Database Systems
CSE 414

Lecture 12-13: JSON and SQL++
(mostly not in textbook)
NoSQL (cont)
JSON (cont.)
JSON Semantics: a Tree!

```json
{
    "person": [
        {
            "name": "Mary",
            "address": {
                "street": "Maple",
                "no": 345,
                "city": "Seattle"
            }
        },
        {
            "name": "John",
            "address": "Thailand",
            "phone": 2345678
        }
    ]
}
```
**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

<table>
<thead>
<tr>
<th>Person</th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
<td></td>
</tr>
</tbody>
</table>

```json
{
    "person": [
        {
            "name": "John",
            "phone": 3634
        },
        {
            "name": "Sue",
            "phone": 6343
        },
        {
            "name": "Dirk",
            "phone": 6383
        }
    ]
}
```
Mapping Relational Data to JSON

May inline foreign keys

Person

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
</tbody>
</table>

Orders

<table>
<thead>
<tr>
<th>personName</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2002</td>
<td>Gizmo</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
<td>Gadget</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
<td>Gadget</td>
</tr>
</tbody>
</table>

```
{  
  "Person": [  
    {  
      "name": "John",  
      "phone": 3646,  
      "Orders": [  
        {  
          "date": 2002,  
          "product": "Gizmo"  
        },  
        {  
          "date": 2004,  
          "product": "Gadget"  
        }  
      ]  
    },  
    {  
      "name": "Sue",  
      "phone": 6343,  
      "Orders": [  
        {  
          "date": 2002,  
          "product": "Gadget"  
        }  
      ]  
    }  
  ]  
}
```
JSON = Semi-structured Data (1/3)

- Missing attributes:

```
{“person”: 
[ {“name”: “John”, “phone”: 1234}, 
{“name”: “Joe”} ]
}
```

- 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
{"person":
    [{"name": "John", "phone": 1234},
     {"name": "Mary", "phone": [1234, 5678]}]
}
```

• Impossible in one table:

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

Two phones!
JSON = **Semi-structured Data** (3/3)

- Attributes with different types in different objects
  
  ```json
  {"person":
   [{"name": "Sue", "phone": 3456},
    {"name": {"first": "John", "last": "Smith"}, "phone": 2345}
   ]
  }
  ```

- 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
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}}
Try these queries yourself:

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

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

```sql
-- error
SELECT age FROM { 'name': 'Alice', 'age': ['30', '50'] } x;
```
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": "123456789"}
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)
Dataset with Existing Key

USE lec16;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    Name: string,
    email: string?
}

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

{“Name”: “Alice”}
{“Name”: “Bob”}
…
USE lec16;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    myKey: uuid,
    Name: string,
    email: string?
}

Note: no myKey since it will be auto-generated

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

{“Name”: “Alice”}
{“Name”: “Bob”}
...
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

mondial.adm is totally semi-structured:
{"mondial": {“country”: [...], “continent”: [...], …, “desert”: [...]}}

<table>
<thead>
<tr>
<th>country</th>
<th>continent</th>
<th>organization</th>
<th>sea</th>
<th>…</th>
<th>mountain</th>
<th>desert</th>
</tr>
</thead>
<tbody>
<tr>
<td>[{“name”: “Albania”, …}, {“name”: “Greece”, …}, …]</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

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

Country:

<table>
<thead>
<tr>
<th>-car_code</th>
<th>name</th>
<th>…</th>
<th>ethnicgroups</th>
<th>religions</th>
<th>…</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>…</td>
<td>[ … ]</td>
<td>[ … ]</td>
<td>…</td>
<td>[ … ]</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
<td>…</td>
<td>[ … ]</td>
<td>[ … ]</td>
<td>…</td>
<td>[ … ]</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Indexes

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

Cannot index inside a nested collection

USE lec16;
CREATE INDEX cityname
ON country(city.name)
TYPE BTREE;

Country:

<table>
<thead>
<tr>
<th>-car_code</th>
<th>name</th>
<th>ethnicgroups</th>
<th>religions</th>
<th>...</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>...</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>...</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BG</td>
<td>Belgium</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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;
``` ~> {"age": ["30", "50"]}

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

```
-- error
SELECT age FROM { 'name': 'Alice', 'age': ['30', '50'] } x;
```
SQL++ Overview

SELECT ... FROM ... WHERE ... [GROUP BY ...]
Retrieve Everything

```
SELECT x.mondial FROM world x;
```

Answer

```
{“mondial”:
  {“country”: [ country1, country2, …],
   “continent”: […],
   “organization”: […],
   …
  …
}
```
Retrieve countries

```
{{"mondial":
  {{"country": [ country1, country2, ...],
    "continent": [...],
    "organization": [...],
    ...
  }},
  ...
}}
```

Answer

```
{{"country": [ country1, country2, ...]}}
```

```
SELECT x.mondial.country FROM world x;
```
Retrieve countries, one by one

```
SELECT y as country FROM world x, x.mondial.country y;
```

Answer

```
{{"country": country1}
{{"country": country2}
...
```
Escape characters

SELECT y.-car_code` as code, y.name as name
FROM world x, x.mondial.country y order by y.name;

Answer

{"code": "AFG", "name": "Afganistan"} 
{"code": "AL", "name": "Albania"} 
...

"-car_code" illegal field
Use ` ... `
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, x.B y;
```
Heterogeneous Collections

The problem:

```sql
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';
```

```json
[{
  "mondial": {
    "country": [ country1, country2, ... ],
    "continent": [ ... ],
    "organization": [ ... ],
    ...
  }
}]
```

- city is an array
- city is an object
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' and is_array(z.city);

The problem:

...“province”: [ ...
  {“name”: ”Attiki”,
   “city” : [ {“name”: “Athens”...}, {“name”: “Pireus”...}, ..]
  ...},
  {“name”: ”Ipiros”,
   “city” : {“name”: “Ioannia”...}
  ...],
Heterogeneous Collections

SELECT z.name as province_name, z.city.name as city_name
FROM world x, x.mondial.country y, y.province z
WHERE y.name='Greece' and not is_array(z.city);

The problem:

... provincial: [ ...
  provincial: "Attiki",
  city: [ { provincial: "Athens"...}, { provincial: "Pireus"...}, ..]
...
  provincial: "Ipiros",
  city: { provincial: "Ioannia"...
...
]
Heterogeneous Collections

SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
    (CASE WHEN is_array(z.city) THEN z.city
         ELSE [z.city] END) u
WHERE y.name='Greece';

The problem:

...  
"province": [ ...
    {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..] ...
    },
    {"name": "Ipiros",
    "city" : {"name": "Ioannia"...
    ...
    },
    "province": [ country1, country2, ...],
    "continent": [...],
    "organization": [...],
    ...  
]
The problem:

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
(CASE WHEN z.city is missing THEN []
    WHEN is_array(z.city) THEN z.city
    ELSE [z.city] END) u
WHERE y.name='Greece';
```

Even better
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], [], [b, c, d]]
• Unnest(arr) = [a, b, b, c, d]

SELECT y
FROM arr x, x y
Unnesting Specific Field

**A nested collection**

\[
\text{coll} = \\
\{\{A:a1, F:\{B:b1, \{B:b2\}\}, \text{G}:\{\{C:c1\}\}\}, \\
\{A:a2, F:\{B:b3, \{B:b4, \{B:b5\}\}, \text{G}:\[]\}, \\
\{A:a3, F:\{\}, \text{G}:\{\{C:c2, \{C:c3\}\}\}\}
\]

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

\[
\text{Unnest}_G(\text{coll}) = \\
\{\{A:a1, F:\{B:b1, \{B:b2\}\}, \text{C}:c1\}, \\
\{A:a3, F:\{\}, \text{C}:c2\}, \\
\{A:a3, F:\{\}, \text{C}:c3\}\}
\]

**SQL++**

\[
\text{SELECT x.A, y.B, x.G} \\
\text{FROM coll x, x.F y}
\]

\[
\text{SELECT x.A, x.F, z.C} \\
\text{FROM coll x, x.G z}
\]
Nesting (like group-by)

A flat collection

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

\[
\text{Nest}_A(\text{coll}) = \{\{A:a1, \text{GRP}:\{\{B:b1\}, \{B:b2\}\}\}, \\
\{A:a2, \text{GRP}:\{\{B:b2\}\}\}\}
\]

\[
\text{Nest}_B(\text{coll}) = \{\{B:b1, \text{GRP}:\{\{A:a1\}, \{A:a2\}\}\}, \\
\{B:b2, \text{GRP}:\{\{A:a1\}\}\}\}
\]

\[
\begin{align*}
\text{SELECT} & \text{ DISTINCT } x.A, \\
& (\text{SELECT} y.B \text{ FROM coll y WHERE } x.A = y.A) \text{ as GRP} \\
\text{FROM} & \text{ coll x}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \text{ DISTINCT } x.A, g \text{ as GRP} \\
\text{FROM} & \text{ coll x} \\
\text{LET} & g = (\text{SELECT} y.B \text{ FROM coll y WHERE } x.A = y.A)
\end{align*}
\]
Group-by / Aggregate

A nested collection

```plaintext
coll =
[A:a1, F:{B:b1, {B:b2}}, G:{C:c1}>,</A:a2, F:{B:b3, {B:b4}, {B:b5}}, G:[ ]},
{A:a3, F:{B:b6}, G:{C:c2, {C:c3}}}]]
```

Count the number of elements in the F collection
coll_count = collection count

SELECT 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

These are NOT equivalent!
(Why?)
Group-by / Aggregate

A flat collection

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

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
GROUP BY x.A

Are these equivalent?
Join

Two flat collection

coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
coll2 = [{B:b1, C:c1}, {B:b1, C:c2}, {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
Multi-Value Join

river =
["name": "Danube", "-country": "SRB A D H HR SK BG RO MD UA"],
{"name": "Colorado", "-country": "MEX USA"},
...

SELECT ...
FROM country x, river y,
  split(y.`-country`, " ") z
WHERE x.`-car_code` = z

split("MEX USA", " ") =
["MEX", "USA"]
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
Flattening SQL++ Queries

A nested collection

coll =
{{A:a1, F:[{B:b1}, {B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3}, {B:b4}, {B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2}, {C:c3}]}}

Flat Representation

coll:

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

SQL++

\[
\text{SELECT } x.A, y.B \\
\text{FROM } \text{coll } x, x.F \ y \\
\text{WHERE } x.A = 'a1'
\]

SQL

\[
\text{SELECT } x.A, y.B \\
\text{FROM } \text{coll } x, F \ y \\
\text{WHERE } x.id = y.parent \text{ and } x.A = 'a1'
\]