Introduction to Database Systems
CSE 414

Lecture 11: NoSQL
Announcements

• HW 3 due Friday
  – Upload data with DataGrip editor – see message board
  – Azure timeout for question 5:
    • Try DataGrip or SQLite
• HW 2 Grades and Feedback out
  – Check feedback, some tag errors
• HW 4 posted today, due week from Tuesday
Class Overview

• Unit 1: Intro
• Unit 2: Relational Data Models and Query Languages
  • Unit 3: Non-relational data
    – NoSQL
    – Json
    – SQL++
• Unit 4: RDMBS internals and query optimization
• Unit 5: Parallel query processing
• Unit 6: DBMS usability, conceptual design
• Unit 7: Transactions
• Unit 8: Advanced topics (time permitting)
Two Classes of Database Applications

• OLTP (Online Transaction Processing)
  – Queries are simple lookups: 0 or 1 join
    E.g., find customer by ID and their orders
  – Many updates. E.g., insert order, update payment
  – Consistency is critical: transactions (more later)

• OLAP (Online Analytical Processing)
  – aka “Decision Support”
  – Queries have many joins, and group-by’s
    E.g., sum revenues by store, product, clerk, date
  – No updates
NoSQL Motivation

• Originally motivated by Web 2.0 applications
  – E.g. Facebook, Amazon, Instagram, etc
  – Web startups need to scale up from 10 to 100,000 users very quickly

• Needed: very large scale OLTP workloads
• Give up on consistency
• Give up OLAP
SQLite:
• One data file
• One user
• One DBMS application

• **Consistency** is easy
• But only a limited number of scenarios work with such model
RDBMS Review: Client-Server

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

Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol

Consistency is harder → transactions

Connection (JDBC, ODBC)
Client-Server

• **One server that runs the DBMS (or RDBMS):**
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)
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
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
Web Apps: 3 Tier

Browser

File 1
File 2
File 3
DB Server
Web Apps: 3 Tier

Connection (e.g., JDBC)

App+Web Server

HTTP/SSL

Browser

DB Server

File 1

File 2

File 3
Web Apps: 3 Tier

Web-based applications

Browser

Connection (e.g., JDBC)

HTTP/SSL

File 1

File 2

File 3

DB Server

App+Web Server
Web Apps: 3 Tier

Web-based applications

File 1
File 2
File 3

DB Server

Connection (e.g., JDBC)

App+Web Server

HTTP/SSL

15
Web Apps: 3 Tier

Web-based applications

File 1
File 2
File 3

DB Server

Connection (e.g., JDBC)

App+Web Server

HTTP/SSL

Replicate App server for scaleup

Why not replicate DB server?
Web Apps: 3 Tier

Replicate App server for scaleup

Web-based applications

File 1
File 2
File 3

DB Server

Connection (e.g., JDBC)

HTTP/SSL

App+Web Server

Why not replicate DB server? Consistency!
Replicating the Database

- Two basic approaches:
  - Scale up through **partitioning**
  - Scale up through **replication**

- **Consistency** is much harder to enforce
Scale Through Partitioning

- Partition the database across many machines in a cluster
  - Database now fits in main memory
  - Queries spread across these machines
- Can increase throughput
- Easy for writes but reads become expensive!

Application updates here

Three partitions

May also update here
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 expensive!

Three replicas

App 1 updates here only

App 2 updates here only

CSE 414 - Autumn 2018 20
Relational Model → NoSQL

• Relational DB: difficult to replicate/partition
  • Given `Supplier(sno, ...), Part(pno, ...), Supply(sno, pno)`
    – Partition: we may be forced to join across servers
    – Replication: local copy has inconsistent versions
    – Consistency is hard in both cases (why?)

• NoSQL: simplified data model
  – Given up on functionality
  – Application must now handle joins and consistency
Data Models

Taxonomy based on data models:

- **Key-value stores**
  - e.g., Project Voldemort, Memcached
- **Document stores**
  - e.g., SimpleDB, CouchDB, MongoDB
Key-Value Stores Features

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

• A function that maps any data to a “hash value” (e.g., an integer)
Aside: Hash Functions

- Example: data and hash value are integers
- Simple hash function:
  - $h(key) = key \% 42$;
  - $h(10) = 10$
  - $h(2) = 2$
  - $h(50) = 8$

- What does this have to do with data distribution?
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** – w/ hash function
  - 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?
Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

Example

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

How does query processing work?
Example

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

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

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)
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
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?
Data Models

Taxonomy based on data models:

• **Key-value stores**
  – e.g., Project Voldemort, Memcached

• **Document stores**
  – e.g., SimpleDB, CouchDB, MongoDB
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
Example: storing FB friends

As a graph

Peter
Mary
John
Phil

OR

Add new attributes?
Storing lists?

As a relation

<table>
<thead>
<tr>
<th>Person1</th>
<th>Person2</th>
<th>is_friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>John</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Mary</td>
<td>0</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Phil</td>
<td>Peter</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We will learn the tradeoffs of different data models later this quarter
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
{ "book": [
    {"id":"01",
     "language": "Java",
     "author": "H. Javeson",
     "year": 2015
    },
    {"id":"07",
     "language": "C++",
     "edition": "second",
     "author": "E. Sepp",
     "price": 22.25
    }
]}

CSE 414 - Autumn 2018
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

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

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

• Number

• String = double-quoted

• Boolean = true or false

• nullempy
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

<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>

```
{
  "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>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
</tr>
<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></th>
</tr>
</thead>
<tbody>
<tr>
<td>personName</td>
<td>date</td>
</tr>
<tr>
<td>John</td>
<td>2002</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</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"
        }
      ]
    }
  ]
}
```