Schemas And Types For JSON Data: From Theory to Practice

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

2019 ACM SIGMOD/PODS, June 30-July 5, 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 (~ 25 min)

Schema Tools (~ 50 min)
  Our contribution
  Schema Inference Tools
  Data Ingestion

Closing Remarks (~ 1 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
\[\begin{align*}
J &::= B \mid R \mid A \\
B &::= \text{null} \mid \text{true} \mid \text{false} \mid n \mid s \quad n \in \text{Number}, \quad s \in \text{String} \\
R &::= \{l_1 : J_1, \ldots, l_n : J_n\} \\
A &::= [J_1, \ldots, J_n] \quad n \geq 0
\end{align*}\]
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 any JSON value
  • A record can be empty: {}

```json
{
  "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"
      }
    ]
  }
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
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
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
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.
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.
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" }
    }
}
```
The type of records *only* having “a” and “b” fields of type string

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

```json
{
  "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

```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" ]
}
```
```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 [9].
- Formal semantics and study done in [16, 11], from which we borrow subsequent examples.
- Main properties in a nutshell [11]:

<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></td>
<td>- &quot;minProperties&quot;: i - &quot;maxProperties&quot;: i</td>
</tr>
<tr>
<td></td>
<td>- &quot;properties&quot;:{k₁: J₁, ..., kₘ: Jₘ}</td>
</tr>
<tr>
<td></td>
<td>- &quot;patternProperties&quot;:{&quot;e₁&quot;:J₁, ..., &quot;eₖ&quot;:Jₖ}</td>
</tr>
<tr>
<td></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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keywords for array schemas:</th>
</tr>
</thead>
<tbody>
<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] - &quot;allOf&quot;: [J₁, ..., Jₘ]</td>
</tr>
<tr>
<td>- &quot;not&quot;: J - &quot;enum&quot;: [A₁, ..., Aₙ]</td>
</tr>
</tbody>
</table>
```json
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "patternProperties": {
    "a(b|c)a": {
      "type": "number",
      "multipleOf": 2
    }
  },
  "additionalProperties": {
    "type": "number",
    "minimum": 1,
    "maximum": 1
  }
}
```
Arrays

```json
{
    "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}  
          }  
        ]  
      ]  
    }  
  }  
}
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 [16].
Expressivity: JSON Schema is inherently as expressive as NFAs

• JSON string encoding, e.g., "abbc" → \{"a":{"b":{"b":{"c": Null}}}\}

• 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.
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 = require('joi');
const schema = Joi.string().min(6).max(10);
const updatePassword = function (password) {
  Joi.assert(password, schema);
  console.log('Validation success!');
};
updatePassword('password');
Important: *closed record assumption*

```javascript
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
Important: **closed record assumption**

```javascript
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 JavaScript (but translators exist)</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 (work 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 exist, 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

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

\[
B ::= \text{null} \mid \text{true} \mid \text{false} \mid n \mid s \\
R ::= \{l_1 : J_1, \ldots, l_n : J_n\} \\
A ::= [J_1, \ldots, J_n] \quad n \geq 0 \\
J ::= B \mid R \mid A
\]

$n \in \text{Number}, \ s \in \text{String}$

Basic values

Records

Arrays

JSON expressions
The type system

\[ \mathcal{B} ::= Null \mid \text{Bool} \mid \text{Num} \mid \text{Str} \]

\[ R ::= \{l_1 : J_1, \ldots, l_n : J_n\} \quad n \geq 0 \]

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

\[ J ::= \mathcal{B} \mid R \mid A \]

**Basic types**

**Records**

**Arrays**

**JSON expressions**
The type system

\[ B ::= \text{Null} \mid \text{Bool} \mid \text{Num} \mid \text{Str} \]

\[ R ::= \{l_1 : T_{q_1}, \ldots, l_n : T_{q_n}\} \quad q_i \in \{'!', '?'\} \quad n \geq 0 \]

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

\[ J ::= B \mid R \mid A \]

Basic types

Record types

Arrays

JSON expressions
The type system

\[ \mathcal{B} ::= \text{Null} \mid \text{Bool} \mid \text{Num} \mid \text{Str} \]

\[ \mathcal{R} ::= \{ l_1 : T_{q_1}, \ldots, l_n : T_{q_n} \} \quad q_i \in \{ ', ' \} \quad n \geq 0 \]

\[ \mathcal{A} ::= [T] \]

\[ J ::= B \mid R \mid A \]

Basic types
Record types
Array types
JSON expressions
The type system

\[ \mathcal{B} ::= \text{Null} \mid \text{Bool} \mid \text{Num} \mid \text{Str} \]

\[ \mathcal{R} ::= \{ l_1 : \mathcal{T}_1 q_1, \ldots, l_n : \mathcal{T}_n q_n \} \]

\[ \mathcal{A} ::= [\mathcal{T}] \]

\[ \mathcal{T} ::= \mathcal{B} \mid \mathcal{R} \mid \mathcal{A} \mid +(\mathcal{T}_1, \ldots, \mathcal{T}_n) \]

- \( q_i \in \{!'', '?' \} \)
- \( n \geq 0 \)

**Basic types**

**Record types**

**Array types**

**JSON types**

\( n \geq 0 \)
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? \}*
    • 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
Type flexibility

• Assume a collection:

• We can represent it as:

• Or more precisely as:

• Or even:

• No choice is "better", it is even possible that I want to see more/less information in different moments
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?}*
  - + { a: Int, b: Int, e: Int, f: 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
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, 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
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 \}*

- No choice is "better", it is even possible that I want to see more/less information in different moments.
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:

• No choice is “better”, it is even possible that I want to see more/less information in different moments
The equivalence parameter approach

- The parameter: we let the analyst decide size vs precision by fixing a parameter.
- The equivalence parameter: the analyst chooses a notion of similarity – two types are merged into one if they are "similar enough".
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 chose a notion of similarity – two types are merged into one if they are "similar enough"
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? }, …:

• 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 }, …:

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: Int, b: (Int+Str), c: (Int+Str)?, d: 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: Int, b: Int, e: Int, f: Int \}*

• Others
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

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

- Simple
- Highly parallelizable
- Parametric

But: too inflexible, in practice you would not employ the same equivalence everywhere
The equivalence approach

• Simple
The equivalence approach

- Simple
- Highly parallelizable

But: too inflexible, in practice you would not employ the same equivalence everywhere
The equivalence approach

- Simple
- Highly parallelizable
- Parametric

But: too inflexible, in practice you would not employ the same equivalence everywhere
The equivalence approach

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

\[ +_K(\{ \text{docs} : \\
\quad +_K(\{ \text{byline} : \\
\quad \quad +_K(\{ \text{organization} : +_K(\text{Str})? \\
\quad \quad \quad \text{original} : +_K(\text{Str}) \\
\quad \quad \quad \text{person} : [+_K(\{ \text{fn} : +_K(\text{Str})?, \text{ln} : +_K(\text{Str})?, \text{mn} : +_K(\text{Str})?, \text{org} : +_K(\text{Str})?}) ] \\
\quad \}) \\
\}) \\
\}) \]
The byline

\[
+_{K}\{\text{byline}:
    +_{K}\{\text{organization} : +_{K}(\text{Str})?
    \text{original} : +_{K}(\text{Str})
    \text{person} : [+_{K}\{\text{fn} : +_{K}(\text{Str})?, \text{ln} : +_{K}(\text{Str})?, \text{mn} : +_{K}(\text{Str})?, \text{org} : +_{K}(\text{Str})?]\}\}
\]
Expanding the byline

$$+_{K}\{ \text{byline} :$$
$$+_{L}\{ \text{organization} : +_{K}(\text{Str})$$
$$\text{original} : +_{K}(\text{Str})$$
$$\text{person} : [+_{K}() ] \},$$
$$\{ \text{original} : +_{K}(\text{Str})$$
$$\text{person} : [+_{K}\{ \text{fn} : +_{K}(\text{Str})?, \text{ln} : +_{K}(\text{Str})?, \text{mn} : +_{K}(\text{Str})?, \text{org} : +_{K}(\text{Str})?\} ] \} \}$$
Collapsing the byline

\[ +_K\{ \text{byline} : \\
+_K\{ \text{organization} : +_K(\text{Str})? \\
\text{original} : +_K(\text{Str}) \\
\text{person} : +_K\{ \text{fn} : +_K(\text{Str})?, \text{ln} : +_K(\text{Str})?, \text{mn} : +_K(\text{Str})?, \text{org} : +_K(\text{Str})? \} \} \} \]
Expanding person

\[ +_K \{ \text{byline} : \\
    +_K \{ \text{organization} : +_K \text{(Str)?} \\
    \text{original} : +_K \text{(Str)} \\
    \text{person} : [+_L \{ \text{fn} : +_K \text{(Str)}, \text{ln} : +_K \text{(Str)}, \text{mn} : +_K \text{(Str)}, \}
    \{ \text{org} : +_K \text{(Str)} \} \} ] \\
\} ] \\
\]
Expanding person - two

\[ +_K \{ \text{byline : } +_K \{ \text{organization : } +_K (\text{Str})? \}
\begin{align*}
&\text{original : } +_K (\text{Str}) \\
&\text{person : } [+_L \{ \{ \text{fn : ...}, \text{ln : ...}, \text{mn :...}, \text{org :...} \}, \\
&\{ \text{fn : ...}, \text{ln : ...}, \text{mn :...} \}, \\
&\{ \text{fn : ...}, \text{ln : ...}, \text{org :...} \}, \\
&\{ \text{fn : ...}, \text{ln : ...} \}, \\
&\{ \text{fn : ...}, \text{org :...} \}, \\
&\{ \text{ln : ...}, \text{org :...} \}, \\
&\{ \text{fn : ...} \}, \\
&\{ \text{fn : ...}, \text{mn :...}, \text{org :...} \}, \\
&\{ \text{fn : ...}, \text{mn :...} \}, \\
&\{ \text{ln : ...} \} \} \} \}
\end{align*} \]
Counting

• We infer this type

• How common is 'optional'? How frequent is a branch? How big a collection?
• We infer this type

```javascript
{   title : Str ;
    text : [ Str ] + Null ;
    author : { address : T? ; affiliation : T? ; ...}? ;
    abstract : Str?
}
```
Counting

• We infer this type
  
  ```
  {   title : Str ;
      text : [ Str ] + Null ;
      author : { address : T? ; affiliation : T? ; ...}? ;
      abstract : Str?
  }
  ```

• How common is ‘optional’? How frequent is a branch? How big a collection?
Let us count

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

```json
{  title : Str,
    text : ([ Str ] + Null),
    author : { address : T?, affiliation : T?, ...}?,
    abstract : Str?
}1000
```
Let us count

{ title : Str[^1000],
  text : ([ Str ] + Null),
  author : { address : T?, affiliation : T?, ...}?,
  abstract : Str?
}[^1000]
Let us count

```json
{  title: Str^1000,
   text: ([ Str ] + Null)^1000,
   author: { address: T?, affiliation: T?, ...}?,
   abstract: Str?
}^1000
```
Let us count

{  title : Str^{1000},
    text : ([ Str ]^{800} + Null)^{1000},
    author : { address : T?, affiliation : T?, ...}?,
    abstract : Str?
}^{1000}
Let us count

{  title : Str^{1000},
    text : ([ Str ]^{800} + Null^{200})^{1000},
    author : { address : T?, affiliation : T?, ...}?,
    abstract : Str?
}^{1000}
Let us count

{ title : Str^{1000},
  text : ([ Str^{8000} ]^{800} + Null^{200})^{1000},
  author : { address : T?, affiliation : T?, ...}?,
  abstract : Str?
}^{1000}
Let us count

```javascript
{
    title: Str^1000,
    text: ([ Str^800 ]^800 + Null^200)^1000,
    author: { address: T?, affiliation: T?, ...}^800,
    abstract: Str^20
}
```
Let us count

{ 
  title: Str\textsuperscript{1000},
  text: ([Str\textsuperscript{800}]\textsuperscript{800} + \text{Null}\textsuperscript{200})\textsuperscript{1000},
  author: { address: T\textsuperscript{400}, affiliation: T?, ...}\textsuperscript{800},
  abstract: Str\textsuperscript{20}
}\textsuperscript{1000}
Let us count

{  title : Str^{1000},
    text : ([Str^{8000}]^{800} + Null^{200})^{1000},
    author : { address : T^{400}, affiliation : T^{200}, ...}^{800},
    abstract : Str^{20}
}^{1000}
Let us count

{  title : Str\(^{1000}\),
   text : ([ Str\(^{800}\) ]\(^{800}\) + Null\(^{200}\))\(^{1000}\),
   author : ({ address : T\(^{400}\), ...})\(^{400}\) + { affiliation : T\(^{200}\), ...})\(^{400}\)\(^{800}\),
   abstract : Str\(^{20}\)
}\(^{1000}\)
• A family of approaches
• Simple, fully formalized
• Parametric
• Parallelizable
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

\[
T ::= B | R | A | M \\
B ::= \text{Null} | \text{Bool} | \text{Str} | \text{Num} | \text{Time} \\
R ::= \langle l_1, T_1, \text{Bool} \rangle, \ldots, \langle l_n, T_n, \text{Bool} \rangle \\
A ::= [ T, \text{Bool} ] \\
M ::= (T, T, \text{Bool})
\]

\text{Bool} \text{ indicates } \text{nullability}

- No cardinality information
- No union type
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)
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

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

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

R1 and R2 encoded as a string, re-parsing is required
Spark SQL schema inference: illustration

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

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

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

{ byline: [ ] }
Spark SQL schema inference: illustration

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

Schema for R1, R2, R3
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

```json
{
  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

```json
{
  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",
      ..
    }
  ]
}
```
The mongodb-schema inference: illustration

```
{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:"email", ...
    types:[{name:"string", count:1...,
      {name:"undefined", count:2...}
    ],
   {name:"last",...
']
```
The mongodb-schema inference: illustration

```json
{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

```json
{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: "email", ...}
  types: [{name: "string", count: 1...},
   {name: "undefined", count: 2...}]
  ]
}
```
```json
{
  "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...
        }
      ]
    },
    {
      "name": "email",
      "types": [{"name": "string", count: 1...}, {"name": "undefined", count: 2...}]
    }
  ]
}
```
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

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

Conclusion

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

Data Ingestion
• 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"}
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"}

<table>
<thead>
<tr>
<th>owner</th>
<th>R</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>Sherlock</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>Mycroft</td>
</tr>
</tbody>
</table>

D=Number of optional or repeated fields appearing along the path
R=Number of repeated fields repeating along the path
# Parquet Shredding Mechanism

**R1**

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

**R2**

```json
{"owner": "Mycroft"}
```

<table>
<thead>
<tr>
<th>owner</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Sherlock</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Mycroft</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ownerNumbers</th>
<th>R</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>456</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

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"}

<table>
<thead>
<tr>
<th>owner</th>
<th>ownerNumbers</th>
<th>contacts.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>D</td>
<td>Value</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Sherlock</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Mycroft</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
</tr>
</tbody>
</table>

D=Number of optional or repeated fields appearing along the path
R=Number of repeated fields repeating along the path
### Parquet Shredding Mechanism

**R1**

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

**R2**

```json
{"owner": "Mycroft"}
```

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>0</td>
<td>0</td>
<td>Sherlock</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>Mycroft</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ownerNumbers</td>
<td>0</td>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>456</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>contacts.name</td>
<td>0</td>
<td>1</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Greg</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>D</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>contacts.number</td>
<td>0</td>
<td>2</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>owner</th>
<th>ownerNumbers</th>
<th>contacts.name</th>
<th>contacts.number</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>D</td>
<td>Value</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Sherlock</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Mycroft</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Greg</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>0</td>
</tr>
</tbody>
</table>
D=Number of optional or repeated fields appearing along the path
R=Number of repeated fields repeating along the path

- Guide the data transformation: mandatory/optional/repeated fields
- Avoid storing D for mandatory fields
Closing Remarks
• 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

- Learn what are the value conditions that are mostly used in queries
- Dependent types can be exploited
- Schema evolution management
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


[10] Mongo DB.  
JSON: data model, query languages and schema specification.
In PODS ’17, pages 123–135, 2017.

The JavaScript Object Notation (JSON) Data Interchange Format.
Standards Track.

Schema profiling of document-oriented databases.
*Studio 3T, 2017.*  
Available at https://studio3t.com.

*Dremel: Interactive analysis of web-scale datasets.*  

*Foundations of json schema.*  