Introduction to Data Management CSE 344

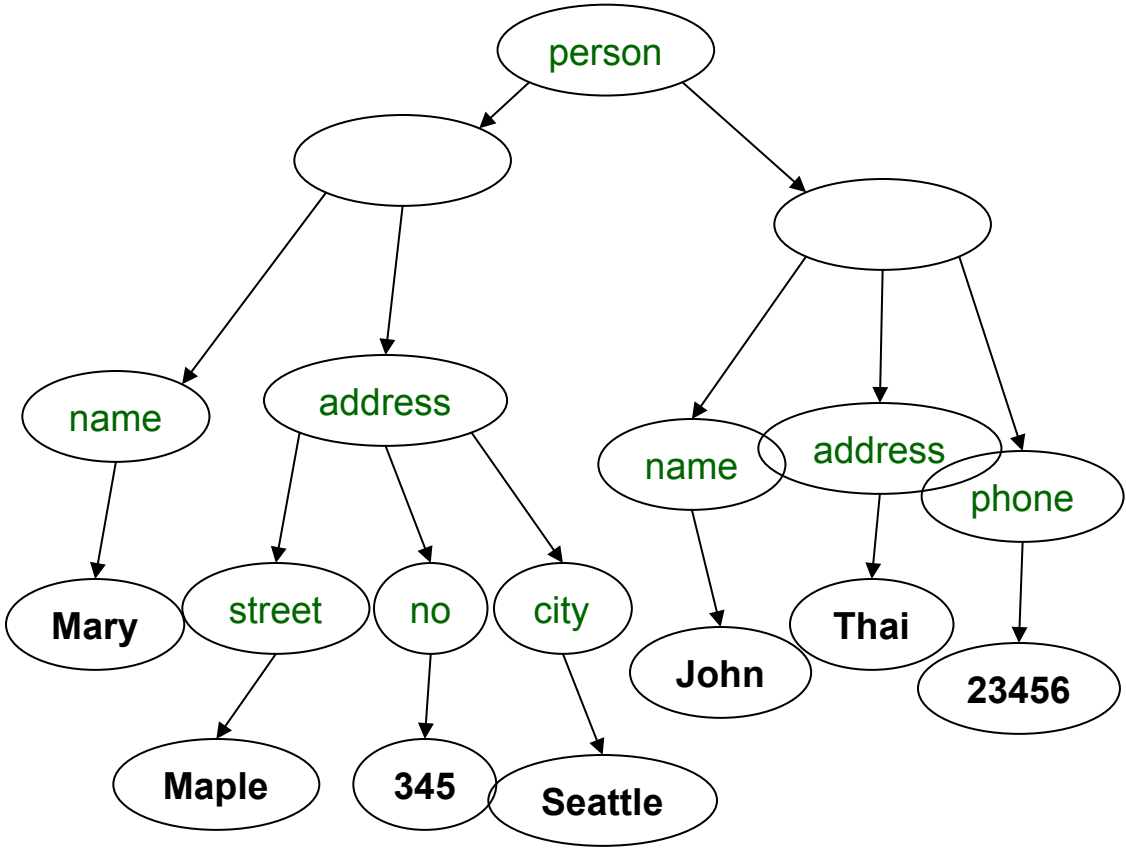
Lecture 16: JSon and N1QL

Announcements

- No lecture on Monday (President's Day)
- HW5 due on Friday night

JSon Semantics: a Tree !

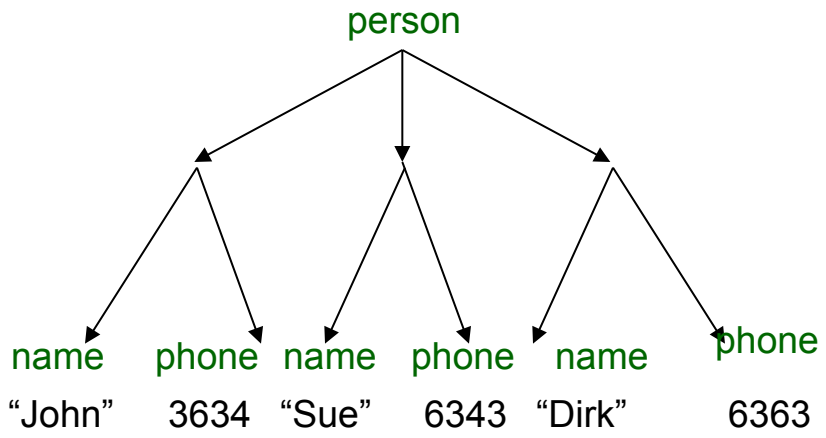
```
{“person”:
   [ {“name”: “Mary”,
      “address”:
         {“street”:“Maple”,
          “no”:345,
          “city”: “Seattle”}},
     {“name”: “John”,
      “address”: “Thailand”,
      “phone”:2345678}}
     ]
}
```



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



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 (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”:[1234,5678]}]
}
```

Two phones !

- Impossible in one table:

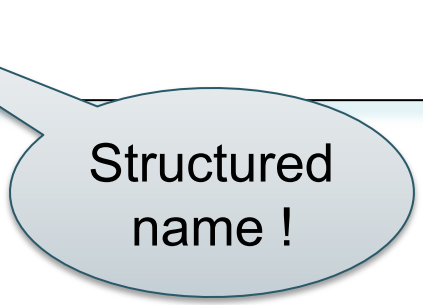
name	phone	
Mary	2345	3456

???

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*
  - Ideally 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 textbook)
  - Supported inside many relational engines (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 (we'll struggle with it in HW5), may be replaced by AQL (better designed)
  - MongoDB: has a pattern-based language
  - JSONiq http://www.jsoniq.org/

N1QL

- Used in CouchDB only
- SQL-ish notation with extensions:
  - Nested collections
  - Dependent joins

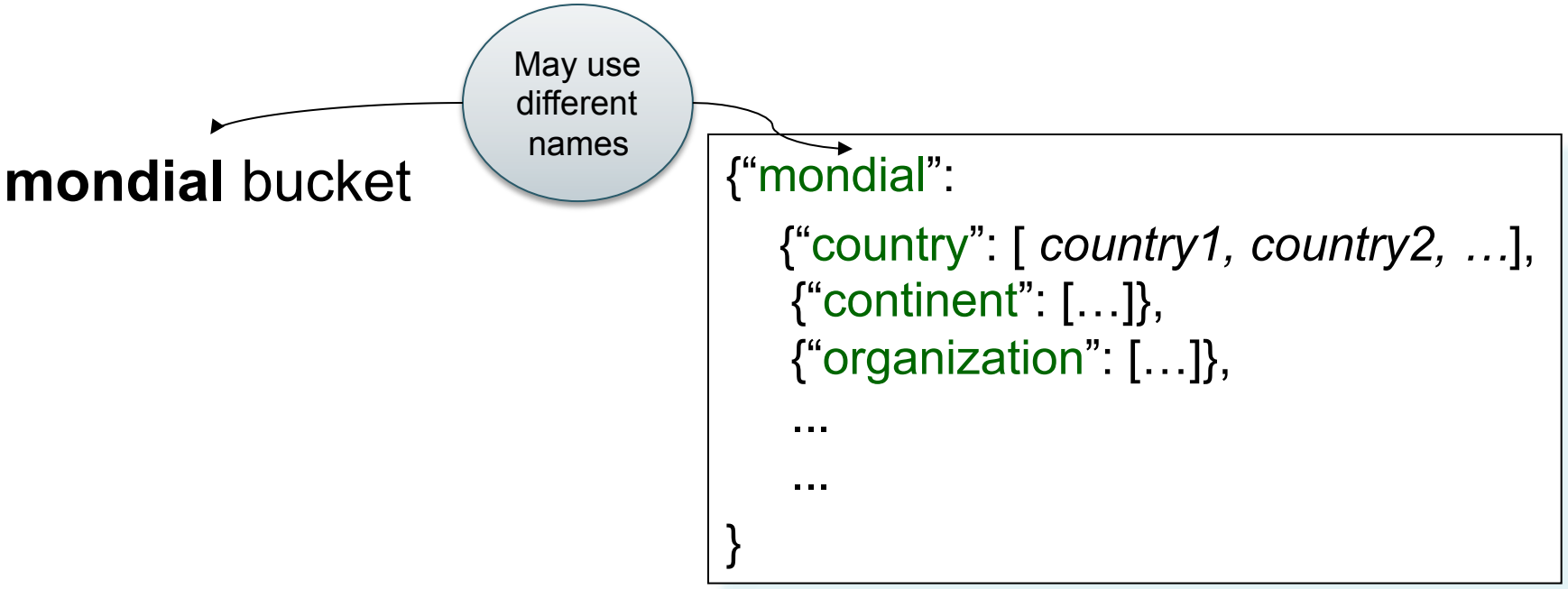
N1QL Overview

SELECT … FROM bucket ... WHERE

Buckets in CouchDB

- A bucket = a database
- A CouchDB server can hold several buckets
- Buckets stored in main memory:
  - To load a new bucket need to make room by deleting an old one

Mondial.json



This is like several tables:

Country

code	capital	…

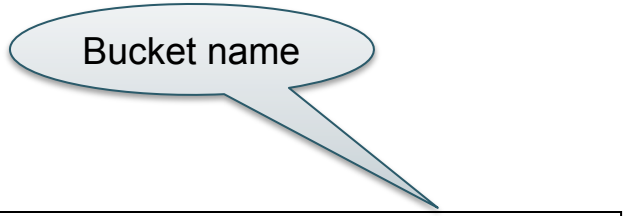
Continent

id	name	…

…

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

Retrieve Everything



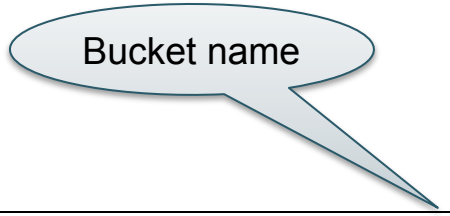

```
SELECT x FROM mondial x;
```

Answer: some metadata +

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

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

Retrieve Countries




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

Answer: some metadata + [*country1, country2, …*]

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

Retrieve Countries (2)

```
select z from mondial x unnest x.mondial y unnest y.country z;
```

Answer: some metadata + [*country1, country2, …*]

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

Retrieve Countries (2)

```
select z from mondial x unnest x.mondial y unnest y.country z;
```

Answer: some metadata + [*country1, country2, …*]

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

Retrieve Countries (2)

```
select z from mondial x unnest x.mondial y unnest y.country z;
```

Answer: some metadata + [*country1, country2, …*]

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

Retrieve Countries (2)

```
select z from mondial x unnest x.mondial y unnest y.country z;
```

Answer: some metadata + [*country1, country2, …*]

Unnesting A Collection

- Unnest({{a,b,c}, {d,b},{g,a,f}})
  = {a,b,c,d,b,g,a,f}

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

Retrieve Names

```
select z.name
from mondial x
     unnest x.mondial y
     unnest y.country z;
```

Answer: some metadata +

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

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

Accessing Arrays

Retrieve the 3'rd country

```
select z[3] from mondial x unnest x.mondial y unnest y.country z;
```

Answer: some metadata + *country3*

```
{“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: some metadata + *country2*

Non-standard Names

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

  x.population

- Mondial.json has some nostandard 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

- Group by, order by = as usual
- ARRAY_LENGTH(collection) = an alternative to count(*)
- TONUMBER(field) = converts from string to number
- You can find more online (see HW5)

Final Thoughts

- Unclear what will become of N1QL in the futurer; however, its treatement of nested collections is similar to other languages: XQuery, Dremel, AQL, …

- Querying non-relational data is painful
  - E.g. find *all* rivers that pass through France, not just those located entirely in France.