Introduction to Data Management

Semi-Structured Data

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
## Recap: OLTP vs OLAP

<table>
<thead>
<tr>
<th>OLTP (Online Transaction Processing)</th>
<th>OLAP (Online Analytical Processing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction-heavy workloads</td>
<td>Complex query workloads</td>
</tr>
<tr>
<td>Many simple lookup or single-join queries</td>
<td>Many joins, aggregations, etc.</td>
</tr>
<tr>
<td>Many small updates and inserts</td>
<td>Little to no updates</td>
</tr>
<tr>
<td>Managing consistency is critical</td>
<td>Query optimization and processing is critical</td>
</tr>
<tr>
<td>Flights, banking, etc. (many users)</td>
<td>Business intelligence (few users)</td>
</tr>
</tbody>
</table>
Recap: NoSQL in a Nutshell

- NoSQL → Looser data model
  - Give up built-in OLAP/analysis functionality
  - Give up built-in ACID consistency
Recap: Let’s Drop ACID

- RDBMSs have the ACID consistency model
- NoSQL sys. have the **BASE** consistency model

**Basically Available**
- Most failures do not cause a complete system outage

**Soft state**
- System is not always write-consistent

**Eventually consistent**
- Data will eventually converge to agreed values
Why the Sacrifice?

Why can’t we have both Consistency and Availability?
CAP Theorem

- Old name: Brewer’s Conjecture
- In a distributed data store, one can only provide two of the following three guarantees:
  - **Consistency**
    - Every read receives the most recent write or an error
  - **Availability**
    - Every request must respond with a non-error
  - **Partition tolerance**
    - Continued operation in presence of dropped or delayed messages
RDBMS vs NoSQL Systems

- Distributed RDBMS
  - Partition tolerance + **Consistency**
- NoSQL Systems
  - Partition tolerance + **Availability**
RDBMS vs NoSQL Systems

- Distributed RDBMS
  - Partition tolerance + **Consistency**

- NoSQL Systems
  - Partition tolerance + **Availability**

Both must provide partition tolerance by virtue of being distributed systems.
Partition tolerance + **Consistency**
Partition tolerance + **Consistency**

![Diagram showing DB Node 1 and DB Node 2 connected to a Client]

- DB Node 1
- DB Node 2
- Client

$V_0$
RDBMS vs NoSQL Systems

Partition tolerance + **Consistency**

![Diagram showing partition tolerance and consistency in NoSQL systems]

- DB Node 1
- DB Node 2
- Client

Write $V_1$
RDBMS vs NoSQL Systems

Partition tolerance + **Consistency**

- **DB Node 1**: $V_1$
- **DB Node 2**: $V_0$
- **Client**
Partition tolerance + Consistency

DB Node 1

DB Node 2

Client

V₁

V₀

Done
RDBMS vs NoSQL Systems

Partition tolerance + Consistency

DB Node 1

Client

DB Node 2

$V_1$

$V_0$
Partition tolerance + **Consistency**
Partition tolerance + **Consistency**

DB Node 1: $V_1$

DB Node 2: $V_0$

Client

Tries but fails to check consistency of $V$
Partition tolerance + **Consistency**
Partition tolerance + **Consistency**

DB Node 1

DB Node 2

Client

V₁

V₀

Error/Timeout

Consistent!
But not available.
Partition tolerance + **Availability**
Partition tolerance + **Availability**

- **DB Node 1**: $V_0$
- **DB Node 2**: $V_0$
- **Client**
Partition tolerance + **Availability**

- DB Node 1: $V_0$
- DB Node 2: $V_0$
- Client
  - Write $V_1$
RDBMS vs NoSQL Systems

Partition tolerance + **Availability**
Partition tolerance + *Availability*
Partition tolerance + **Availability**
RDBMS vs NoSQL Systems

Partition tolerance + Availability

DB Node 1

DB Node 2

Client

V₁

V₀

Read V
RDBMS vs NoSQL Systems

Partition tolerance + **Availability**

DB Node 1

$V_1$

DB Node 2

$V_0$

Client

Tries but fails to check consistency of $V$
Partition tolerance + Availability
Partition tolerance + **Availability**

- DB Node 1: $V_1$
- DB Node 2: $V_0$
- Client: Available!

But not consistent.
**RDBMS vs NoSQL Systems**

Partition tolerance + *Availability*

**Diagram:**
- **DB Node 1**
  - $V_1$
- **Client**
- **DB Node 2**
  - $V_0$

**Importance:**
These are only cases when the network infrastructure goes down. Usually, nodes should be able to check on other nodes.
Proof of CAP Theorem

- **2002 original paper (S. Gilbert & N. Lynch)**
- **More digestible blog post (M. Whittaker)**

Proof by contradiction: Assume we had a system that guaranteed availability, consistency, and partition tolerance...
Proof of CAP Theorem

Partition tolerance + Consistency + Availability?

DB Node 1

\[ V_1 \]

Client

DB Node 2

\[ V_0 \]

Error/Timeout

Violates availability!
Proof of CAP Theorem

Partition tolerance + Availability + Consistency?

DB Node 1

V₁

Client

DB Node 2

V₀

V₀

Violates consistency!
On A Practical Note

- RDBMSs are intended to be highly consistent
  - Boost availability by sacrificing some consistency
- NoSQL systems are intended to be highly available
  - Boost consistency by sacrificing some availability
- Most applications OK with some compromise
  - “Return most of data most of the time”
  - DBMS choice has many factors
    - Consistency/Availability requirements
    - Scalability
    - Usability
    - OLAP/analysis requirements
    - ...
NoSQL Data Models

Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA,BBB,CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA,BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA,DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA,2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3,ZZZ,5623</td>
</tr>
</tbody>
</table>

Wide-Column Store

(Extensible Record Store)

Graph Database

Document Store

XML

```xml
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

JSON

```json
{
  "empinfo": {
    "employees": [
      {
        "name": "James Kirk",
        "age": 40
      },
      {
        "name": "Jean-Luc Picard",
        "age": 45
      },
      {
        "name": "Wesley Crusher",
        "age": 27
      }
    ]
  }
}
```
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

Tags surround the respective data
Semi-Structured Documents

- Some notion of **tagging** to mark down semantics

- Examples:
  - XML
  - Protobuf
  - JSON

Not human readable in serialized format
Semi-Structured Documents

▪ Some notion of **tagging** to mark down semantics

▪ Examples:
  - XML
  - Protobuf
  - JSON

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

Tags introduce the respective data

```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"
        }
      ]
    }
  ]
}
```
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](image)

- 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

- 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

Less well-defined / More flexible

Inefficient encoding / Easy exchange of data
### JavaScript Object Notation (JSON)

- "Lightweight text-based open standard designed for **human-readable** data interchange"

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

- "Lightweight text-based open standard designed for human-readable data interchange"

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

**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  
    }  
  ]
}
```

**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
    }
  ]
}
```

**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
    }
  ]
}
```

**Arrays** are an *ordered* list of values:
- Order is preserved in interpretation
- May contain any mix of types
- Enclosed by [ ]
- JSON Standard too expressive
  - Implementations restrict syntax
  - Ex: Duplicate fields

```json
{
  "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

```json
{  
  "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?

```json
{
   "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!**
What does semi-structured data structure encode?

**Tree semantics!**

These objects don't have labels, as they are in an array.
### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

What is a table in semi-structured land?
From Relational to Semi-Structured

Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

What is a table in semi-structured land?

Tables are just an array of elements (rows)
What is a table in semi-structured land?

Tables are just an array of elements (rows)

Rows are just simple (unnested) objects

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>
### From Relational to Semi-Structured

**Person**

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

What is a table in semi-structured land?

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

How can NULL be represented?

```json
{
   "person": [  
   {
      "name": "Dan",
      "phone": "555-123-4567"
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678"
   },
   {
      "name": "Magda",
      "phone": "555-345-6789"
   }
   ]
}
```
From Relational to Semi-Structured

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>NULL</td>
</tr>
</tbody>
</table>

How can NULL be represented?

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>NULL</td>
</tr>
</tbody>
</table>

How can NULL be represented?

```json
{
    "person":[
      {
        "name": "Dan",
        "phone": "555-123-4567"
      },
      {
        "name": "Alvin",
        "phone": "555-234-5678"
      },
      {
        "name": "Magda",
        "phone": null
      }
    ]
}
```
How can NULL be represented?

```json
{
    "person":[
        {
            "name": "Dan",
            "phone": "555-123-4567"
        },
        {
            "name": "Alvin",
            "phone": "555-234-5678"
        },
        {
            "name": "Magda"
        }
    ]
}
OK for field to be missing!
```
Are there things that the Relational Model can’t represent?

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

```json

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

```
From Relational to Semi-Structured

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

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

Non-flat data!
- Array data
- Multi-part data
- Heterogeneous collections

```json
{
    "person":[
        {
            "name": "Dan",
            "phone": "555-123-4567"
        },
        {
            "name": "Alvin",
            "phone": "555-234-5678"
        },
        {
            "name": "Magda",
            "phone": "555-345-6789"
        }
    ]
}
```
From Relational to Semi-Structured

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>???</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

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

Non-flat data!
- **Array data**
- Multi-part data
- Heterogeneous collections

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

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

Non-flat data!
- Array data
- **Multi-part data**
- Heterogeneous collections

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>???</td>
</tr>
</tbody>
</table>

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

Non-flat data!
- Array data
- Multi-part data
- **Heterogeneous collections**
### From Relational to Semi-Structured

**Person**

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

**Orders**

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

How do we represent foreign keys?
<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

```json
{
  "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 Data

#### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

#### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

Precomputed equijoin!
### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

### Product

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>
### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

### Product

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>

Is this many-to-many relationship easily convertible to JSON? Nest the data? Person ⊑ Orders ⊑ Product
## From Relational to Semi-Structured

### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

### Product

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>

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 Data

#### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

#### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

#### Product

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>

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

Nest the data?

Product  ▸ Orders  ▸ Person
## From Relational to Semi-Structured Data

### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

### Product

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>

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 Data

#### Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

#### Orders

<table>
<thead>
<tr>
<th>PName</th>
<th>Date</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

#### Price

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>

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

Convert each table to a separate array/document?
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!

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Phone</td>
</tr>
<tr>
<td>Dan</td>
<td>555-123-4567</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-234-5678</td>
</tr>
<tr>
<td>Magda</td>
<td>555-345-6789</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Orders</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PName</td>
<td>Date</td>
</tr>
<tr>
<td>Dan</td>
<td>1997</td>
</tr>
<tr>
<td>Alvin</td>
<td>2000</td>
</tr>
<tr>
<td>Alvin</td>
<td>2012</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProdName</td>
<td>Price</td>
</tr>
<tr>
<td>Furby</td>
<td>9.99</td>
</tr>
<tr>
<td>Magic8</td>
<td>15.99</td>
</tr>
<tr>
<td>Tomagachi</td>
<td>18.99</td>
</tr>
</tbody>
</table>
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
Next time

- AsterixDB as a case study of Document Store
  - Introducing AsterixDB and SQL++