

CSE 344

APRIL 16TH – SEMI-STRUCTURED DATA

ADMINISTRATIVE MINUTIAE

- **HW3 due Wednesday**
- **OQ4 due Wednesday**
- **HW4 out Wednesday (Datalog)**
- **Exam May 9th**
 - 9:30-10:20

WHERE WE ARE

So far we have studied the *relational data model*

- Data is stored in tables(=relations)
- Queries are expressions in SQL, relational algebra, or Datalog

Today: Semistructured data model

- Popular formats today: XML, JSon, protobuf

DATA MODELS

Taxonomy based on data models:

Key-value stores

- e.g., Project Voldemort, Memcached

Document stores

- e.g., SimpleDB, CouchDB, MongoDB



Extensible Record Stores

- e.g., HBase, Cassandra, PNUTS

MOTIVATION

In Key, Value stores, the Value is often a very complex object

- Key = '2010/7/1', Value = [all flights that date]

Better: allow DBMS to understand the *value*

- Represent *value* as a JSON (or XML...) document
- [all flights on that date] = a JSON file
- May search for all flights on a given date

DOCUMENT STORES FEATURES

Data model: (key,document) pairs

- Key = string/integer, unique for the entire data
- Document = JSon, or XML

Operations

- Get/put document by key
- Query language over JSon

Distribution / Partitioning

- Entire documents, as for key/value pairs

We will discuss JSon

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 SYNTAX

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


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

Semistructured 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 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"

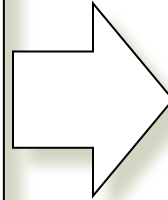
Ordered lists of values:

- [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

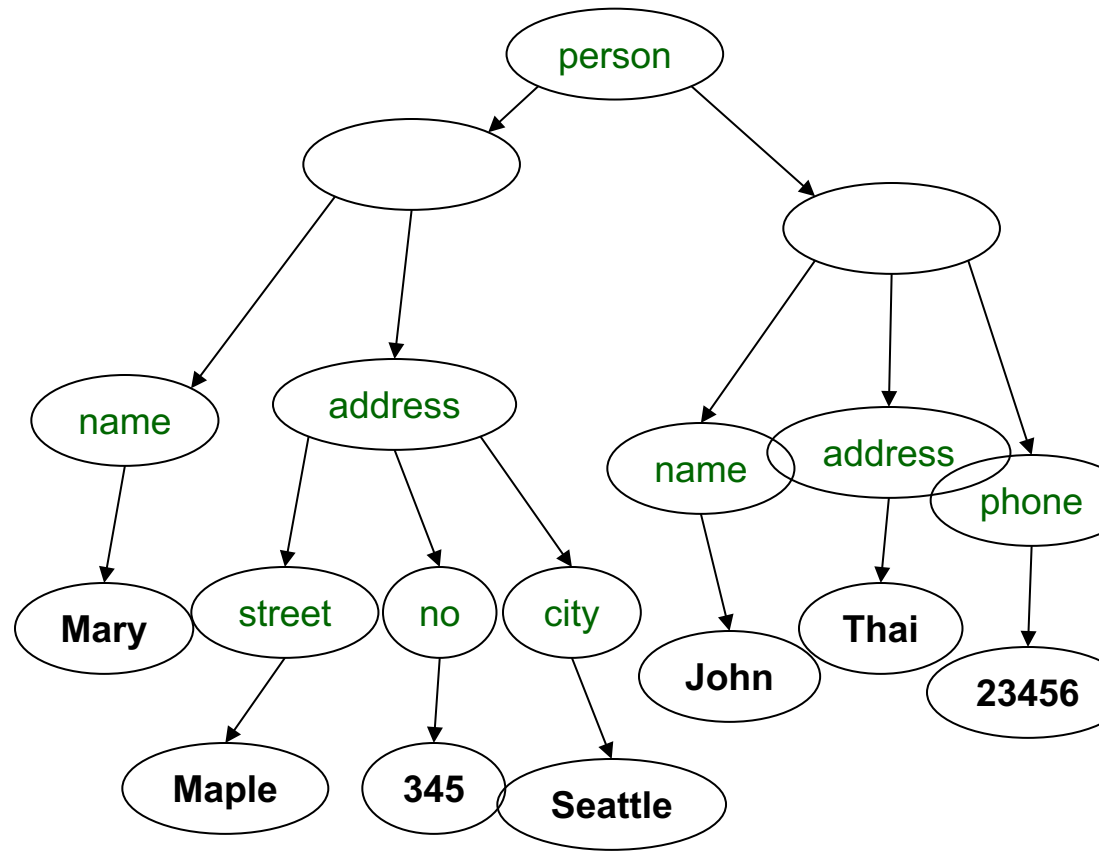
String = double-quoted

Boolean = true or false

null empty

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

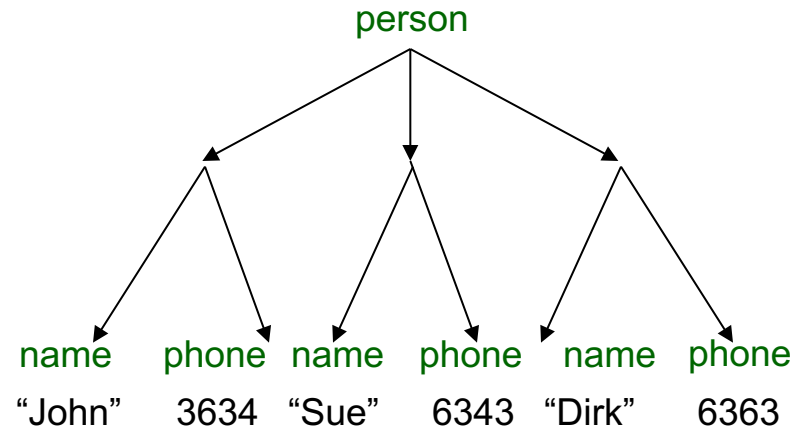
Consequence: JSON is much more flexible

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" } ]  
}
```

no phone !

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:

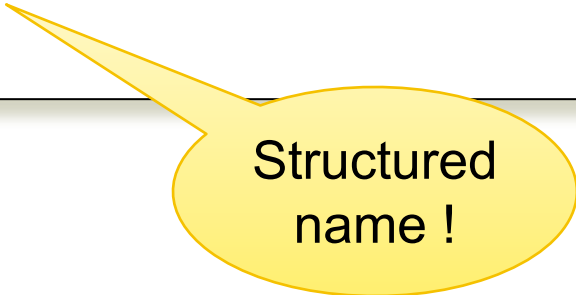
name	phone	
Mary	2345	3456

???

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 }  
  ]  
}
```



Structured
name !

Nested collections

Heterogeneous collections

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

JSon:

- CouchBase: N1QL, may be replaced by AQL (better designed)
- Asterix: SQL++ (based on SQL)
- MongoDB: has a pattern-based language
- JSONiq <http://www.jsoniq.org/>

ASTERIXDB AND SQL++

AsterixDB

- No-SQL database system
- Developed at UC Irvine
- Now an Apache project
- Own query language: AsterixQL or AQL, based on XQuery


SQL++

- SQL-like syntax for AsterixQL

ASTERIX DATA MODEL (ADM)

Objects:

- {"Name": "Alice", "age": 40}
- Fields must be distinct:
{"Name": "Alice", "age": 40, ~~"age": 50~~}



Can't have
repeated fields

Arrays:

- [1, 3, "Fred", 2, 9]
- Note: can be heterogeneous

Multisets:

- {{1, 3, "Fred", 2, 9}}

EXAMPLES

Try these queries:

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

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

Can only select from
multi-set or array

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


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}] x;
```

```
{ "b": { "int64": 2 } }  
{ }
```

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

```
{ "b": { "int64": 2 } }  
{ }
```

SQL++ OVERVIEW

Data Definition Language (DDL): create a

- Dataverse
- Type
- Dataset
- Index

Data Manipulation Language (DML): select-from-where

DATAVERSE

A Dataverse is a Database

```
CREATE DATAVERSE lec344
```

```
CREATE DATAVERSE lec344 IF NOT EXISTS
```

```
DROP DATAVERSE lec344
```

```
DROP DATAVERSE lec344 IF EXISTS
```

```
USE lec344
```

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 lec344;  
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 lec344;  
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", "phone": "123456789"}
```

TYPES WITH NESTED COLLECTIONS

```
USE lec344;  
DROP TYPE PersonType IF EXISTS;  
CREATE TYPE PersonType AS CLOSED {  
  Name : string,  
  phone: [string]  
}
```

```
{"Name": "Carol", "phone": ["1234"]}  
{"Name": "David", "phone": ["2345", "6789"]}  
{"Name": "Eric", "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 lec344;  
DROP TYPE PersonType IF EXISTS;  
CREATE TYPE PersonType AS CLOSED {  
  Name : string,  
  email: string?  
}
```

```
{"Name": "Alice"}  
{"Name": "Bob"}  
...
```

```
USE lec344;  
DROP DATASET Person IF EXISTS;  
CREATE DATASET Person(PersonType) PRIMARY KEY Name;
```

DATASET WITH AUTO GENERATED KEY

```
USE lec344;  
DROP TYPE PersonType IF EXISTS;  
CREATE TYPE PersonType AS CLOSED {  
  myKey: uuid,  
  Name : string,  
  email: string?  
}
```

```
{"Name": "Alice"}  
{"Name": "Bob"}  
...
```

Note: no **myKey**
since it will be
autogenerated

```
USE lec344;  
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

EXAMPLE

mondial.adm is totally semistructured:

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

country	continent	organization	sea	...	mountain	desert
[{"name": "Albania", ...}, {"name": "Greece", ...}, ...]

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

Country:

-car_code	name	...	ethnicgroups	religions	...	city
AL	Albania	...	[...]	[...]	...	[...]
GR	Greece	...	[...]	[...]	...	[...]
...			

INDEXES

Can declare an index on an attribute of a top-most collection

Available:

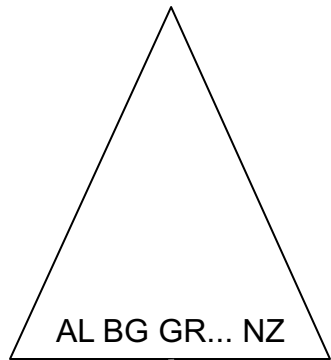
- BTREE: good for equality and range queries
E.g. name="Greece"; $20 < \text{age}$ and $\text{age} < 40$
- RTREE: good for 2-dimensional range queries
E.g. $20 < x$ and $x < 40$ and $10 < y$ and $y < 50$
- KEYWORD: good for substring search

INDEXES

Cannot index inside
a nested collection

```
USE lec344;  
CREATE INDEX countryID  
ON country(`-car_code`)  
TYPE BTREE;
```

```
USE lec344;  
CREATE INDEX cityname  
ON country(city.name)  
TYPE BTREE;
```



Country:

-car_code	name	...	ethnicgroups	religions	...	city
AL	Albania	...	[...]	[...]	...	[...]
GR	Greece	...	[...]	[...]	...	[...]
...			
BG	Belgium	...				
...						