

Introduction to Data Management SQL++

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

Announcements

- **No in-person Lecture on Friday, May 24:**
Please watch recorded video instead (on canvas)
- HW6/2 due on Friday
- HW7 to be released on Friday, due on May 31

Course Evals

- **Opens on Saturday, May 25**
- Please take a few minutes before we start to fill out the course evals
- I read every word of your comments and make adjustments where needed based on the feedback; have done this in the past

Recap: semistructure data

- Loose terminology; any "parsable" file qualifies
- Self-describing, "data first"
- We discuss only Json
- Other formats: protobuf, XML, csv

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

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

- 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

NOT ALLOWED
(duplicated authors)

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

OK
(author array)

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

Thinking About Semi-Structured Data

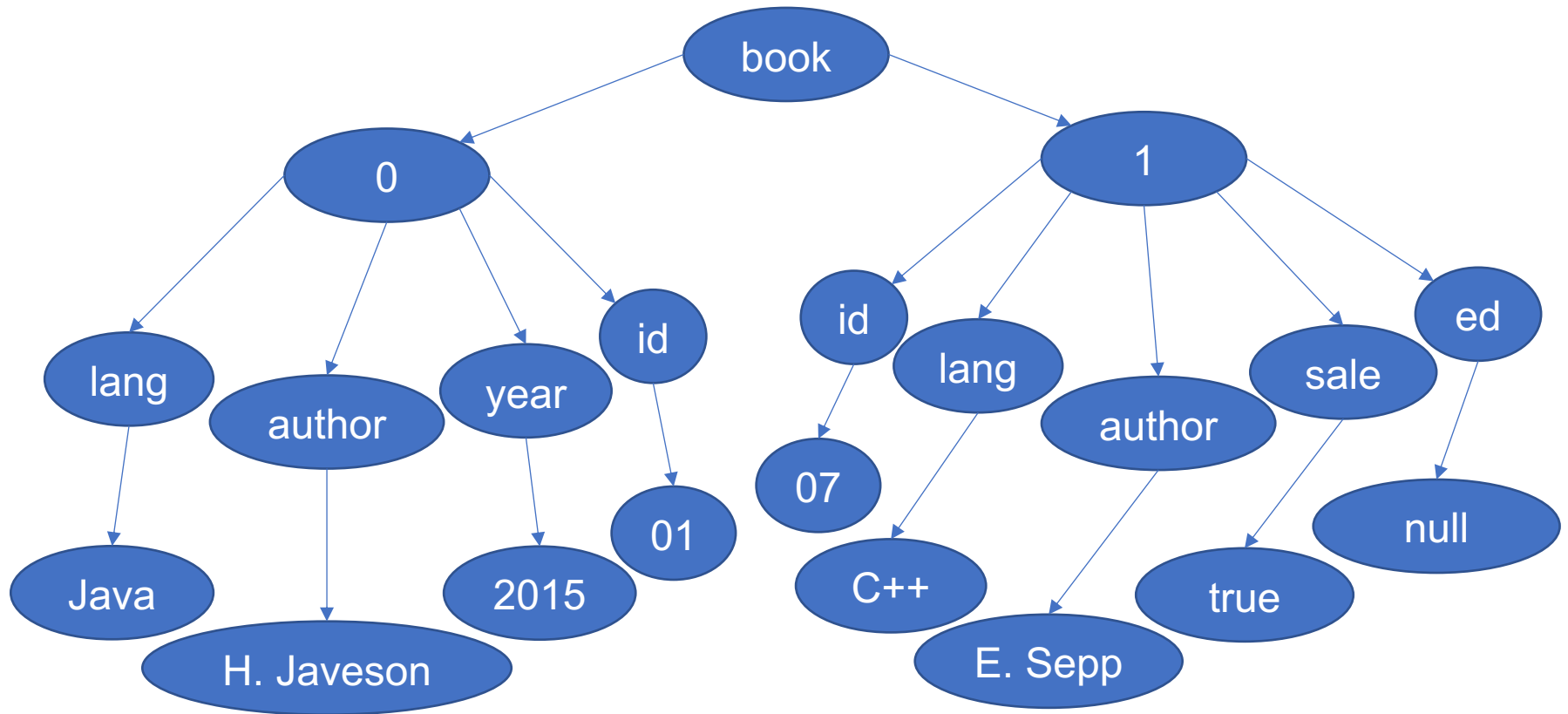
What is the data model of semistructured data?

```
{
  "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 is the data model of semistructured data?

A Tree!



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?

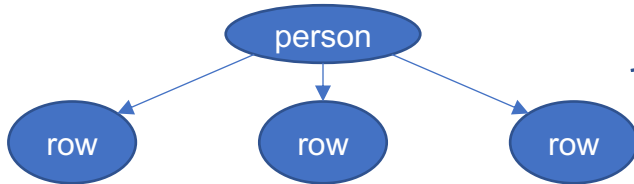
person

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?



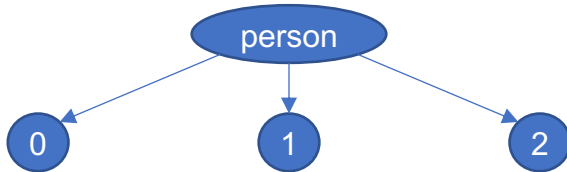
A table is just a collection of rows

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?



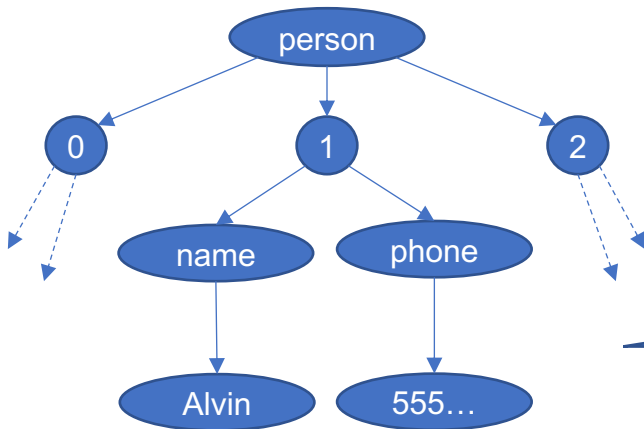
Or, an array of rows (elements)

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?



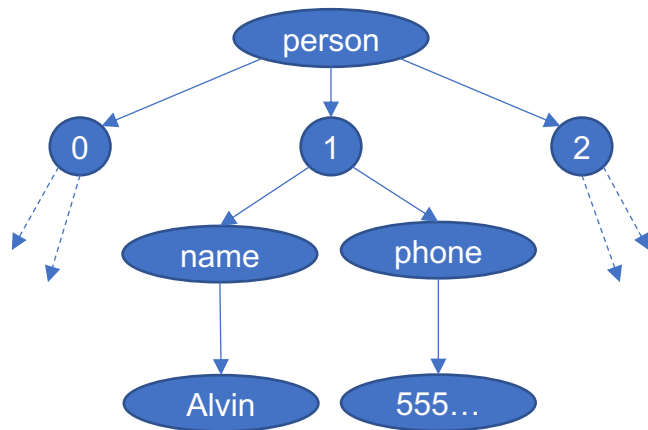
Or, an array of rows (elements)

Rows are 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

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

Non-flat data!

- Array data
- Multi-part data

```
{
  "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	???
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

```
{
  "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**

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

Discussion

- Relational data is represented using shallow, regular trees
- NULLs can be encoded either as NULL, or by simply omitting the attribute
- The value of an attribute is no longer restricted to be an atomic type: it can be a record, an array, ...

How do we represent relationships?

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
a many-to-one relationship?

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

How do we represent
a many-to-many relationship?

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

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

How do we represent
a many-to-many relationship?

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

How do we represent
a many-to-many relationship?

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

How do we represent
a many-to-many relationship?

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

How do we represent
a many-to-many relationship?

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

How do we represent
a many-to-many relationship?

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

How do we represent
a many-to-many relationship?

Convert each table to a
separate array/document?

We wanted to avoid joining
in the first place!

Summary of Semistructured Data

- **Self-describing**

- Data and its schema presented together

- **Irregular/flexible**

- Missing attributes
- Repeated attributes (or arrays)
- Attribute may have different types in different objects

- **1-to-many relationships:** very natural

- **Many-many relationships:** cumbersome

AsterixDB and SQL++

- 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 DBMS"
 - 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 foundations
 - 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
 - Amazon Partiql, <https://partiql.org/>

* There are much better papers on Nested Relational Algebra (ask DanS)

Asterix Data Model (ADM)

- ADM = nearly identical to JSON
- Adds: **multiset** or **bag**
 - Encapsulated by double curly braces `{{ }}`
- Adds: **universally unique identifier (uuid)**
 - Ex: 123e4567-e89b-12d3-a456-426655440000
 - Useful for auto-generating unique keys

Introducing the New and Improved SQL++



SQL++ Mini Demo

Demo Time!

Installing AsterixDB

(Details in HW7 spec)

Download from

<https://asterixdb.apache.org/download.html>

Start local cluster from:

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

Run by typing this in your browser:

127.0.0.1:19002

Stop cluster when you're done:

<asterix root>/opt/local/bin/stop-sample-cluster

SQL++ Hello World

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

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;
```

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;
```

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;
```

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;
```


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;
```

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:

```
/*
{ "phone": [300, 150] }
{ "phone": 420 }
*/
```

Next Time

- Patterns in querying semi-structured data
- SQL++ behind the mask

