CSE 344

JULY 11TH

SEMI-STRUCTURED DATA
ADMINISTRATIVE MINUTIAE

• HW3 due today

• HW4 out tomorrow (Datalog)

• WQ4 & WQ5 due Friday
  • relational algebra & Datalog
So far we have studied the relational data model

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

**Traditional RDBMSs cannot scale to support modern web apps**

- limited to one (big) machine
- scale web servers only until the DB becomes the bottleneck
Scale up by spreading the data or requests across machines

- BUT consistency becomes much harder
  - may need to give up on consistency checks involving multiple rows
  - e.g., no checking of foreign key constraints
  - can still check column non-null, greater than 0, etc.
- BUT joins become much harder if data is spread across machines
  - may need to give up on these
  - (we will see how to do these in a couple weeks)
NOSQL REVIEW

Original NoSQL systems put scalability before everything else

- For **OLTP** workloads only: users read/write small amount of data
  - huge database but each request looks at only a small part
- Simplest version *efficiently* supports only get/put of (key, value) pairs
  - provide no ability to join
  - provide no ability to select on anything but key!
  - these are “distributed hash tables” not databases
- Other types support extensible columns or documents
  - can efficiently select on anything in one row, but still no joins
- Scale data by partitioning across many servers
  - primary key is hashed to choose server where row lives
  - row-level consistency takes no inter-machine communication
NOSQL REVIEW

Two standard techniques (for distributed systems in general)

- partitioning: each row lives on only one machine
  - good for OLTP
  - consistency checks can be done on one machine
- replication: each row lives on all (or some) machines
  - bad for OLTP: machines could have different versions of a row
  - good for OLAP, assuming each machine has all the data
  - have all the data needed for joins on the machine
  - does not scale to large data, just large users / requests
- modern systems use both
  - e.g., partition primary copy, replicate (stale) backups
NOSQL REVIEW

Newer NoSQL systems can support full feature set
  • research systems (e.g., Asterix) do this now
  • expect to see production systems in the future

NoSQL now sometimes used to simply mean non-relational data
  • semi-structured data (documents)
WHERE WE ARE

Today: Semistructured data model

- Popular formats today: XML, JSon, protobuf
  - book discusses XML (out of favor now)
  - we will discuss JSon
  - (protobuf is just a binary / condensed form of this)
- semi-structured data is more flexible for users
- no free lunch: lack of structure also has costs
  - less work when writing
  - more work when querying
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.

(History: easiest data format within the browser)

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

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.

- Rows replaced by objects

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” (or “field”)

Ordered lists of values:

- `[obj1, obj2, obj3, …]`
AVOID USING DUPLICATE KEYS

The standard allows them, but many implementations don’t

```json
{"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!

```json
{
  "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 (but wasteful)

JSon is also **semistructured data**

- (more soon...)
# 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>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>Sue</td>
<td>6343</td>
</tr>
<tr>
<td>Dirk</td>
<td>Dirk</td>
<td>6363</td>
</tr>
</tbody>
</table>

```
{ 
  "person": [ 
    { "name": "John", "phone": 3634 },
    { "name": "Sue", "phone": 6343 },
    { "name": "Dirk", "phone": 6363 }
  ]
}
```
MAPPING RELATIONAL DATA TO JSON

May inline foreign keys

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
</tbody>
</table>

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

```json
{"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:**

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

Could represent in a table with nulls:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1234</td>
</tr>
<tr>
<td>Joe</td>
<td>-</td>
</tr>
</tbody>
</table>
JSON=SEMI-STRUCTURED DATA (2/3)

Repeated attributes

```json
{"person":
  [{"name":"John", "phone":1234},
   {"name":"Mary", "phone":[1234,5678]}]
}
```

Impossible in one table:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>2345</td>
</tr>
<tr>
<td></td>
<td>3456</td>
</tr>
</tbody>
</table>

Two phones!
Attributes with different types in different objects

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

Nested collections

Heterogeneous collections

- Downside: you need to think about these cases in queries!
QUERY LANGUAGES FOR SS DATA

XML: XPath, XQuery (textbook)
- Supported inside many RDBMS (SQL Server, DB2, Oracle)
- Several standalone XPath/XQuery engines
- XPath widely used in the browser: CSS, JQuery

JSON:
- CouchBase: N1QL, may be replaced by AQL (better designed)
- Asterix: SQL++ (based on SQL)
- MongoDB: has a pattern-based language
ASTERIXDB AND SQL++

AsterixDB

- NoSQL 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\}

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;
```

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

Can only select from multi-set or array
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

**Semi-structured data**

- type defines the structured part
- rest is unstructured
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"}
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
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   email: string?
};

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

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

USE lec344;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;

{“Name”: “Alice”}
{“Name”: “Bob”}
...

Note: no myKey since it will be 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”: [...]} }

<table>
<thead>
<tr>
<th>country</th>
<th>continent</th>
<th>organization</th>
<th>sea</th>
<th>...</th>
<th>mountain</th>
<th>desert</th>
</tr>
</thead>
<tbody>
<tr>
<td>[{“name”:”Albania”,...}, {“name”:”Greece”,...}, ...]</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

Country:

<table>
<thead>
<tr>
<th>-car_code</th>
<th>name</th>
<th>...</th>
<th>ethnicgroups</th>
<th>religions</th>
<th>...</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td></td>
<td>[ ... ]</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td></td>
<td>[ ... ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INDEXES

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

- used to improve query performance
  - allows DBMS to perform certain lookups efficiently
- (more next week...)

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
INDEXES

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

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

Country:

<table>
<thead>
<tr>
<th>-car_code</th>
<th>name</th>
<th>ethnicgroups</th>
<th>religions</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>BG</td>
<td>Belgium</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>