Introduction to Database Systems
CSE 414

Lecture 12: NoSQL

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)

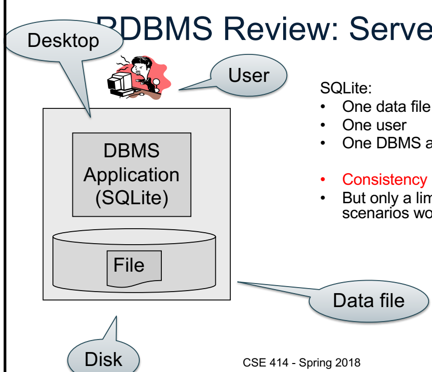
Two Classes of Database Applications

- 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 scaleup from 10 to 100000 users very quickly
- Needed: very large scale OLTP workloads
- Give up on consistency
- Give up OLAP

RDBMS Review: Serverless

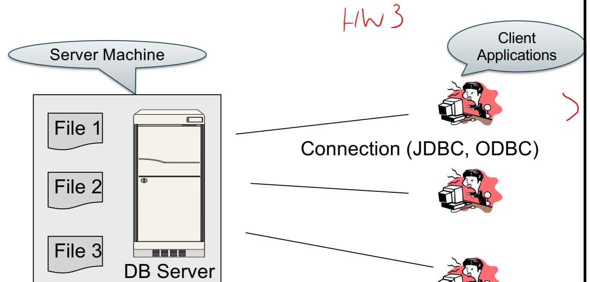


SQLite:
- One data file
- One user
- One DBMS application

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

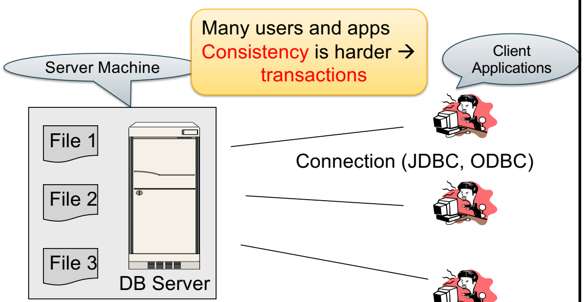
RDBMS Review: Client-Server

HW 3



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

RDBMS Review: Client-Server



- 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)

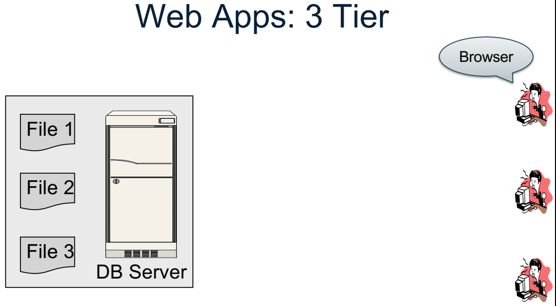
Client-Server

- 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

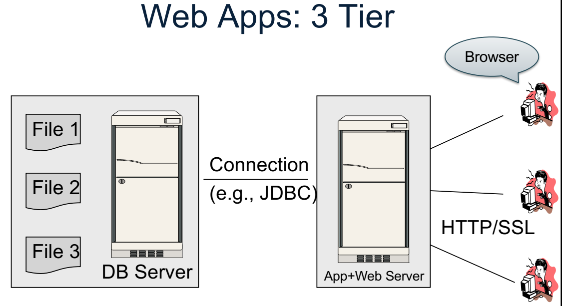
Client-Server

- 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

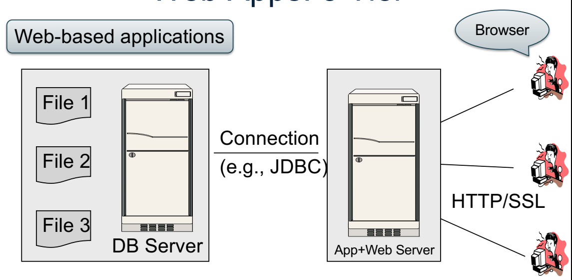
Web Apps: 3 Tier



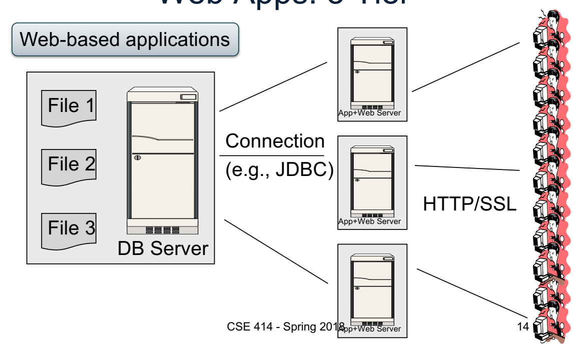
Web Apps: 3 Tier

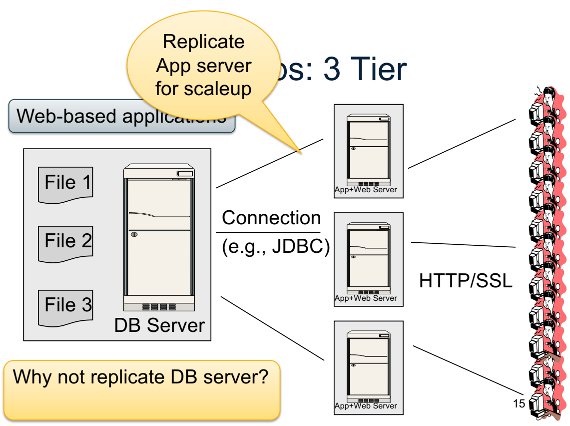


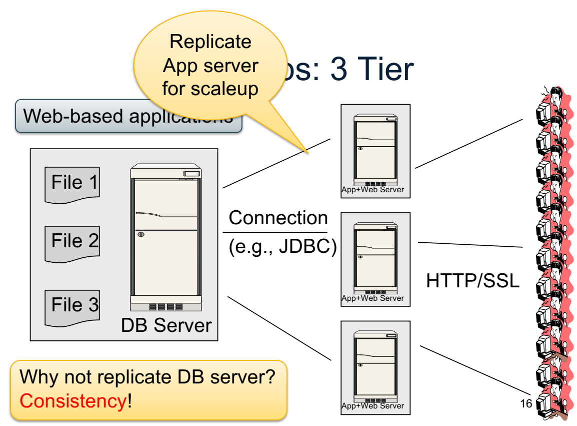
Web Apps: 3 Tier



Web Apps: 3 Tier





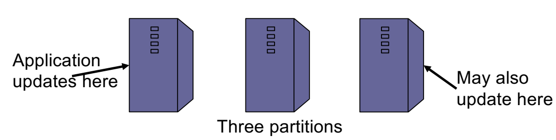


Replicating the Database

- Two basic approaches:
  - Scale up through partitioning
  - Scale up through replication
- Consistency is much harder to enforce

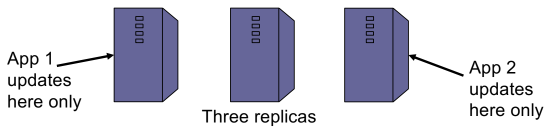
Scale Through Partitioning

- 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!



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!



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

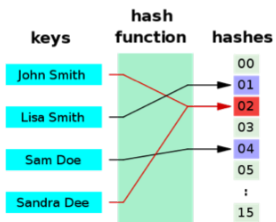
- **Data model**: (key,value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)
- **Operations**
  - `get(key)`, `put(key,value)`
  - Operations on value not supported

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?

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

How does query processing work?

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 does query processing work?

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
- 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

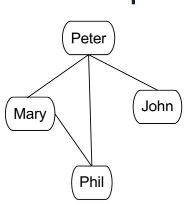
- In Key, Value stores, the Value is often a very complex object
  - Key = '2010/7/1', Value = [all flights that date]
- Better: allow DBMS to understand the *value*
  - Represent *value* as a JSON (or XML...) document
  - [all flights on that date] = a JSON file
  - May search for all flights on a given 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



OR

Add new attributes?

Storing lists?

Person1	Person2	is_friend
Peter	John	1
John	Mary	0
Mary	Phil	1
Phil	Peter	1
…	…	…

As a graph

As a relation

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

JSon Syntax

```
{ "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 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

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

⇒

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

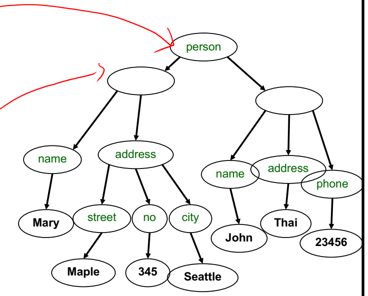
[ ]

JSon Datatypes

- Number
- String = double-quoted
- Boolean = true or false
- nullempty

JSon Semantics: a Tree !

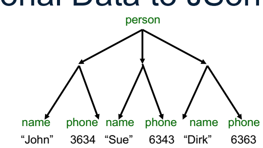
```
{“person”:
  [ {“name”: “Mary”,
     “address”:
        {“street”:“Maple”,
         “no”:345,
         “city”: “Seattle”}},
    {“name”: “John”,
     “address”: “Thailand”,
     “phone”:2345678}}
  ]
}
```



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



Person

name	phone
John	3634
Sue	6343
Dirk	6363

```
{“person”:
   [{“name”: “John”, “phone”:3634},
    {“name”: “Sue”,  ”phone”:6343},
    {“name”: “Dirk”,  ”phone”:6383}
   ]
}
```


Mapping Relational Data to JSon

May inline foreign keys

Person

name	phone
John	3634
Sue	6343

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

```
{“Person”:
   [{“name”: “John”,
     “phone”:3646,
     “Orders”:[{“date”:2002,
                “product”:”Gizmo”},
               {“date”:2004,
                “product”:”Gadget”}
              ]
    },
    {“name”: “Sue”,
     “phone”:6343,
     “Orders”:[{“date”:2002,
                “product”:”Gadget”}
              ]
    }
   ]
}
```