fields
- Avoid storing *D* for mandatory fields

Closing Remarks

Wrap-up

- In this tutorial we described so far
    - Several schema languages for JSON
    - Tools for inferring schemas
    - Tools exploiting schema information for data loading
- Cross-domain techniques
    - Schema inference as a classification problem [13]

Many research opportunities

- User-centric schema inference

Many research opportunities

- User-centric schema inference
    - Learn how data are used inside user applications
        - Create more detailed schemas for frequently accessed data
        - Create more compact schemas for data rarely accessed

Many research opportunities

- User-centric schema inference
    - 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 in queries
        - dependent types can be exploited

Many research opportunities

- User-centric schema inference
  - 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 in queries
    - dependent types can be exploited
- Schema evolution management

References i

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

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

[3] Dremel made simple with parquet, 2013.
Available at
`https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-`

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

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

References ii

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

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

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

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

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

References iii

[11] 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.

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

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

References iv

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

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis.
Dremel: Interactive analysis of web-scale datasets.
PVLDB, 3(1):330–339, 2010.

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

References v

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