Introduction to Data Management
CSE 344

Lecture 16: JSon and N1QL

Announcements

- HW5 released
  - Due on 11/11
  - No WQ5
- Midterm on Monday

JSon

An Example Semi-structured Data Format

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 vs Relational

- Relational data model
  - Rigid flat structure (tables)
  - Schema must be fixed in advanced
  - Binary representation: good for performance, bad for exchange
  - Query language based on Relational Calculus

- Semi-structured data model / JSon
  - Flexible, nested structure (trees)
  - Does not require predefined schema ("self describing”)
  - Text representation: good for exchange, bad for performance
  - Most common use: Language API; query languages emerging

JSon Syntax

```
{  "book": [
    {"id":"01",
       "language": "Java”,
       "author": ”H. Javeson”,
        “year”: 2015
    },
    {"id":"07",
       "language": "C++",
       "edition": "second"
       "author": ”E. Sepp”,
       “price”: 22.25
    }
  ]
}
```

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 is followed by ':'(colon) followed by the value
- Square brackets hold arrays and values are separated by ,(comma).

JSon Data Structures

- Collections of name-value pairs:
  - {“name1”: value1, “name2”: value2, …}
  - The “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

```
{"id":"07",
   "title": "Databases",
   "author": "Garcia-Molina",
   "author": "Ullman",
   "author": "Widom"
}
```

⇒

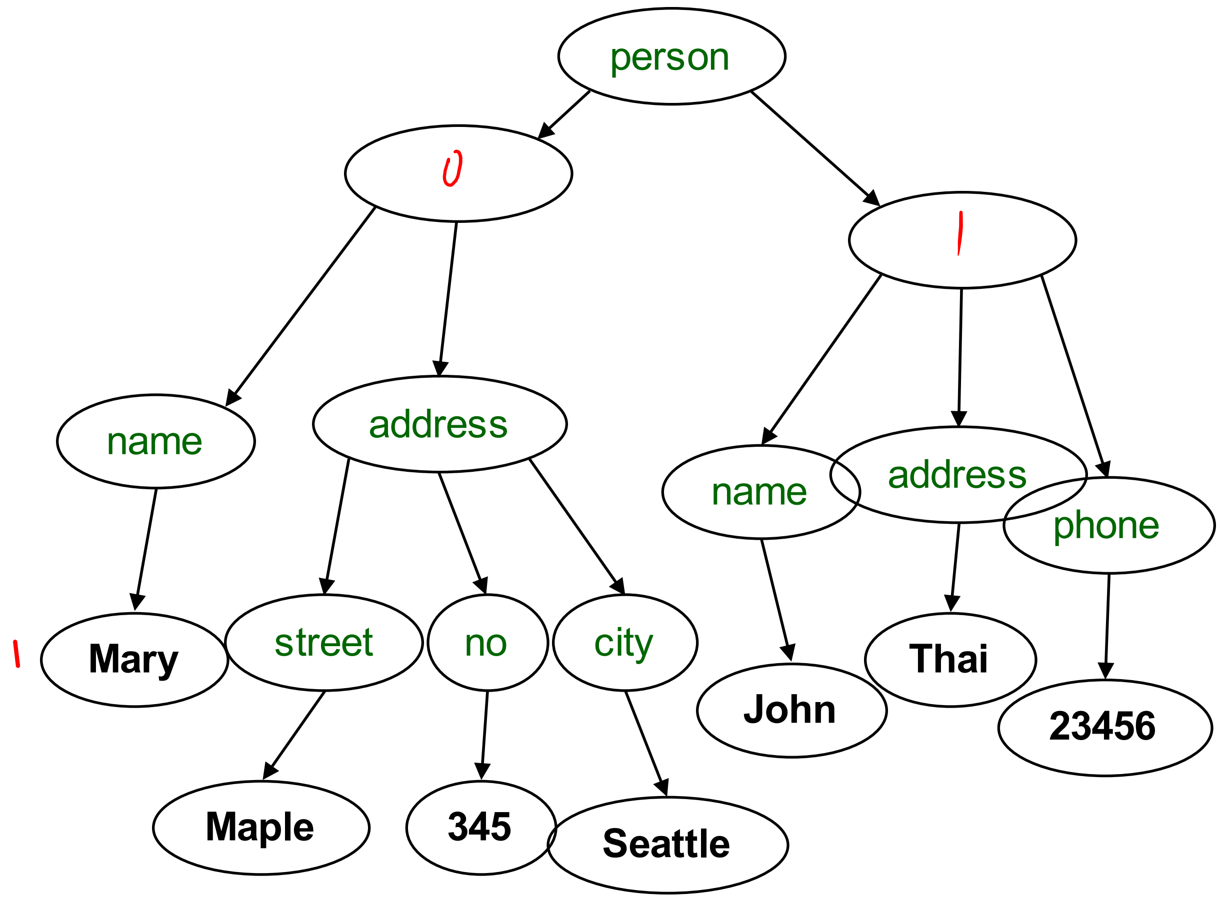
```
{"id":"07",
   "title": "Databases",
   "author": ["Garcia-Molina",
              "Ullman",
              "Widom"]
}
```

JSon Datatypes

- Number
- String = double-quoted
- Boolean = true or false
- nullempty

JSon Semantics: a Tree !

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

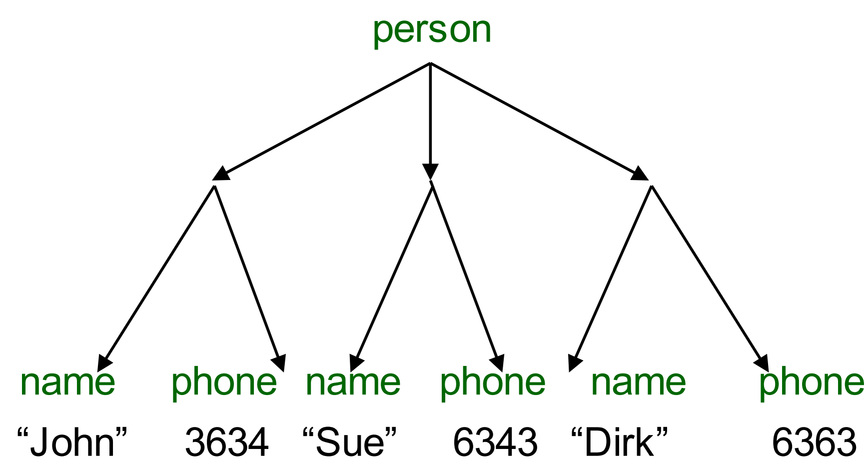


JSon Data

- 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

- Hence JSon is semi-structured data

Mapping Relational Data to JSon



Person

name	phone
John	3634
Sue	6343
Dirk	6363

```
{"person":
     [{"name": "John", "phone":3634},
       {"name": "Sue",  "phone":6343},
       {"name": "Dirk",  "phone":6383}
     ]
}
```

Mapping Relational Data to JSon

May inline foreign keys

Person

name	phone
John	3634
Sue	6343

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

```
{"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

- Missing attributes:

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

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	-

JSon Semi-structured Data

- Repeated attributes

```
{“person”:
   [{“name”:”John”, “phone”:1234},
    {“name”:”Mary”, “phone”:[1234,5678]}]
}
```

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456

???

JSon Semi-structured Data

- Attributes with different types in different objects

```
{"person":
   [{"name":"Sue", "phone":3456},
    {"name":{"first":"John","last":"Smith"},"phone":2345}
   ]
}
```

Structured name !

- Nested collections
- Heterogeneous collections

Query Language For JSon: N1QL

- Pronounced as “nickel”
- Used in CouchDB only for retrieving documents
  - Recall CouchDB is a JSon (i.e., document) store
- SQL-ish notation with extensions:
  - Nested collections
  - Dependent joins

N1QL Overview

Also called a "key space"

```
SELECT … FROM bucket ... WHERE
```

- Each bucket defines a set of key:value pairs
- By default N1QL flattens all documents in the same bucket into the same key space

Buckets in CouchDB

- A bucket = a database
  - Contains a set of documents
  - Document ID must be unique
- A CouchDB server can hold many buckets
  - By default it allocates all main memory to the first bucket, so to create more buckets we need to either resize / delete previous ones

Loading Data Example

Load into bucket “foo”

t1.json

{“hello”: “world”}

t2.json

{“foo”: “bar”}

```
cbdocloader <…> -b foo t1.json;
cbdocloader <…> -b foo t2.json;
```

Bucket name doesn’t need to be the same as file name

t1	{"hello":"world"}
t2	{"foo":"bar"}

HW 5: mondial.json

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

Defines a key "mondial" that maps to a big object that contains attributes country, continent, organization, etc

Each country object has its own attributes

In relational data model, this is like several tables:

Country

code	capital	…

Continent

id	name	…

…

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

Basic Retrieval

Bucket name

Load data

```
cbdocloader <…> -b mondial mondial.json;
```

Document contents

```
SELECT x FROM mondial x;
```

Answer: some metadata +

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

Basic Retrieval

t1.json

{“hello”: “world”}

t2.json

{“foo”: “bar”}

```
SELECT x.* from foo x;
```

```
"results": [
  { "hello": "world" },
  { "foo": "bar" }
]
```

- All keys are flattened into same key space
- `USE KEYS` to select from a particular document:
  - `SELECT x.* FROM foo x USE KEYS ["t1"];`
  - `"results": [ {"hello": "world"} ]`

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

Nested data

A path

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

Answer: [*country1, country2, …*]

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

Nested data

Path using array syntax

```
SELECT x.mondial.country[0] FROM mondial x;
```

Answer: [*country1*]

Supported Datatypes

- Boolean
- Numbers
- Strings
- Arrays
  - Mixing types is fine, e.g.,: [“one”, “two”], [1, “two”]
- NULL
- MISSING
  - Returned when asking for a non-existent field in a document
- Date
- Binary data

```
{"contacts": {
  {"name": "Peter",
  {"children": [ {"name": "John”, "age": 21} ]}
  }, ... }
```

Operators and Functions

- Standard comparisons (=, <, >, etc)
- Collection operators
  - ANY

```
SELECT x.contacts.name FROM contacts x
  WHERE ANY child in children
  SATISFIES child.age > 18 END
```

  - `EVERY`, `EXISTS`, `IN`, etc
- Aggregate functions
  - `AVG(e)`, `COUNT(*)`, `MAX(e)`, `MIN(e)`, etc

UNNEST

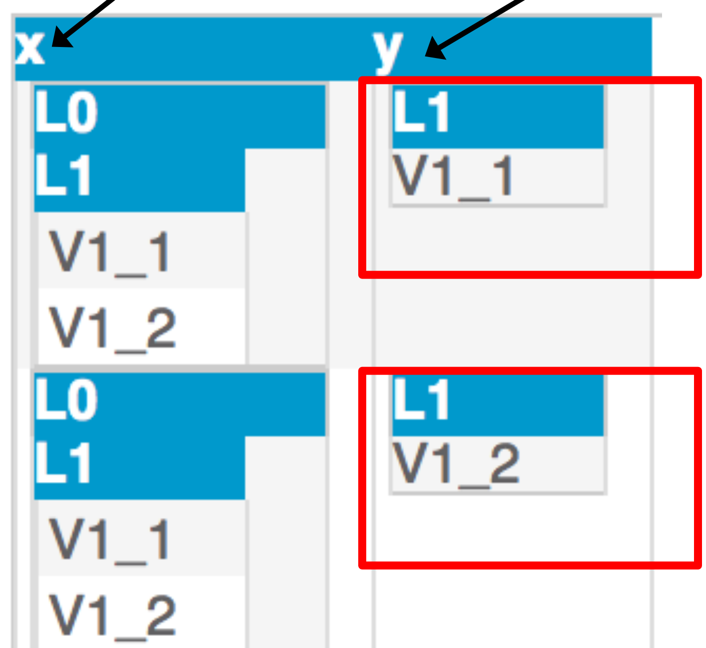
- Performs a join of the nested array with its parent object

```
{"L0": [
  { "L1": "V1_1" },
  { "L1": "V1_2" }
  ] }
```

```
/* after loading into bucket foo.. */
SELECT * FROM foo x UNNEST x.L0 y
```

Answer:

```
{"y": { "L1": "V1_1" },
 "x": { "L0": [
          { "L1": "V1_1" },
          { "L1": "V1_2" } ] } },
{"y": { "L1": "V1_2" },
 "x": { "L0": [
          { "L1": "V1_1" },
          { "L1": "V1_2" } ] } }
```



2 objs are returned, where each “L0” obj (called x) is paired up with each of the nested “L1” objects (called y).

UNNEST

- Performs a join of the nested array with its parent object

```
{"L0": [
  { "L1": "V1_1" },
  { "L1": "V1_2" }
  ] }
```

```
/* after loading into bucket foo.. */
SELECT y FROM foo x UNNEST x.L0 y
```

Answer:

```
{ "y": { "L1": "V1_1" } },
{ "y": { "L1": "V1_2" } }
```

Same as before but only the y objects are projected.

UNNEST: More Complex Example

- Performs a join of the nested array with its parent object

```
{"customers":
 [ {"name": Mary,
    "address": [ {"street": "addr1", "zip": 98100},
                 {"street": "addr2", "zip": 98101} ] },
  {"name": Derek
   "address": [ {"street": "addr3", "zip": 98200},
                {"street": "addr4", "zip": 98201} ] },
    … ] }
```

```
SELECT c.name, a.* FROM bucket c.customers UNNEST c.address a
```

Answer:

```
[ {"name": "Mary", "street": "addr1", "zip": 98100 },
  {"name": "Mary", "street": "addr2", "zip": 98101},
  {"name": "Derek", "street": "addr3", "zip": 98200},
  {"name": "Derek", "street": "addr3", "zip": 98201}, …]
```

- UNNESTs can be chained

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

Retrieve Countries (2)

```
SELECT z
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z;
```

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

Retrieve Countries (2)

```
SELECT z
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z;
```

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

Retrieve Countries (2)

```
SELECT z
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z;
```

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

Retrieve Countries (2)

```
SELECT z
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z;
```

Answer: [*country1, country2, …*]

- By default, both sides of `UNNEST` needs to be non-missing and non-null
  - Otherwise no result is produced
- Variants: `LEFT OUTER UNNEST`
  - What do you think it means?

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

Retrieve Names

```
SELECT z.name
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z;
```

Answer:

```
[ {"name": "Albania”},
  {"name": "Greece”},
  {"name": "Macedonia”},
  {"name": "Serbia”},
  ...
]
```

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

Accessing Arrays

```
SELECT z
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z
LIMIT 1 OFFSET N - 1
```

Retrieve the N'th country

Answer: *countryN*

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

WHERE Clause

```
SELECT z
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z
WHERE z.name = “Greece”
```

Answer:

```
{"name": "Greece"
 "population": ...
...
}
```

Aside: Non-standard Names

- Normally, a JSon name like "population” is referenced like this:

```
x.population
```

- Mondial.json has some non-standard names: “-car_code”, "-area”, "-capital”
- Reference them like this:

```
x.["-car_code"]
```

Joins

- Pretty much like in SQL, but we need to unnest to get to the right collection(s) to join

All Rivers in France

key

```
{"mondial":
    {"country": [  …  {"-car_code": "F", "name":"France", ...}    ...]
    ...
    {"river": [ ... {""-id": "river-Loire", "-country": "F", "name":  "Loire",....} ... ]
}
```

Foreign Key

```
SELECT u.name
FROM mondial x
UNNEST x.mondial y
UNNEST y.country z
UNNEST y.river u
WHERE z.name = "France" and z.["-car_code"]=u.["-country"]
```

Answer:

```
[{"name":"Loire"},{"name":"Saone"},{"name":"Isere"},
   {"name":"Seine"},{"name":"Marne"}]
```

Other Constructs of Interest

- `Group by, order by`
  - Just like SQL
- `ARRAY_LENGTH(collection)`
  - Just like `count(*)`
- `TONUMBER(field)`
  - Converts from string to number
- More online (see HW5)

Final Thoughts

- JSon: a semi-structructed data model
  – What is new: nested collections
- Many parallels between N1QL and SQL constructs
- Querying non-relational data is painful
  – E.g. find *all* rivers that pass through France, not just those located entirely in France.