Database Systems
CSE 414

Lecture 12-13: JSON and SQL++
(mostly not in textbook)

NoSQL (cont)

JSON (cont.)

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 = semi-structured data

Mapping Relational Data to JSON
Mapping Relational Data to JSON

May inline foreign keys

<table>
<thead>
<tr>
<th>Person</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>date</td>
</tr>
<tr>
<td>John</td>
<td>2002</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3646</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
</tbody>
</table>

JSON = Semi-structured Data (1/3)

- Missing attributes:

```json
{  "name": "John",  "phone": 1234,  "Orders": [  {"date": 2002,  "product": "Gizmo"},  {"date": 2004,  "product": "Gadget"}  ]}
```

- Could represent a table with nulls

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1234</td>
</tr>
<tr>
<td>Joe</td>
<td>-</td>
</tr>
</tbody>
</table>

JSON = Semi-structured Data (2/3)

- Repeated attributes

```json
{  "name": "John",  "phone": 1234,  "Orders": [  {"date": 2002,  "product": "Gizmo"},  {"date": 2004,  "product": "Gadget"}  ]}
```

- Impossible in one table:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>2345</td>
</tr>
<tr>
<td>Mary</td>
<td>3456</td>
</tr>
</tbody>
</table>

JSON = Semi-structured Data (3/3)

- Attributes with different types in different objects

```json
{  "name": "Sue",  "phone": 3456,  "Orders": [  {"date": 2002,  "product": "Gadget"}  ]}
```

- Nested collections
- Heterogeneous collections

Discussion

- Data exchange formats
  - well suited for exchanging data between apps
  - XML, JSON, Protobuf
- Increasingly, some systems use them as a data model:
  - 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 end of lecture, textbook)
  - Supported inside many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines
- Protobuf: used internally by google, and externally in BigQuery. similar to compiled JSON
- JSON:
  - CouchBase: N1QL
  - MongoDB: has a pattern-based language
  - JSONiq http://www.jsoniq.org/
  - AsterixDB: AQL and SQL++
AsterixDB

- NoSQL database system (document store)
- Developed at UC Irvine
  - Now an Apache project
- Designed to be installed on a cluster
  - multiple machines (nodes) together implement the DBMS
  - allows scale to much larger amounts of data
- Weak support for multi-node transactions
- Good support for multi-node queries

AsterixDB (cont.)

- Data is partitioned over nodes by primary key
  - queries involve not only disk, but also network I/O
- Supports advanced queries
  - joins
  - nested queries
  - grouping and aggregation
- No statistics maintained yet (per docs)
  - may need more hints to get good performance
  - expect this to improve

AQL and SQL++

- Asterix’s own query language is AQL
  - based on XQuery (for XML)
- SQL++
  - SQL-like syntax for AQL
  - more familiar to database users

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {"Name": "Alice", "age": 40}
  - Fields must be distinct:
    - {"Name": "Alice", "age": 40, "age": 50}
- Arrays:
  - [1, 3, "Fred", 2, 9]
  - Note: can be heterogeneous
- Bags:
  - [{1, 3, "Fred", "Fred", 2, 9}]

Examples

Try these queries yourself:

```
SELECT age FROM [{name: 'Alice', age: ['30', '50']}] x;
```

```
SELECT age FROM [{name: 'Alice', age: ["30", '50']}] x;
```

`-- error`

```
SELECT age FROM {'name': 'Alice', 'age': ['30', '50']} x;
```

`-- error`

```
SELECT age FROM {'name': 'Alice', 'age': ["30", '50']};
```
Data Types

- Supports SQL / JSON data type:
  - boolean, integer, float (various precisions), null
- Some SQL types not in JSON:
  - date, time, interval
- Some new types:
  - geometry (point, line, …)
  - UUID = universally unique identifier
  (systems generated, globally unique key)

Null vs. Missing

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

```
SELECT x.b FROM [{'a': 1, 'b': 2}, {'a': 3}] x;
```
```
{ "b": 2 }
{ }
```

```
SELECT x.b FROM [{'a': 1, 'b': 2}, {'a': 3, 'b': missing}] x;
```
```
{ "b": 2 }
{ }
```

SQL++ Overview

- Data definition language:
  - Dataverse (= database)
  - Dataset (= table)
    - each row uses a declared Type
- Types
  - declares the required parts
  - can allow for extra data (open vs. closed types)
- Indexes
- Query language: select-from-where

Dataverse

A Dataverse is a Database

- CREATE DATaverse lec16
- CREATE DATaverse lec16 IF NOT EXISTS
- DROP DATaverse lec16
- DROP DATaverse lec16 IF EXISTS
- USE lec16

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

Closed Types

USE lec16;
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}

-- not OK:
{"Name": "Carol", "age": 35, "phone": "1234567890"}
Open Types

```
USE lec16;
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}
```

-- Now it's OK:
```
{"Name": "Carol", "age": 35, "phone": "123456789"}
```

Types with Nested Collections

```
USE lec16;
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": []}
```

Datasets

- Dataset = relation
- Must have a type
  - can be a trivial OPEN type
- Must have a key
  - can be declared "autogenerated" if UUID
  - (SQL systems usually support auto-incremented unique integer IDs)

```
USE lec16;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;

{"Name": "Alice"}
{"Name": "Bob"} ...
```

```
USE lec16;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;

Use lec16;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;
```

Discussion of NFNF

- 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
Example from HW5

```json
mondiad.adm is totally semi-structured:
{"mondiad": ["country": [...], "continent": [...], ... "desert": [...] ]}
```

<table>
<thead>
<tr>
<th>country</th>
<th>continent</th>
<th>organization</th>
<th>sea</th>
<th>mountain</th>
<th>desert</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Albania&quot;</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&quot;Greece&quot;</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

country.adm, sea.adm, mountain.adm are more structured

```sql
COUNTRY:
- car_code
- name
- ethnicgroups
- religions
- city
AL Albania | | | | |
GR Greece | | | | |

Indexes

- Can declare an index on an attribute of a top-most collection
- Available:
  - BTREE: good for equality and range queries
    E.g. name="Greece"; 20 < age and age < 40
  - RTREE: good for 2-dimensional range queries
    E.g. 20 < x and x < 40 and 10 < y and y < 50
  - KEYWORD: good for substring search

```sql
USE lec16;
CREATE INDEX countryID ON country(name) TYPE BTREE;
```

Indexes

- Cannot index inside a nested collection

```sql
USE lec16;
CREATE INDEX countryID ON country(name) TYPE BTREE;
```

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - \{"Name": "Alice", "age": 40\}
  - Fields must be distinct:
    \{"Name": "Alice", "age": 40, "age": 50\}
- Arrays:
  - \[1, 3, "Fred", 2, 9\]
  - Arrays can be heterogeneous
- Bags:
  - \{"1, 3, "Fred", "Fred", 2, 9\}

Examples

Try these queries yourself:

```sql
SELECT age FROM \{"name": "Alice", "age": ["30", "50"]\} x;
-> \{"age": ["30", "50"]\}
```

```sql
SELECT age FROM \{\{"name": "Alice", "age": ["30", "50"]\}\} x;
-> \{"age": ["30", "50"]\}
```

SQL++ Overview

```
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```
Nested Collections

- If the value of attribute B is some other collection, then we can simply iterate over it.

```
SELECT x.A, y.C, y.D
FROM mydata x, y.B y;
```

Heterogeneous Collections

The problem:

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name='Greece';
```
The problem:

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z
WHERE y.name='Greece' and is_array(z.city);
```

Useful Functions

- `is_array`
- `is_boolean`
- `is_number`
- `is_object`
- `is_string`
- `is_null`
- `is_missing`
- `is_unknown = is_null or is_missing`

Useful Paradigms

- Unnesting
- Nesting
- Group-by / aggregate
- Join
- Multi-value join
Basic Unnesting

- An array: [a, b, c]
- A nested array: arr = [[a, b], [c, d]]
- Unnest(arr) = [a, b, c, d]

Unnesting Specific Field

```
UnnestF(coll) = [{A:a1, B:b1, G:{C:c1}},
{A:a1, B:b2, G:{C:c1}},
{A:a2, B:b3, G:{}},
{A:a2, B:b4, G:{}},
{A:a2, B:b5, G:{}},
{A:a3, B:b6, G:{C:c2, C:c3}}]
```

```
UnnestG(coll) = [{A:a1, F:{B:b1, B:b2}, C:c1},
{A:a3, F:{B:b6}, C:c2},
{A:a3, F:{B:b6}, C:c3}]
```

Select

```
SELECT DISTINCT x.A, y.B, x.G
FROM coll x, x.F y
```

```
SELECT x.A, x.F, z.C
FROM coll x, x.G z
```

Nesting (like group-by)

A flat collection

```
coll = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
```

```
NestA(coll) = [{A:a1, GRP:{B:b1, B:b2}},
{A:a2, GRP:{B:b2}}]
```

```
NestB(coll) = [{B:b1, GRP:{A:a1, A:a2}},
{B:b2, GRP:{A:a1}}]
```

```
SELECT DISTINCT x.A, coll_count(x.F) as cnt
FROM coll x
```

```
SELECT x.A, count(*) as cnt
FROM coll x, x.F y
GROUP BY x.A
```

```
GROUP BY x.A
```

```
SELECT DISTINCT x.A, coll_count(g) as cnt
FROM coll x
LET g = (SELECT y.B FROM coll y WHERE x.A = y.A)
```

```
SELECT x.A, count(*) as cnt
FROM coll x, x.F y
GROUP BY x.A
```

```
GROUP BY x.A
```

Join

Two flat collection

```
coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
coll2 = [{B:b1, C:c1}, {B:b3, C:c3}]
```

```
SELECT x.A, x.B, y.C
FROM coll1 x, coll2 y
WHERE x.B = y.B
```
Multi-Value Join

- A many-to-one relationship should have one foreign key, from "many" to "one"
  - each of the "many" points to the same "one"
- Sometimes, people represent it in the opposite direction, from "one" to "many":
  - Ex: list of employees of a manager
  - reference could be space-separated string of keys
  - need to use split(string, separator) to split it into a collection of foreign keys

Behind the Scenes

Query Processing on NFNF data:
- Option 1: give up on query plans
- Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan

You can apply the second approach yourself, to work with semi-structured data using a familiar RDBMS
- for data analysis, this may be more efficient until semi-structured DBMSs catch up to RDBMSs