Announcements

• Office hours changes this week
  – Check schedule
• HW 4 due next Tuesday
  – Start early
• WQ 4 due tomorrow

JSON - Overview

• JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.

• The filename extension is .json.

We will emphasize JSON as semi-structured data

JSON Terminology

• Data is represented in name/value pairs.
• Curly braces hold objects
  – Each object is a list of name/value pairs separated by , (comma)
  – Each pair is a name followed by ':' (colon) followed by the value
• Square brackets hold arrays and values are separated by , (comma).

JSON Syntax

```
"book": [{
  "id": "95",
  "language": "Java",
  "author": "H. Javanon",
  "year": 2015
},
{"id": "97",
 "language": "C++",
 "edition": "second",
 "author": "E. Sepp",
 "price": 22.25
}
]
```

JSON Data Structures

• Objects, i.e., collections of name-value pairs:
  – { "name1": value1, "name2": value2, ...}
  – "name" is also called a "key"

• Ordered lists of values:
  – [obj1, obj2, obj3, ...]
Avoid Using Duplicate Keys

The standard allows them, but many implementations don’t

JSON Primitive Datatypes

- Number
- String
  - Denoted by double quotes
- Boolean
  - Either true or false
- null/empty

JSON Semantics: a Tree!

Recall: arrays are ordered in JSON!

JSON Data

- JSON is self-describing
- Schema elements become part of the data
  - Relational schema: person(name, phone)
  - In JSON "person", "name", "phone" are part of the data, and are repeated many times
- Consequence: JSON is much more flexible
- JSON = semistructured data

Mapping Relational Data to JSON
Mapping Relational Data to JSON

May inline multiple relations based on foreign keys

<table>
<thead>
<tr>
<th>Person</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>personName</td>
</tr>
<tr>
<td>John</td>
<td>John</td>
</tr>
<tr>
<td>Sue</td>
<td>Sue</td>
</tr>
</tbody>
</table>

```json
{ "Person": [{ "name": "John", "phone": 3646, "Orders": [{ "date": 2002, "product": "Gizmo" }, { "date": 2004, "product": "Gadget" }] }, { "name": "Sue", "phone": 6343, "Orders": [{ "date": 2002, "product": "Gadget" }] }] }
```

Discussion: Why Semi-Structured Data?

- Semi-structured data model is good as data exchange formats
  - i.e., exchanging data between different apps
  - Examples: XML, JSON, Protobuf (protocol buffers)
- Increasingly, systems use them as a data model for databases:
  - SQL Server supports for XML-valued relations
  - CouchBase, MongoDB: JSON as data model
  - Dremel (BigQuery): Protobuf as data model

Query Languages for Semi-Structured Data

- XML: XPath, XQuery (see textbook)
  - Supported inside many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines
- Protobuf: SQL-ish language (Dremel) used internally by Google, and externally in BigQuery
- JSON:
  - CouchBase: N1QL
  - Asterix: SQL++ (based on SQL)
  - MongoDB: has a pattern-based language
  - JSONiq: http://www.jsoniq.org/

Asterix Data Model (ADM)

- Based on the Json standard
- Objects:
  - ("Name": "Alice", "age": 40)
  - Fields must be distinct:
    - ("Name": "Alice", "age": 40, "age": 50)
- Ordered arrays:
  - [1, 3, "Fred", 2, 9]
  - Can contain values of different types
- Multisets (aka bags):
  - {{1, 3, "Fred", 2, 9}}
  - Mostly internal use only but can be used as inputs
  - All multisets are converted into ordered arrays (in arbitrary order) when returned at the end

Examples

What do these queries return?

```sql
SELECT x.phone FROM [{"name": "Alice", "phone": [300, 158]}] AS x;
```

```
SELECT x.phone FROM [{ ("name": "Alice", "phone": [300, 158]) }] AS x;
```

```sql
ERROR
SELECT x.phone FROM ("name": "Alice", "phone": [300, 158]) AS x;
```

- Can only query from multi-set or array (not objects)
Datatypes

- Boolean, integer, float (various precisions), geometry (point, line, ...), date, time, etc
- UUID = universally unique identifier
Use it as a system-generated unique key

null v.s. missing

- "age": null = the value NULL (like in SQL)
- "age": missing = { } = really missing

Finally, a language that we can use!

```
SELECT x.age FROM Person AS x WHERE x.age > 21 GROUP BY x.gender HAVING x.salary > 10000 ORDER BY x.name;
```

is exactly the same as

```
SELECT x.age FROM Person AS x WHERE x.age > 21 GROUP BY x.gender HAVING x.salary > 10000 SELECT x.age ORDER BY x.name;
```

SQL++ Overview

- Data Definition Language: create a
  - Type
  - Dataset (like a relation)
  - Dataverse (a collection of datasets)
  - Index
    * For speeding up query execution

- Data Manipulation Language: SELECT-FROM-WHERE

Dataverse

A Dataverse is a Database
(i.e., collection of tables)

CREATE DATaverse myDB
CREATE DATaverse myDB IF NOT EXISTS

DROP DATaverse myDB
DROP DATaverse myDB IF EXISTS

USE myDB
### Closed Types

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name: string,
  age: int,
  email: string?
}

{"name": "Alice", "age": 30, "email": "a@alice.com"}
{"name": "Bob", "age": 40}
```

---

### Open Types

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
  name: string,
  age: int,
  email: string?
}

{"name": "Alice", "age": 30, "email": "a@alice.com"}
{"name": "Bob", "age": 40}
```

---

### Types with Nested Collections

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  Name : string,
  phone: [string]
}

{"Name": "Carol", "phone": ["1234"]}
{"Name": "David", "phone": ["2345", "6789"]}
{"Name": "Evan", "phone": []}
```

---

### Dataset

- Dataset = relation
- Must have a type
  - Can be a trivial OPEN type
- Must have a key
  - Can also be a trivial one

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name: string,
  email: string?
}

USE myDB;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
PRIMARY KEY Name;
```

---

### Dataset with Existing Key

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name: string,
  phone: [string]
}

{"name": "Alice"}
{"name": "Bob"}
```

---

### Type

- Defines the schema of a collection
- It lists all required fields
- Fields followed by ? are optional
- CLOSED type = no other fields allowed
- OPEN type = other fields allowed
Dataset with Auto Generated Key

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  myKey: uuid,
  Name : string,
  email: string?
}

USE myDB;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
  PRIMARY KEY myKey AUTOGENERATED;
```

This is no longer 1NF

- NFNF = Non First Normal Form
- One or more attributes contain a collection
- One extreme: a single row with a huge, nested collection
- Better: multiple rows, reduced number of nested collections