Introduction to Database Systems
CSE 414

Lecture 13: Json and SQL++

Announcements

- HW5 + WQ5 will be out tomorrow
  - Both due in 1 week
- Midterm in class on Friday, 5/4
  - Covers everything (HW, WQ, lectures, sections, readings) up to and including next Monday's lecture and HW5 + WQ5
  - Review session: 5/2 in MUE 153, 5-7pm
- Make sure you are good for AWS
  - You will need it for HW6

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 Data Structures

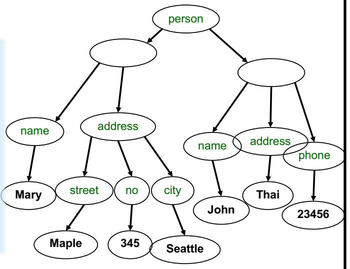
- Objects, i.e., collections of name-value pairs:
  - `{“name1”: value1, “name2”: value2, …}`
  - “`name`” is also called a “key”
- *Ordered* lists of values:
  - `[obj1, obj2, obj3, ...]`

JSon Primitive Datatypes

- Number
- String
  - Denoted by double quotes
- Boolean
  - Either `true` or `false`
- nullempty

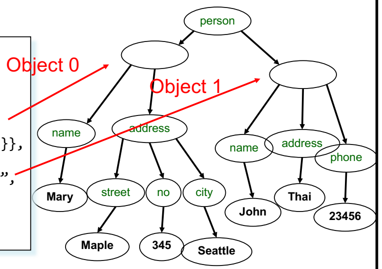
JSon Semantics: a Tree !

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



JSon Semantics: a Tree !

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

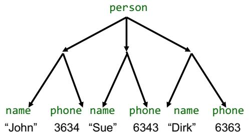


Recall: arrays are *ordered* in Json!

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 multiple relations based on foreign keys

Person

name	phone
John	3634
Sue	6343

FK

Orders

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

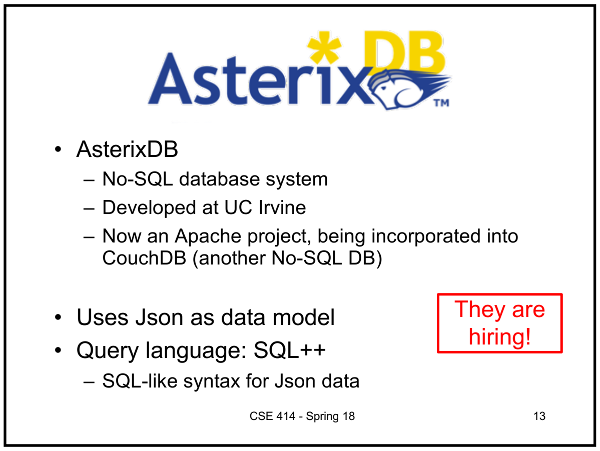
```
{“Person”:
  [{"name": "John",
    "phone":3646,
    "Orders":[
     {"date":2002,"product":"Gizmo"},
     {“date”:2004,"product":"Gadget"}
    ]
   },
   {"name": "Sue",
    "phone":6343,
    "Orders":[
    {"date":2002,"product":"Gadget"}
    ]
   }
  ]
}
```

Discussion: Why Semi-Structured Data?

- Semi-structured data model is good as *data exchange formats*
  - i.e., exchanging data between different apps
  - Examples: XML, JSon, Protobuf (protocol buffers)
- Increasingly, systems use them as a data model for databases:
  - SQL Server supports for XML-valued relations
  - CouchBase, MongoDB: JSon as data model
  - Dremel (BigQuery): Protobuf as data model

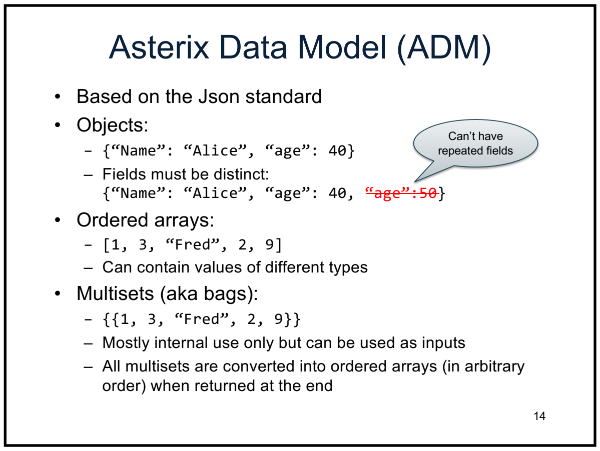
Query Languages for Semi-Structured Data

- XML: XPath, XQuery (see textbook)
  - Supported inside many RDBMS (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
  - Asterix: SQL++ (based on SQL)
  - MongoDB: has a pattern-based language
  - JSONiq http://www.jsoniq.org/



- AsterixDB
  - No-SQL database system
  - Developed at UC Irvine
  - Now an Apache project, being incorporated into CouchDB (another No-SQL DB)
- Uses Json as data model
- Query language: SQL++
  - SQL-like syntax for Json data

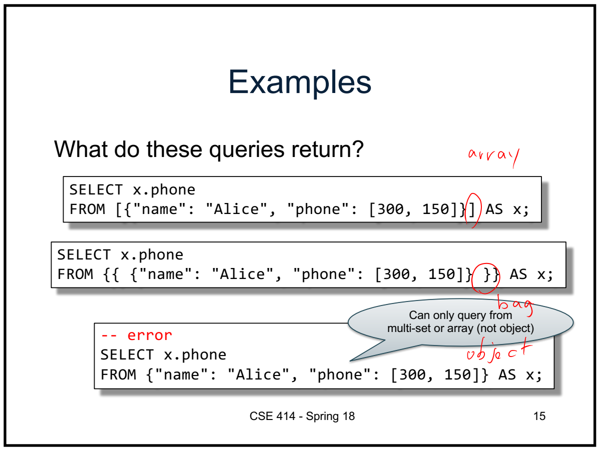
They are hiring!



Asterix Data Model (ADM)

- Based on the Json standard
- Objects:
  - `{“Name”: “Alice”, “age”: 40}`
  - Fields must be distinct:
    `{“Name”: “Alice”, “age”: 40, ~~“age”:50~~}`
- Ordered arrays:
  - `[1, 3, “Fred”, 2, 9]`
  - Can contain values of different types
- Multisets (aka bags):
  - `{{1, 3, “Fred”, 2, 9}}`
  - Mostly internal use only but can be used as inputs
  - All multisets are converted into ordered arrays (in arbitrary order) when returned at the end

Can't have repeated fields



Examples

What do these queries return?

array

```
SELECT x.phone
FROM [{"name": "Alice", "phone": [300, 150]}] AS x;
```

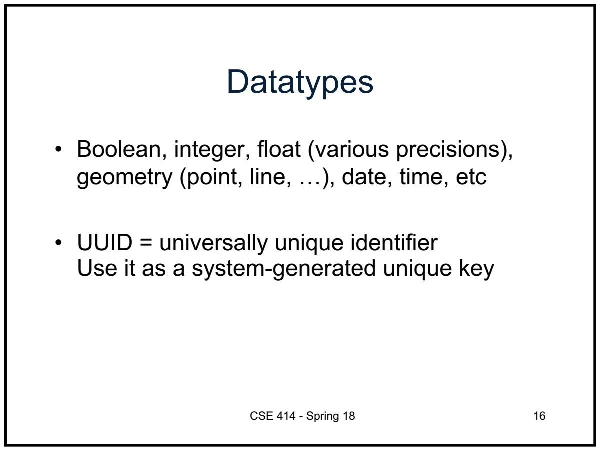
```
SELECT x.phone
FROM {{ {"name": "Alice", "phone": [300, 150]} }} AS x;
```

bag

```
-- error
SELECT x.phone
FROM {"name": "Alice", "phone": [300, 150]} AS x;
```

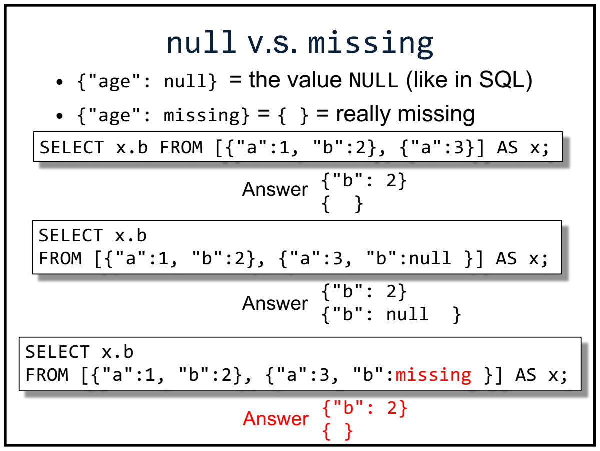
Can only query from multi-set or array (not object)

object



Datatypes

- Boolean, integer, float (various precisions), geometry (point, line, …), date, time, etc
- UUID = universally unique identifier
  Use it as a system-generated unique key



`null` v.s. `missing`

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

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

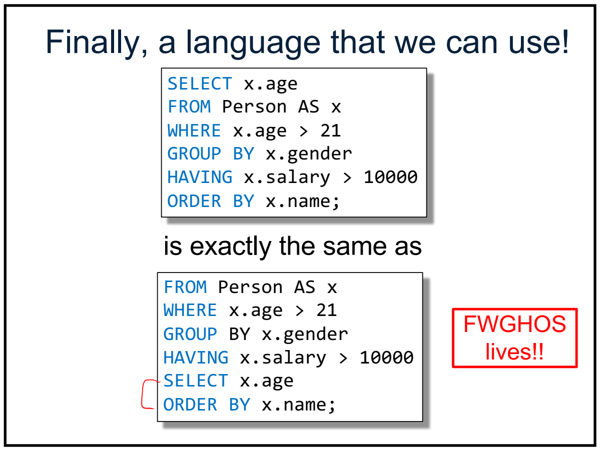
Answer
```
{"b": 2}
{  }
```

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

Answer
```
{"b": 2}
{"b": null  }
```

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

Answer
```
{"b": 2}
{ }
```



Finally, a language that we can use!

```
SELECT x.age
FROM Person AS x
WHERE x.age > 21
GROUP BY x.gender
HAVING x.salary > 10000
ORDER BY x.name;
```

is exactly the same as

```
FROM Person AS x
WHERE x.age > 21
GROUP BY x.gender
HAVING x.salary > 10000
SELECT x.age
ORDER BY x.name;
```

FWGHOS lives!!

SQL++ Overview

- Data Definition Language: create a
  - Type
  - Dataset (like a relation)
  - Dataverse (a collection of datasets)
  - Index
    - For speeding up query execution
- Data Manipulation Language: `SELECT-FROM-WHERE`

Dataverse

A Dataverse is a Database
(i.e., collection of tables)

```
CREATE DATAVERSE myDB
CREATE DATAVERSE myDB IF NOT EXISTS

DROP DATAVERSE myDB
DROP DATAVERSE myDB IF EXISTS

USE myDB
```

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 myDB;
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", ~~"phone": "123456789"~~}

Open Types

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
   name: string,
   age: int,
   email: string?
}
```

{"name": "Alice", "age": 30, "email": "a@alice.com"}

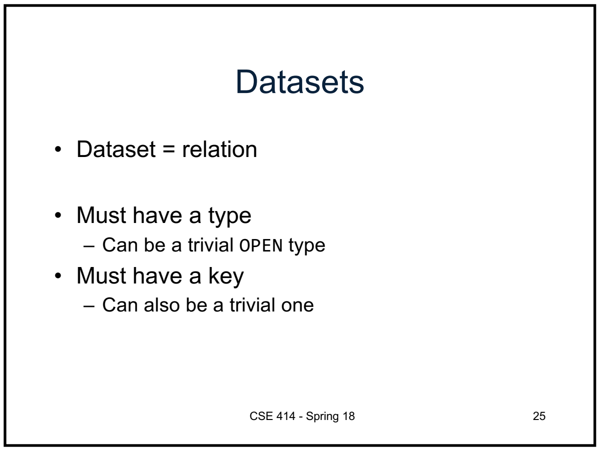
{"name": "Bob", "age": 40}

-- now it's OK:
{"name": "Carol", "age":20, ~~"phone": "123456789"~~}

Types with Nested Collections

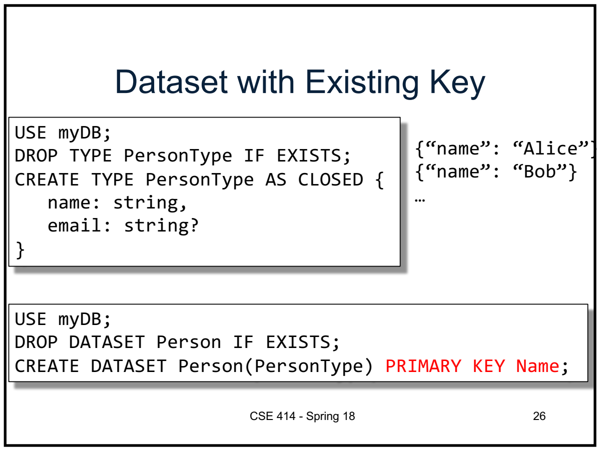
```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   phone: [string]
}
```

{"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 also be a trivial one

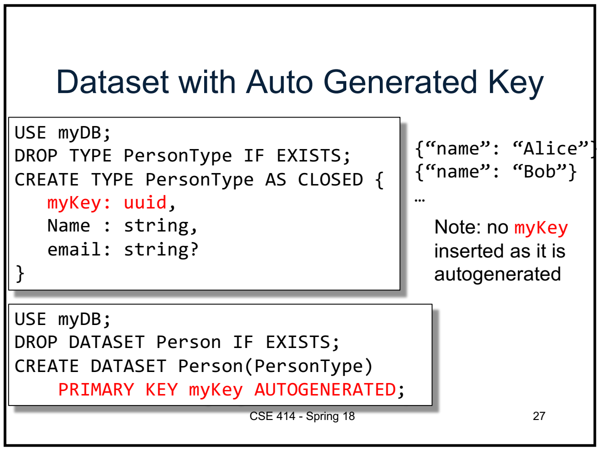


Dataset with Existing Key

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   name: string,
   email: string?
}
```

```
{“name”: “Alice”}
{“name”: “Bob”}
…
```

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



Dataset with Auto Generated Key

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

```
{“name”: “Alice”}
{“name”: “Bob”}
…
```

Note: no `myKey` inserted as it is autogenerated

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