Lecture 22: NoSQL & JSON
(mostly not in textbook…
only Ch 11.1)
Announcements

• HW7 Due Tomorrow (last assignment)
• Ask if you are unsure about how many late days you have left.
NoSQL
NoSQL Motivation

- Originally motivated by Web 2.0 applications

- Goal is to scale simple OLTP-style workloads to millions or billions of users
  - Ex: Facebook has 1.3B daily active users
    - use often correlated in time within in each region
    - > 10M req/sec if 25% of users arrive within one hour
    - SQL Server would crumble under that workload

- Users are doing both updates and reads
What is the Problem?

• Single server DBMS are too small for Web data
• **Solution**: scale out to multiple servers

• This is hard for the *entire* functionality of DMBS
  – as we will see next…

• NoSQL: reduce functionality for easier scaling
  – Simpler data model
  – Fewer guarantees
Serverless Architecture

SQLite:
- One data file
- One user
- One DBMS application

- Scales well! (many apps)
- But only a limited number of scenarios work with such model
- (Can be in browser / phone!)

Desktop

User

DBMS Application (SQLite)

File

Data file

Disk
Client-Server Architecture

- One server running the database
- Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol

Supports many apps and many users simultaneously
Client-Server

• One server that runs the DBMS (or RDBMS):
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)
• Many clients run apps and connect to DBMS
  – Microsoft’s Management Studio (for SQL Server), or
  – psql (for postgres)
  – Some Java program (HW7) or some C++ program
• Clients “talk” to server using JDBC/ODBC protocol
3-Tiered Architecture
3-Tiered Architecture

DB Server

File 1
File 2
File 3

Web-based applications

App+Web Server

Connection (e.g., JDBC)

HTTP/SSL

Browser

CSE 344 - Summer 2017
Why don’t we replicate the DB server too?
Replicating the Database

• Much harder, because the state must be unique, in other words the database must act as a whole
  – Current DB instance must be *consistent always*
    • Ex: foreign keys must exist
    • as a result, some updates must occur simultaneously

• Two basic approaches:
  – Scale up through *partitioning*
  – Scale up through *replication*
Scale Through **Partitioning**

- Partition the database across many machines in a cluster
  - Database could fit in main memory
  - Queries spread across these machines
- Can increase throughput
- Easy for (simple) writes but reads become harder
Scale Through Replication

- Create multiple copies of each database partition
- Spread queries across these replicas
- Can increase throughput and lower latency
- Can also improve fault-tolerance
- Easy for reads but writes become harder
NoSQL 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
Key-Value Stores Features

- **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)

- **Operations**
  - Get(key), Put(key, value)
  - Operations on value not supported

- **Distribution / Partitioning**
  - No replication: key k is stored at server h(k)
  - 3-way replication: key k stored at h1(k), h2(k), h3(k)

How does get(k) work? How does put(k, v) work?
Example

• How would you represent the Flights data as key, value pairs?

• Option 1: key=fid, value=entire flight record

• Option 2: key=date, value=all flights that day

• Option 3: key=(origin,dest), value=all flights between
Key-Value Stores Internals

• Data remains in main memory
  – one implementation: distributed hash table
• Most systems also offer a persistence option
• Others use replication to provide fault-tolerance
  – Asynchronous or synchronous replication
  – Tunable consistency: read/write one replica or majority
• Some offer transactions others do not
  – multi-version concurrency control or locking
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
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
  – Limited, non-standard query language on JSON

• **Distribution / Partitioning**
  – Entire documents, as for key/value pairs

**Different From Key Value Store**

We will discuss JSON later today
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
Extensible Record Stores

• Based on Google’s BigTable
  – HBase is an open source implementation of BigTable

• Data model is rows and columns
  – can add both new rows and new columns

• Scalability by splitting rows and columns over nodes
  – Rows partitioned through hashing on primary key
  – Columns of a table are distributed over multiple nodes by using “column groups”
NoSQL Summary

• Simpler data model with weaker guarantees
• But they scale as far as we need them to

• Meanwhile…
  SQL systems continue to improve
Recent SQL Progress

• Modern systems need to store data across the globe
  – individual data centers go offline
  – need servers close to users to be efficient

• Speed of light is a fundamental limit
  – 200+ms latency (across US) is visible to users

• Systems must weaken guarantees

• Google’s Spanner (supports SQL):
  – write data over the whole globe (a bit slowly)
  – reads occur slightly in the past
JSon
An Example Semi-structured Data Format
Where We Are

• So far we have studied the *relational data model*
  – Data is stored in tables (relations)
  – Queries are expressions in the SQL / datalog / relational algebra

• Today: Semistructured data model
  – Popular formats today: XML, JSON, protobuf
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 – Personal History

• 10 years ago…
  – JavaScript interpreters were very slow
  – native browser function parsed JSON 100x faster

• XML was also an option, but
  – IE had a memory leak in its XML parser

• JSON used in Gmail etc. for this reason
• Spread organically to server-side systems
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 Syntax

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

Data made up of objects, lists, and atomic values (integers, floats, strings, booleans).
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

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

What is author?
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
  }
],
"phone": 23456}
```
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
  – also uses more space (but can be compressed)
• JSon is an example of semistructured data
Mapping Relational Data to JSON

Person

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
</tr>
</tbody>
</table>

```json
{
  "person": [
    {
      "name": "John",
      "phone": 3634
    },
    {
      "name": "Sue",
      "phone": 6343
    },
    {
      "name": "Dirk",
      "phone": 6383
    }
  ]
}
```
Mapping Relational Data to JSON

May inline foreign keys

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

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

```json
{
"Person":
[
{
"name": "John",
"phone": 3646,
"Orders":
[
{
"date": 2002,
"product": "Gizmo"
}
]
},
{
"name": "Sue",
"phone": 6343,
"Orders":
[
{
"date": 2002,
"product": "Gadget"
}
]
}
}
```
JSon Semi-structured Data

• 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

• 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>
</tbody>
</table>
JSOn Semi-structured Data

- Attributes with different types in different objects

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

- Nested collections
- Heterogeneous collections

CSE 344 - Summer 2017
JSON vs XML

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

```
<people>
  <person name=“Mary”>
    <address street=“Maple” no=“345” city=“Seattle”/>
  </person>
  <person name=“John”>
    <address country=“Thailand”/>
    <phone number=“2345678”/>
  </person>
</people>
```

JSON less verbose. XML can be more strict
YAML: Yet Another Markup Language

```yaml
{ "person": [ { "name": "Mary", "address": { "street": "Maple", "no": 345, "city": "Seattle" } }, { "name": "John", "address": "Thailand", "phone": 2345678 } ] }```
Next Time: Working with big data

• MapReduce = high-level programming model and implementation for large-scale parallel data processing

• Google: paper published 2004
• Free variant: Hadoop