CSE 344

JANUARY 31ST – SEMI-STRUCTURED DATA
ADMINISTRATIVE MINUTIAE

• HW3 due Friday
• OQ due Wednesday
• HW4 out Wednesday
• Exam next Friday
  • 3:30 - 5:00
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 100000 users very quickly

Needed: very large scale OLTP workloads
Give up on consistency
Give up OLAP
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

NoSQL: reduce functionality for easier scale up

• Simpler data model
• Very restricted updates
RDBMS REVIEW: SERVERLESS

SQLite:
One data file
One user
One DBMS application

Consistency is easy
But only a limited number of scenarios work with such model

Desktop
User
DBMS Application (SQLite)
File
Data file
Disk
RDBMS REVIEW: CLIENT-SERVER

One server running the database

Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol
RDBMS REVIEW: CLIENT-SERVER

Many users and apps
Consistency is harder \(\rightarrow\) transactions

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

DB Server

File 1

File 2

File 3
WEB APPS: 3 TIER

Browser

HTTP/SSL

DBC Server

Connection (e.g., JDBC)

App+Web Server
WEB APPS: 3 TIER

Web-based applications

Browser

File 1

File 2

File 3

Connection (e.g., JDBC)

DB Server

App+Web Server

HTTP/SSL
WEB APPS: 3 TIER

Web-based applications

File 1
File 2
File 3
DB Server

App+Web Server

Connection (e.g., JDBC)

HTTP/SSL
Replicate App server for scaleup

Web-based applications

Why not replicate DB server?
Web-based applications

Why not replicate DB server?
Consistency!

WEB APP + DB SERVER

Replicate App server for scaleup

Connection (e.g., JDBC)

HTTP/SSL

File 1
File 2
File 3

DB Server

App+Web Server
App+Web Server
App+Web Server
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

May also update 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 expensive!

App 1 updates here only
Three replicas
App 2 updates here only
RELATIONAL MODEL $\rightarrow$ 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

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)
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 – 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)
**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?
How would you represent the Flights data as key, value pairs?

EXAMPLE

How does query processing work?
How would you represent the Flights data as key, value pairs?

Option 1: key=fid, value=entire flight record
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 does query processing work?
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

Partitioning:

• Use a hash function $h$, and store every (key, value) pair on server $h(key)$
• In class: discuss get(key), and put(key, value)

Replication:

• Store each key on (say) three servers
• On update, propagate change to the other servers; eventual consistency
• Issue: when an app reads one replica, it may be stale

Usually: combine partitioning+replication
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
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

Data model is rows and columns

Scalability by splitting rows and columns over nodes

• Rows partitioned through sharding on primary key
• Columns of a table are distributed over multiple nodes by using “column groups”

HBase is an open source implementation of BigTable
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
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
    }
  ]
}
JSON VS RELATIONAL

Relational data model

- Rigid flat structure (tables)
- Schema must be fixed in advance
- 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 a comma.
- Each pair is a name followed by a colon (colon) followed by the value.

Square brackets hold arrays and values are separated by a 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!

```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

JSon = semistructured data
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>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
<td></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

### 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>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>personName</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<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"
  }]
}
}
```
Missing attributes:

```
{"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>
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>
Attributes with different types in different objects

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

Nested collections

Heterogeneous collections