Schemas And Types For JSON Data

Mohamed-Amine Baazizi\textsuperscript{1} Dario Colazzo\textsuperscript{2} Giorgio Ghelli\textsuperscript{3} Carlo Sartiani\textsuperscript{4}

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

\textsuperscript{1}LIP6 - Sorbonne Université
\textsuperscript{2}LAMSADE - Université Paris-Dauphine, PSL Research University
\textsuperscript{3}Dipartimento di Informatica - Università di Pisa
\textsuperscript{4}DIMIE - Università della Basilicata
Outline

JSON Primer (~ 10 min)

Schema Languages (~ 20 min)

Types in Programming Languages (~ 15 min)

Schema Tools (~ 30 min)
  Schema Inference Tools
  Parsing Tools

Future Opportunities (~ 10 min)
JSON Primer
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

\[
J ::= B | R | A \\
B ::= \text{null} \mid \text{true} \mid \text{false} \mid n \mid s \quad n \in \text{Number}, \ s \in \text{String} \\
R ::= \{l_1 : J_1, \ldots, l_n : J_n\} \\
A ::= [J_1, \ldots, J_n] \quad n \geq 0
\]
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
• 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]
• 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
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.
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" ]
}
```
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

```json
{
  "definitions": {
    "S1": "..case with organisation field...
    "S2": "..case without organisation field"
  }
}
```

```json
{
  "type": "object",
  "properties": {
    "byline": {
      "anyOf": [
        null
      ]
    }
  }
}
```
A JSON Schema for NYT fragment - S1

```json
{
  "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

```json
{
  "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]:

<table>
<thead>
<tr>
<th>Keywords for string schemas:</th>
<th>Keywords for object schemas:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- &quot;type&quot;: &quot;string&quot;       - &quot;pattern&quot;: exp</td>
<td>- &quot;type&quot;: &quot;object&quot;   - &quot;required&quot;: [k₁, ..., kₙ]</td>
</tr>
<tr>
<td>- &quot;minProperties&quot;: i    - &quot;maxProperties&quot;: i</td>
<td>- &quot;minProperties&quot;: i    - &quot;maxProperties&quot;: i</td>
</tr>
<tr>
<td>- &quot;properties&quot;:{k₁: J₁, ..., kₘ: Jₘ}</td>
<td>- &quot;properties&quot;:{k₁: J₁, ..., kₘ: Jₘ}</td>
</tr>
<tr>
<td>- &quot;patternProperties&quot;:{e₁:J₁, ..., eₖ:Jₖ}</td>
<td>- &quot;patternProperties&quot;:{e₁:J₁, ..., eₖ:Jₖ}</td>
</tr>
<tr>
<td>- &quot;additionalProperties&quot;: J</td>
<td>- &quot;additionalProperties&quot;: J</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keywords for number schemas:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- &quot;type&quot;: &quot;number&quot;       - &quot;multipleOf&quot;: i</td>
</tr>
<tr>
<td>- &quot;minimum&quot;: i          - &quot;maximum&quot;: i</td>
</tr>
<tr>
<td>- &quot;items&quot;:[J₁, ..., Jₙ]</td>
</tr>
<tr>
<td>- &quot;uniqueItems&quot;:true</td>
</tr>
<tr>
<td>- &quot;additionalItems&quot;:J</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boolean combination and comparisons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- &quot;anyOf&quot;: [J₁, ..., Jₙ]</td>
</tr>
<tr>
<td>- &quot;allOf&quot;: [J₁, ..., Jₘ]</td>
</tr>
<tr>
<td>- &quot;not&quot;: J</td>
</tr>
<tr>
<td>- &quot;enum&quot;: [A₁, ..., Aₙ]</td>
</tr>
</tbody>
</table>
Object schemas

```json
{
   "type": "object",
   "properties": {
      "name": {
         "type": "string"
      }
   },
   "patternProperties": {
      "a(b|c)a": {
         "type": "number",
         "multipleOf": 2
      }
   },
   "additionalProperties": {
      "type": "number",
      "minimum": 1,
      "maximum": 1
   }
}
```
{  
  "type" : "array",  
  "items" : [ { "type" : "string" }, { "type" : "string" } ],  
  "additionalItems" : { "type" : "number" },  
  "uniqueItems" : true  
}
Boolean operators, recursion and path expressions

```json
{
  "definitions": {
    "S": {
      "anyOf": [
        {
          "enum": [null],
        },
        {
          "allOf": [
            {
              "type": "array",
              "minItems": 2,
              "maxItems": 2,
              "items": [
                {
                  "$ref": "#/definitions/S",
                },
                {
                  "$ref": "#/definitions/S"
                }
              ]
            },
            {
              "not": {
                "type": "array",
                "uniqueItems": true
              }
            }
          ]
        }
      ]
    }
  }
}
```
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 $O(|S| \times |J|)$, provided that `uniqueItems` is not used.
- Otherwise validation can be performed in $O(|S| \times \log(|J|) \times |J|)$ 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}}}}\}.

- 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.
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
const updatePassword = function (password) {
    Joi.assert(password, schema);
    console.log('Validation success!');
};

updatePassword('password');
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');
Important: closed record assumption

code:

```
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.
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**

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

  • boolean, number, string, null

  • 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]>
      ```
• Basic types:
  • boolean, number, string, null
• 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]`
· 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] [ ] )
• *Interface* object types - structural, transparent, open-ended:

- `{key1: type1, …, keyn: typen}`: 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`
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 }`
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: \texttt{string} \}
  • Interface declaration is just a shorthand (structural typing)
    • e.g.: \texttt{interface} NamedValue { name: \texttt{string} }
  • Optional fields:
    • e.g.: \texttt{interface} SquareConfig { color: \texttt{string}, width? : \texttt{number} }
    • If a width is present, its type is \texttt{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`
• *Interface* object types - structural, transparent, open-ended:
  • `{key₁ : type₁,..., keyₙ : typeₙ}`: 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 }`
· **Interface** object types - structural, transparent, open-ended:
  · `{key₁ : type₁,..., keyᵣ : typeᵣ}`: 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:
  - `{key₁ : type₁,..., keyₙ : typeₙ}`: 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:
  - `{key₁ : type₁,…, keyₙ : typeₙ}`: 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:
  - `{key₁ : type₁,..., keyₙ : typeₙ}`: 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 }

- Union types T | U

- 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>`
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*

• 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>`
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 }`
Advanced types in Typescript

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

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

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

    }
}
```
Arrays and interfaces model the essential JSON features
Arrays and interfaces model the essential JSON features
Union types and optional fields allow one to express semi-structured data
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
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
• 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
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
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
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)
• 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 $S_i$s into a single schema $S$ describing the entire collection
  
  *commutative* and *associative* operation
Input collection

Map

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

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

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

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

Map

Reduce

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

```json
{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)
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

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

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

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

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>coord</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;al&quot;</td>
<td>&quot;jr&quot;</td>
<td>&quot;null&quot;</td>
<td>&quot;..&quot;</td>
</tr>
<tr>
<td>&quot;li&quot;</td>
<td>&quot;ban&quot;</td>
<td>&quot;{lat&quot;:45,..&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;jo&quot;</td>
<td>&quot;do&quot;</td>
<td>&quot;[45,12]&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Re-parsing coord required!
• 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]

{count: 3, 
fields: [
    
    {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",...
    } 
  ]}
• 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

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

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

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

```json
[
    {#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

<table>
<thead>
<tr>
<th>Features</th>
<th>Distributed inference</th>
<th>Spark SQL</th>
<th>Mongodb</th>
<th>Couchbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional fields</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>structural variation</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>cardinality information</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>precision tuning</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Comparison of schema inference techniques

<table>
<thead>
<tr>
<th>Features</th>
<th>Distributed inference</th>
<th>Spark SQL</th>
<th>Mongodb</th>
<th>Couchbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional fields</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>structural variation</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>cardinality information</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>precision tuning</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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
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, "lattenight":true},
 "categories":["Restaurant", "Bars"], "state":"WA", "city":"seattle"}
```

Queries

{"reviews", "city", "attributes.breakfast", "attributes.lunch", "attributes.dinner", "attributes.latenight", "categories"}
• 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
Word:   {"id":"id:\"a\"","reviews":50,"a
Structural ‘’:  00000100000000000000000000000000100000
L1 ‘’ bitmap:  00000100000000000000000000000000100000
L2 ‘’ bitmap:  0000000000000000000000000000000000000000000000000
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

```json
{
    "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
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
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

The API of Mison is tailored to the design goals of Mison. Unlike the APIs of existing parsers that essentially iterate over all fields, Mison only returns the fields that are in the requested query (encoded as a list of fields that are required to be parsed, called query fields). The list of queried fields is called a pattern (Figure 3), the structural index is used by the parser to look up the queried fields. Each speculation is verified by comparing the field name at the logical position to the name of the parsed fields. In case of a match, the speculative parsing is successful and the parsed fields are returned to the application. Otherwise, the speculative parser predicts the logical positions of queried fields based on the pattern tree. If all speculations fail (unobserved pattern), Mison invokes the basic parser as a fall-through method, NextField(), returns the field ID of the next encountered field of interest in the current record until there is no more field of interest in the current record. The actual parsing is driven by two iteration methods. The first one, NextRecord(), skips the remaining of the current record and moves the cursor of the parser to the beginning of the next record. The second method, NextField(), returns the field ID of the next encountered field of interest in the current record until there is no more field of interest in the current record.
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.
• 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
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 \( \geq 40 \)
    - 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
• 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
• 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 the right trade-off between different schema abstraction levels


[12] Mongo DB.  

[13] PostgreSQL.  

Spark sql: Relational data processing in spark.  
Schema inference for massive JSON datasets. 
In EDBT ’17, 2017.

Counting types for massive JSON datasets. 

Parametric schema inference for massive json datasets. 


**Typing massive json datasets.**  
In *XLDI ’12, Affiliated with ICFP*, 2012.

**Automatic generation of normalized relational schemas from nested key-value data.**  

**Schema profiling of document-oriented databases.**  


