Database Systems
CSE 414

Lecture 16: JSON and SQL++
(mostly not in textbook)

Announcement

- Midterm graded
  - to be handed back in section
  - answers were okay on problems 1, 3, 4
  - problem 2 caused a lot of difficulty
    - expect a final question on cost based optimization
- HW5 will be posted tonight: due May 9
  - uses AsterixDB
- Section on installing & using AsterixDB
  - get through problem 1 tomorrow

NoSQL (cont)

Google Cloud Datastore demo

JSON (cont.)

JSON Semantics: a Tree!

{"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 = semistructured 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>

May inline foreign keys

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

JSON = Semi-structured Data (1/3)

- Missing attributes:

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

- Could represent in a table with nulls:

```
name phone
John 1234
Joe  -
```

JSON = Semi-structured Data (2/3)

- Repeated attributes

```
{"person": [{"name": "John", "phone": 1234},
             {"name": "Mary", "phone": [2345, 3456]}]
```

- Impossible in one table:

```
name phone
Mary 2345 3456
```

JSON = Semi-structured Data (3/3)

- Attributes with different types in different objects

```
{"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 SS 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 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:

```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;
```

Datatypes

- 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

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

```sql
{ "b": 2 }
{ }
```

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

```sql
{ "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 autoincremented unique integer IDs)

Dataset with Existing Key

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

{"Name": "Alice"}
{"Name": "Bob"}
**Dataset with Auto Generated Key**

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

```
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

**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

**Example from HW5**

`mondial.adm` is totally semistructured:

```
{ "mondial": { "country": [...], "continent": [...], ..., "desert": [...]} }
```

**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;
```

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

-- 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

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

Answer

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

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"
```
Nested Collections

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

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

Heterogeneous Collections

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:

```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' and is_array(z.city);
```

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' and not is_array(z.city);
```

Heterogeneous Collections

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

The problem:

```sql
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

```sql
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 = 'Greece' THEN [z.city] ELSE [] END) u
WHERE y.name='Greece';
```
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]

Unnesting Specific Field

A nested collection

\[ \text{coll} = \{ \{ A : a1, B : b1 \}, \{ A : a1, B : b2 \}, \{ A : a2, B : b3 \} \} \]

Unnest\(\text{coll}\) = \{ A : a1, B : b1 \} \{ A : a1, B : b2 \} \{ A : a2, B : b3 \}

SELECT x.A, x.B, y FROM coll x, x.F y

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

These are NOT equivalent!

Why?

Nesting (like group-by)

A flat collection

\[ \text{col} = \{ [A : a1, B : b1], [A : a1, B : b2], [A : a2, B : b3] \} \]

Nest\(\text{col}\) = \{ [A : a1, GRP - [B : b1, B : b2]], [A : a2, GRP - [B : b3]] \}

SELECT DISTINCT x.A, y.B FROM col y WHERE x.A = y.A as GRP FROM col x

SELECT DISTINCT x.A, g as GRP FROM col x

LET g = (SELECT y.B FROM col y WHERE x.A = y.A)

Group-by / Aggregate

A nested collection

\[ \text{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

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
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
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

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

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 SS data using a familiar RDBMS

- for data analysis, this may be more efficient until SS DBMSs catch up to RDBMSs

Flattening SQL++ Queries

A nested collection

```
```

SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.A = 'a1'

SELECT x.A, y.B
FROM coll x, F y, G z
WHERE y.B = z.C

SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.id = y.parent and x.A = 'a1'

SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.id = y.parent and x.id = z.parent and y.B = z.C