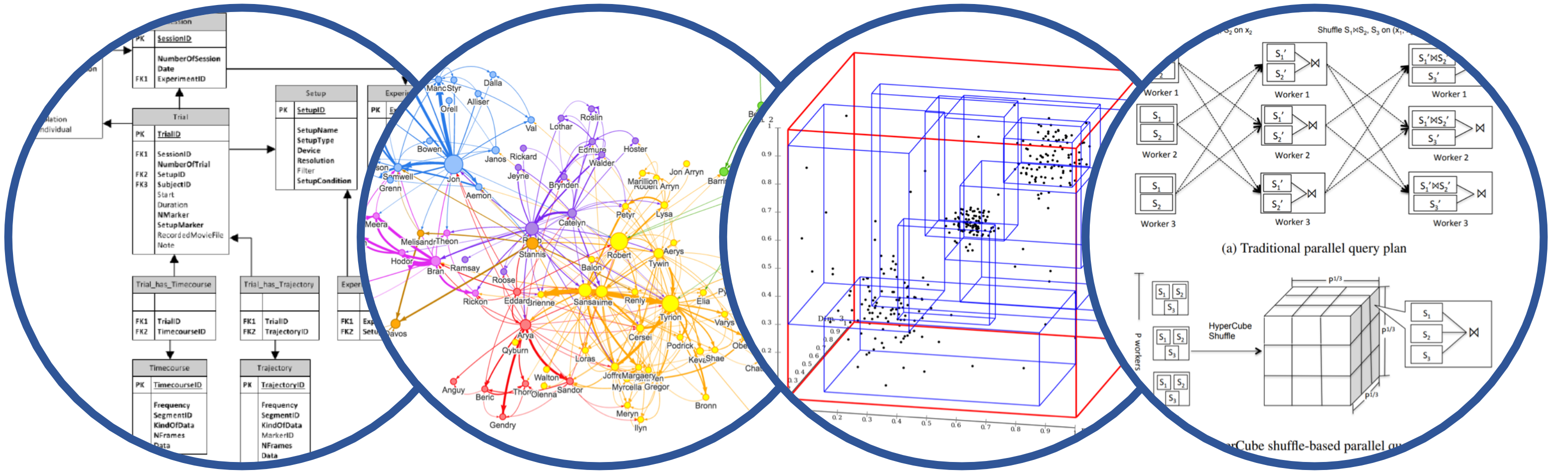


Introduction to Data Management

Semi-structured Data

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- HW 6:
  - due Sunday night
  - Up to 2 late days allowed

- HW 7: (out today, much less complex than normal HW)
  - due the Saturday after final exam
  - NO LATE DAYS ALLOWED

- Schedule for next week:
  - Today: Semi-structured data
  - Monday: SQL++ and HW 7
  - Wednesday: Misc. topics
  - Thursday section: exam review session
  - Friday: **Final exam**

Outline

- AsterixDB as a case study of Document Store
  - Semi-structured data model in JSON
  - Introducing AsterixDB and SQL++

Outline

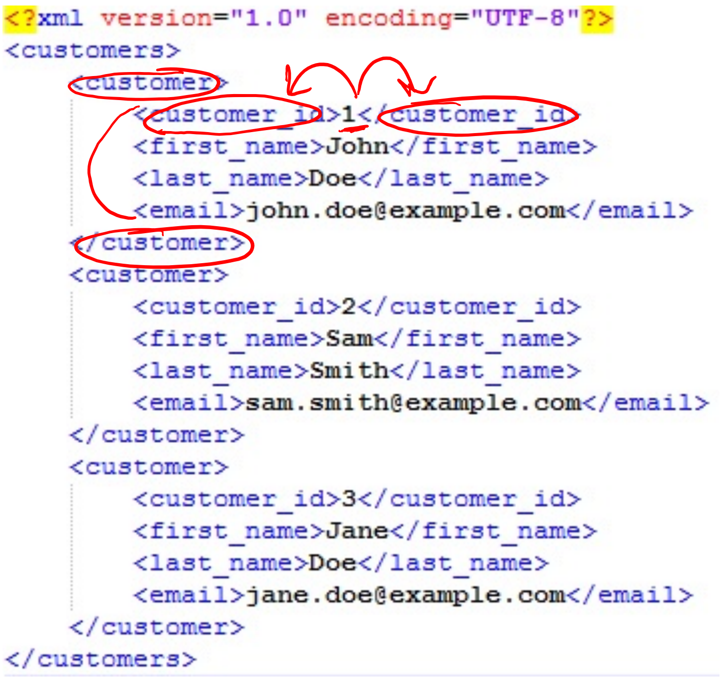
- AsterixDB as a case study of Document Store
    - **Semi-structured data model in JSON**
    - Introducing AsterixDB and SQL++

What is a "document" anyways?

- Loose terminology
- Any "parsable" file qualifies
  - Ex: MongoDB can handle CSV files

Semi-Structured Documents

- Some notion of **tagging** to mark down semantics
- Examples:
  - **XML**
  - Protobuf
  - JSON

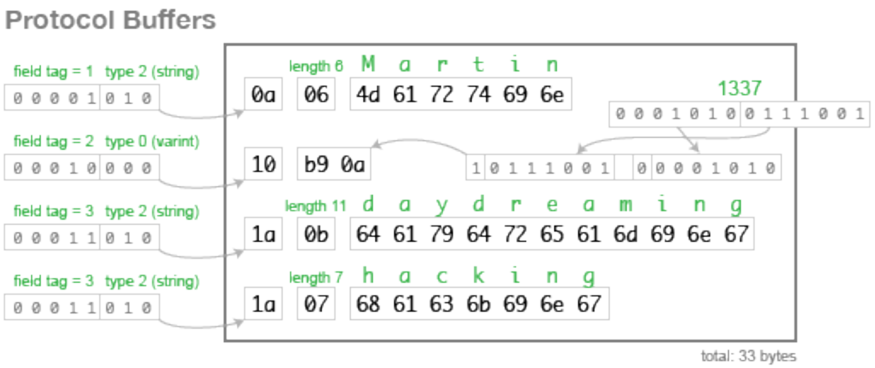


```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
    <customer>
        <customer_id>1</customer_id>
        <first_name>John</first_name>
        <last_name>Doe</last_name>
        <email>john.doe@example.com</email>
    </customer>
    <customer>
        <customer_id>2</customer_id>
        <first_name>Sam</first_name>
        <last_name>Smith</last_name>
        <email>sam.smith@example.com</email>
    </customer>
    <customer>
        <customer_id>3</customer_id>
        <first_name>Jane</first_name>
        <last_name>Doe</last_name>
        <email>jane.doe@example.com</email>
    </customer>
</customers>
```

Tags surround the respective data

Semi-Structured Documents

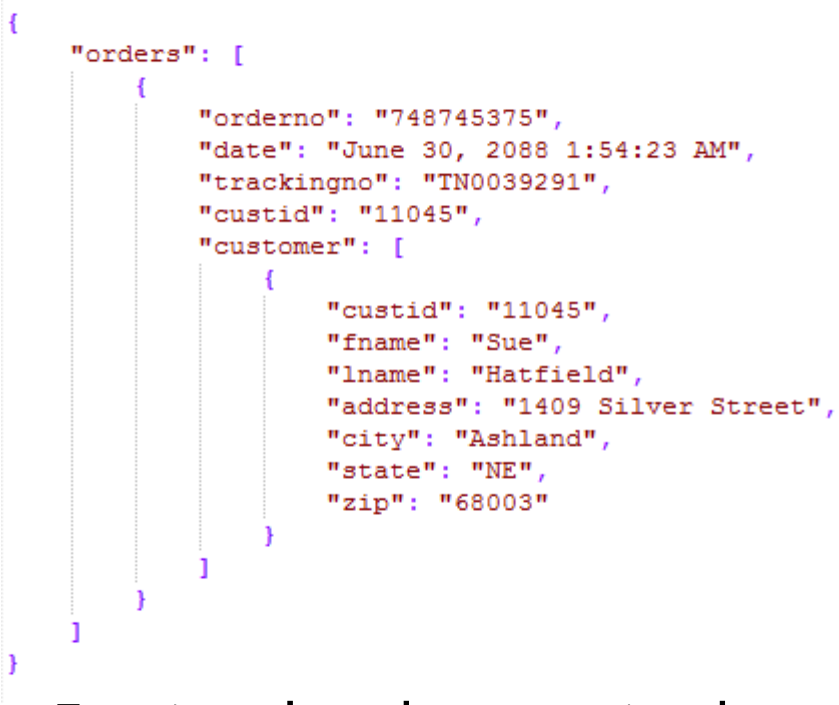
- Some notion of **tagging** to mark down semantics
- Examples:
  - XML
  - **Protobuf**
  - JSON



Able to record field number and type but not name

Semi-Structured Documents

- Some notion of **tagging** to mark down semantics
- Examples:
  - XML
  - Protobuf
  - **JSON**



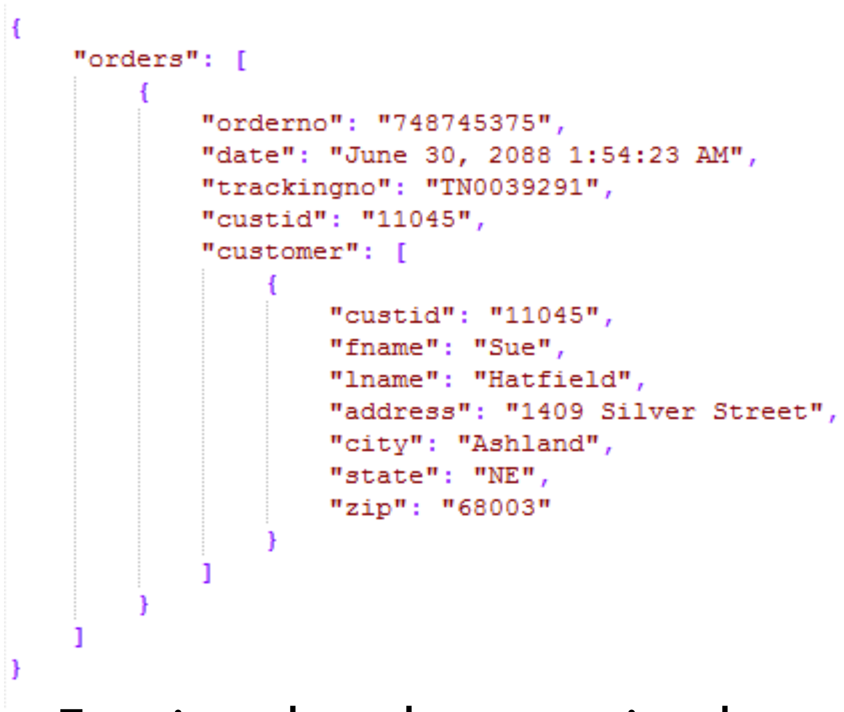
```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

Tags introduce the respective data

Semi-Structured Documents

- Some notion of **tagging** to mark down semantics
- Examples:
  - XML
  - Protobuf
  - **JSON**

Many applications have phased out XML in favor of JSON



```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

Tags introduce the respective data

Relational vs Semi-Structured Tradeoffs

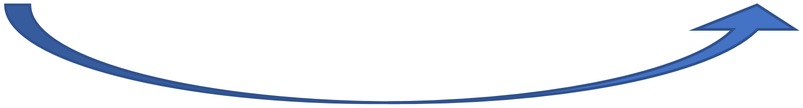
- Relational Model
  - Fixed schema
  - Flat data

- Semi-Structured
  - Self-described schema
  - Tree-structured data

Relational vs Semi-Structured Tradeoffs

- Relational Model
  - Fixed schema
  - Flat data

- Semi-Structured
  - Self-described schema
  - Tree-structured data

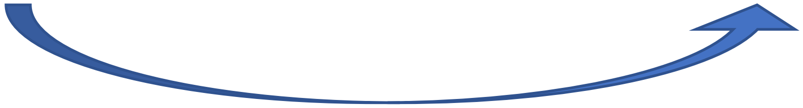


Less well-defined/More flexible

Relational vs Semi-Structured Tradeoffs

- Relational Model
  - Fixed schema
  - Flat data

- Semi-Structured
  - Self-described schema
  - Tree-structured data



Less well-defined/More flexible

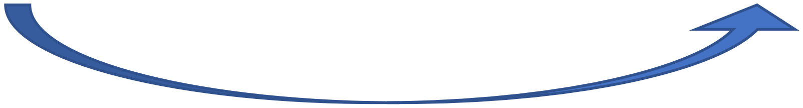
- Basic retrieval process:
  1. Retrieve table
  2. Run through rows
  3. Return data

- Basic retrieval process:
  1. Retrieve document
  2. Parse document tree
  3. Return data

Relational vs Semi-Structured Tradeoffs

- **Relational Model**
  - Fixed schema
  - Flat data

- **Semi-Structured**
  - Self-described schema
  - Tree-structured data



Less well-defined/More flexible

- Basic retrieval process:
  1. Retrieve table
  2. Run through rows
  3. Return data

- Basic retrieval process:
  1. Retrieve document
  2. Parse document tree
  3. Return data

Inefficient encoding/Easy exchange of data

Note

- No database paradigm is "better" than another

- One-size does **not** fit all (M. Stonebraker)
  - Excellent article on data management in 21st century
  - http://cs.brown.edu/research/db/publications/fits_all.pdf

- Everything is getting mixed up anyways

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Types

Primitives include:

- String (in quotes)
- Numeric (unquoted number)
- Boolean (unquoted true/false)
- Null (literally just null)

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Types

Objects are an *unordered* collection of name-value pairs:

- "name": <value>
- Values can be primitives, objects, or arrays
- **Enclosed by { }**

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Types

Objects are an *unordered* collection of name-value pairs:

- "name": <value>
- Values can be primitives, objects, or arrays
- **Enclosed by { }**

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
    "book": [
        {
            "id": "01",
            "language": "Java",
            "author": "H. Javeson",
            "year": 2015
        },
        {
            "author": "E. Sepp",
            "id": "07",
            "language": "C++",
            "edition": null,
            "sale": true
        }
    ]
}
```

Types

Objects are an *unordered* collection of name-value pairs:

- "name": <value>
- Values can be primitives, objects, or arrays
- Enclosed by { }

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Types

Objects are an *unordered* collection of name-value pairs:

- "name": <value>
- Values can be primitives, objects, or arrays
- Enclosed by { }

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Types

Arrays are an *ordered* list of values:

- Order is preserved in interpretation
- May contain any mix of types
- Enclosed by [ ]

JSON Standard – Rules of the Game

- JavaScript Object Notation (JSON)
  - "Lightweight text-based open standard designed for **human-readable** data interchange"

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Types

Arrays are an *ordered* list of values:

- Order is preserved in interpretation
- May contain any mix of types
- Enclosed by [ ]

Read as “book”:[{object1}, {object2}]

Can have mix of types like
[{object1}, “string”, 124, {object2}]

JSON Standard – Rules of the Game

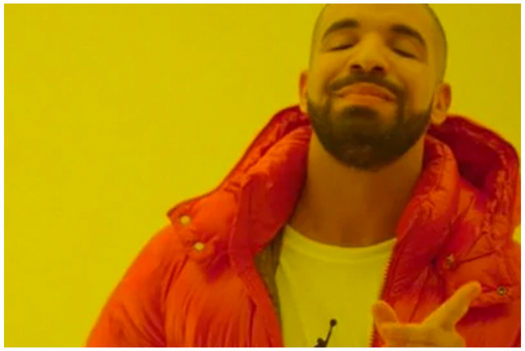
- JSON Standard too expressive
  - Implementations **restrict syntax**
  - Ex: Duplicate fields

```
{
   "id": "01",
   "language": "Java",
   "author": "H. Javeson",
   "author": "D. Suciu",
   "author": "A. Cheung",
   "year": 2015
}
```

JSON Standard – Rules of the Game

- JSON Standard too expressive
  - Implementations **restrict syntax**
  - Ex: Duplicate fields

```
{
   "id": "01",
   "language": "Java",
   "author": "H. Javeson",
   "author": "D. Suciu",
   "author": "A. Cheung",
   "year": 2015
}
```



```
{
   "id": "01",
   "language": "Java",
   "author": ["H. Javeson",
              "D. Suciu",
              "A. Cheung"],
   "year": 2015
}
```

Thinking About Semi-Structured Data

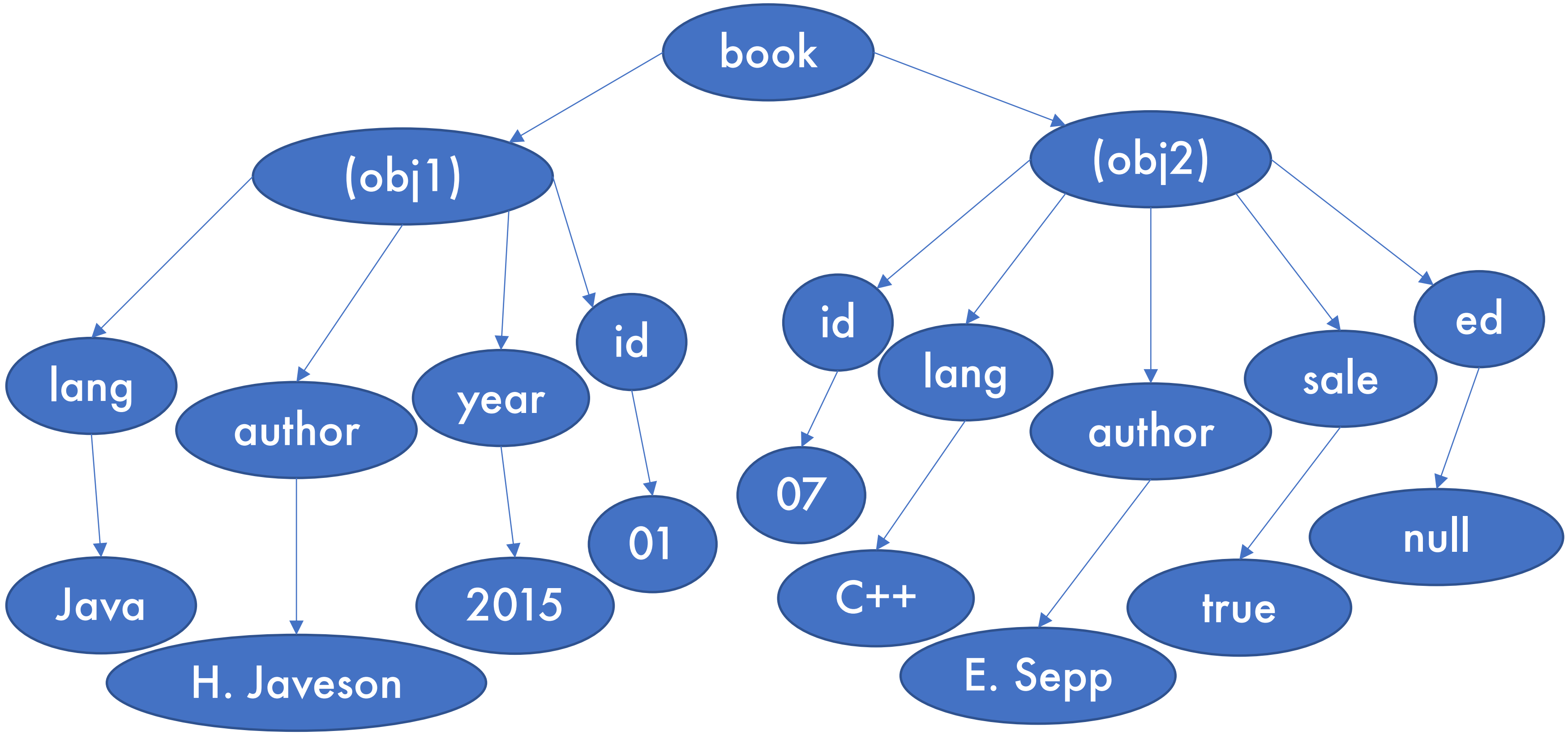
What does semi-structured data structure encode?

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": null,
         "sale": true
      }
   ]
}
```

Thinking About Semi-Structured Data

What does semi-structured data structure encode?

Tree semantics!

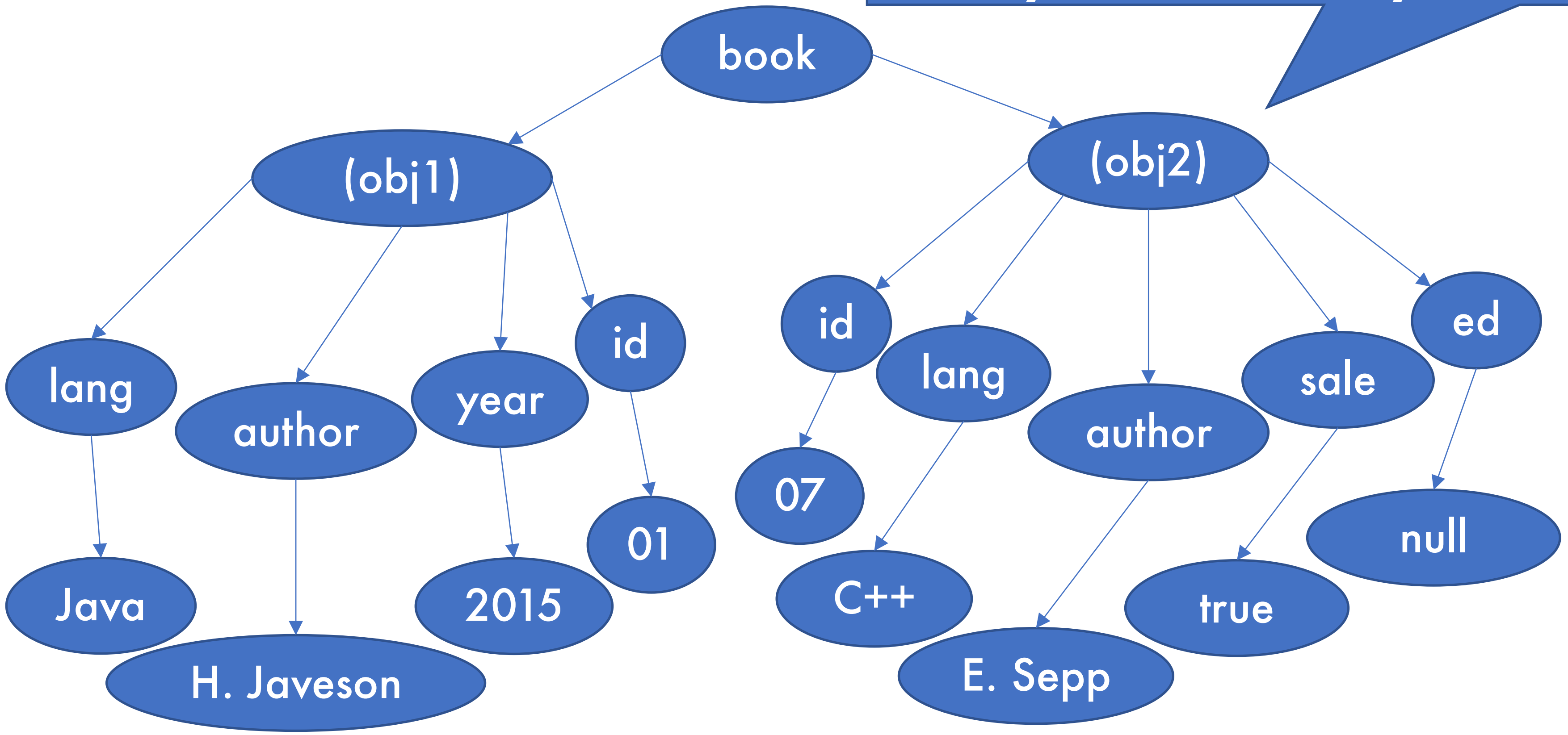


Thinking About Semi-Structured Data

What does semi-structured data structure encode?

Tree semantics!

These object don't have labels, as they are in an array



From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

What is a table in semi-structured land?

person

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

What is a table in semi-structured land?

person

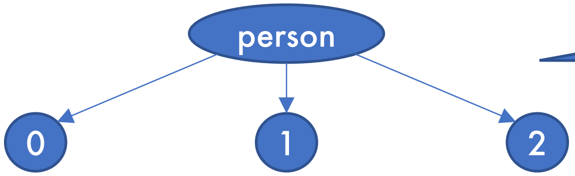
From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

```
{
    "person":[
        {
            …
        },
        {
            …
        },
        {
            …
        }
    ]
}
```

What is a table in semi-structured land?



Tables are just an array of elements (rows)

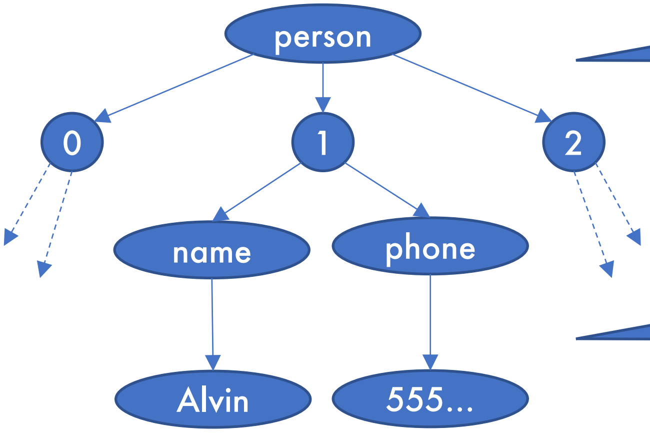
From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

```
{
    "person":[
        {
            …
        },
        {
            …
        },
        {
            …
        }
    ]
}
```

What is a table in semi-structured land?



Tables are just an array of elements (rows)

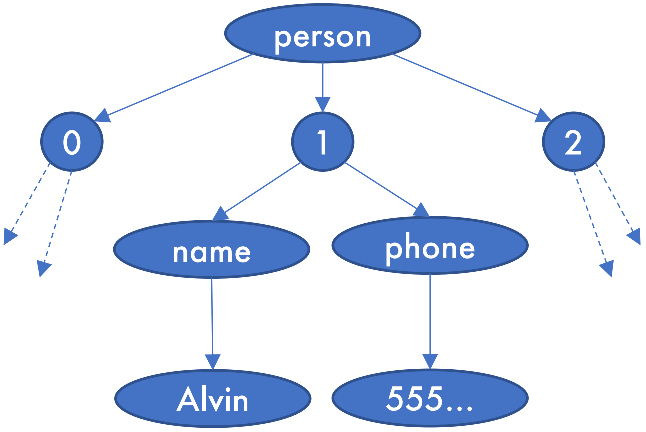
Rows are just simple (unnested) objects

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567"
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda",
         "phone": "555-345-6789"
      }
   ]
}
```



From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

How can NULL
be represented?

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567"
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda",
         "phone": "555-345-6789"
      }
   ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	NULL

How can NULL
be represented?

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567"
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda",
         "phone": "555-345-6789"
      }
   ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	NULL

How can NULL
be represented?

```
{
    "person":[
        {
            "name": "Dan",
            "phone": "555-123-4567"
        },
        {
            "name": "Alvin",
            "phone": "555-234-5678"
        },
        {
            "name": "Magda",
            "phone": null
        }
    ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	NULL

How can NULL
be represented?

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567"
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda"
      }
   ]
}
```

OK for field to
be missing!

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Are there things that the Relational Model can't represent?

```
{
    "person":[
        {
            "name": "Dan",
            "phone": "555-123-4567"
        },
        {
            "name": "Alvin",
            "phone": "555-234-5678"
        },
        {
            "name": "Magda",
            "phone": "555-345-6789"
        }
    ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567"
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda",
         "phone": "555-345-6789"
      }
   ]
}
```

Are there things that the Relational Model can't represent?

Non-flat data!

- Array data
- Multi-part data

From Relational to Semi-Structured

Person

Name	Phone
Dan	???
Alvin	555-234-5678
Magda	555-345-6789

Are there things that the Relational Model can’t represent?

Non-flat data!

- **Array data**
- Multi-part data

Array with 2 objects

```
{
   "person":[
      {
         "name": "Dan",
         "phone": [
            "555-123-4567",
            "555-987-6543"
         ]
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda",
         "phone": "555-345-6789"
      }
   ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
???	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Are there things that the Relational Model can’t represent?

Non-flat data!

- Array data
- **Multi-part data**

Object with 2 key-value pairs

```
{
   "person":[
      {
         "name": {
             "fname": "Dan",
             "lname": "Suciu"
          },
         "phone": "555-123-4567"
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678"
      },
      {
         "name": "Magda",
         "phone": "555-345-6789"
      }
   ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

How do we represent foreign keys?

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567",
         "orders": [
            {
               "date": 1997,
               "product": "Furby"
            }
         ]
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678",
         "orders": [
            {
               "date": 2000,
               "product": "Furby"
            },
            {
               "date": 2012,
               "product": "Magic8"
            }
         ]
      },
      {
         "name": "Magda",
         "phone": "555-345-6789",
         "orders": []
      }
   ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Precomputed equijoin!

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567",
         "orders": [
            {
               "date": 1997,
               "product": "Furby"
            }
         ]
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678",
         "orders": [
            {
               "date": 2000,
               "product": "Furby"
            },
            {
               "date": 2012,
               "product": "Magic8"
            }
         ]
      },
      {
         "name": "Magda",
         "phone": "555-345-6789",
         "orders": []
      }
   ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

Nest the data?
Person → Orders → Product

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

Nest the data?
Person → Orders → Product

We might miss some products!
&
Product data will be duplicated!

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

Nest the data?
Product → Orders → Person

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

Nest the data?
Product → Orders → Person

We might miss some people!
&
People data will be duplicated!

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

Convert each table to a separate array/document?

From Relational to Semi-Structured

Person

Name	Phone
Dan	555-123-4567
Alvin	555-234-5678
Magda	555-345-6789

Orders

PName	Date	Product
Dan	1997	Furby
Alvin	2000	Furby
Alvin	2012	Magic8

Product

ProdName	Price
Furby	9.99
Magic8	15.99
Tomagachi	18.99

Is this many-to-many relationship easily convertible to JSON?

Convert each table to a separate array/document?

We wanted to avoid joining in the first place!

From Relational to Semi-Structured

Big ideas:

- Semi-structured data is **parsed**
  - Data model flexibility
  - Potentially lots of redundancy
- Semi-structured data expresses **unique patterns**
  - Collection/multi-part data
  - Precompute joins
- Semi-structured data **has limits**
  - Relies on relational-like patterns in some situations

A Semi-structured DBMS

- AsterixDB as a case study of Document Store
  - Semi-structured data model in JSON
  - **Introducing AsterixDB and SQL++**

The 5 W's of AsterixDB

- Who
  - M. J. Carey & co.
- What
  - "A Scalable, Open Source BDMS"
  - It is now also an Apache project
- Where
  - UC Irvine, Cloudera Inc, Google, IBM, …
- When
  - 2014
- Why
  - To develop a next-gen system for managing semi-structured data

The 5 W's of SQL++

- Who
  - K. W. Ong & Y. Papakonstantinou
- What
  - A query language that is applicable to JSON native stores and SQL databases
- Where
  - UC San Diego
- When
  - 2015
- Why
  - Stand in for other semi-structured query languages that lack formal semantics.

Why We are Choosing SQL++

- Strong formal semantics
  - Original paper: https://arxiv.org/pdf/1405.3631.pdf
  - Nested relational algebra: https://dl.acm.org/citation.cfm?id=588133
- Many systems adopting or converging to SQL++
  - Apache AsterixDB
  - CouchBase (N1QL)
  - Apache Drill
  - Snowflake

Asterix Data Model (ADM)

- Nearly identical to the JSON standard
- Some additions
  - New primitive: **universally unique identifier (uuid)**
    - Ex: 123e4567-e89b-12d3-a456-426655440000
  - New derived type: **multiset**
    - Like an array but unordered
    - Encapsulated by double curly braces {{ }}

- Queried data must be a multiset or array

Introducing the New and Improved SQL++

SQL++ Mini Demo

General Installation (Details in HW7 spec)

Download from:
https://asterixdb.apache.org/download.html

Start local cluster from:
<asterix root>/opt/local/bin/start-sample-cluster

Use web browser for interaction, default address:
127.0.0.1:19002

Don't forget to stop cluster when you're done:
<asterix root>/opt/local/bin/stop-sample-cluster

SQL++ Mini Demo

General Usage:

Everything is running locally so make sure your computer doesn't die (advise against SELECT *)

Don't use attu, previous quarters people accidentally used other people's instance

Learn something! I dare say that SQL++ is a model for many future query languages.

SQL++ Hello World

```
SELECT x.phone
  FROM [
          {"name": "Dan", "phone": [300, 150]},
          {"name": "Alvin", "phone": 420}
       ] AS x;

-- output, same for-loop semantics like in SQL
-- array data
/*
{ "phone": [300, 150] }
{ "phone": 420 }
*/
```

SQL++ Hello World

```
SELECT x.phone
  FROM {{
          {"name": "Dan", "phone": [300, 150]},
          {"name": "Alvin", "phone": 420}
       }} AS x;

-- same output as array data
-- multiset data
```

SQL++ Hello World

```
-- error
SELECT x.phone
  FROM {"name": "Dan", "phone": [300, 150]} AS x;

-- output
-- trying to query an object
/*
Type mismatch: function scan-collection expects its
1st input parameter to be type multiset or array,
but the actual input type is object
[TypeMismatchException]
*/
```

SQL++ Hello World

```
SELECT x.phone
  FROM [
          {"name": "Dan", "phone": [300, 150]},
          {"name": "Alvin", "phone": null}
       ] AS x;

-- output, null works like in SQL
-- null values
/*
{ "phone": [300, 150] }
{ "phone": null }
*/
```

SQL++ Hello World

```
SELECT x.phone
  FROM [
          {"name": "Dan", "phone": [300, 150]},
          {"name": "Alvin"}
       ] AS x;

-- output, missing data is simply passed over (beware of typos!)
-- missing values
/*
{ "phone": [300, 150] }
{ }
*/
```

SQL++ Hello World

```
SELECT x.fone -- intentional typo
  FROM [
          {"name": "Dan", "phone": [300, 150]},
          {"name": "Alvin", "phone": 420}
       ] AS x;

-- output, beware of typos! No errors are thrown
/*
{ }
{ }
*/
```

SQL++ Hello World

```
  FROM [
          {"name": "Dan", "phone": [300, 150]},
          {"name": "Alvin", "phone": 420}
       ] AS x
 WHERE is_array(x.phone) OR x.phone > 100
 GROUP BY x.name, x.phone
HAVING x.name = "Dan" OR x.name = "Alvin"
SELECT x.phone
 ORDER BY x.name DESC;

-- output, finally the keyword order matches FWGHOS!
/*
{ "phone": [300, 150] }
{ "phone": 420 }
*/
```