Database Systems
CSE 414

Lecture 11: NoSQL & JSON
(mostly not in textbook… only Ch 11.1)
Announcements

• HW5 will be posted on Friday and due on Nov. 14, 11pm

• [No Web Quiz 5]

• Today’s lecture:
  – NoSQL & JSON
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 1.2B *daily* active users
    • use often correlated in time in each region
    • > 10M req/sec if 25% of users arrive within one hour
    • SQL Server would collapse 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 DBMS
  - as we will see next…

- NoSQL: reduce functionality for easier scaling
  - Simpler data model
  - Fewer guarantees
SQLite:
- One data file
- One user
- One DBMS application

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

Supports many apps and many users simultaneously

Server Machine

Client Applications

Connection (JDBC, ODBC)

• One server running the database
• Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol
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 (HW8) or some C++ program
- Clients “talk” to server using JDBC/ODBC protocol
3-Tiered Architecture
3-Tiered Architecture

Connection (e.g., JDBC)

DB Server

File 1
File 2
File 3

Web-based applications

App+Web Server

HTTP/SSL

Browser
3-Tiered Architecture

DB Server

File 1
File 2
File 3

Replicate App server for scaleup

Connection (e.g., JDBC)

App+Web Server

HTTP/SSL

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 always be consistent
    • 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

Transaction starts here Also touches data here

Three partitions
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 is 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

How does query processing work?
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
• No secondary indices
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

We will discuss JSon today or tomorrow
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
Prediction

• My guess: SQL will win again

• Pieces are out there already
  – Spanner: multi-node transactions
  – AsterixDB: multi-node query optimization

• For now, NoSQL still offers key benefits
JSon and Semi-structured Data
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: Semi-structured data model
  – Popular formats today: XML, JSON, protobuf
JSON

• 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 - 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 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

• Semi-structured 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
        }
    ]
}
```
JSon Terminology

• Curly braces hold objects
  – Each object is a list of name/value pairs separated by , (comma)
  – Each pair is a name 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

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

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

- 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
  – also uses more space (but can be compressed)
• JSon is an example of semi-structured data