NoSQL I

Motivations and Data Model

Alvin Cheung
Aditya Parameswaran

Berkeley cs186

Two Classes of Relational Database Apps

- 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: we need transactions

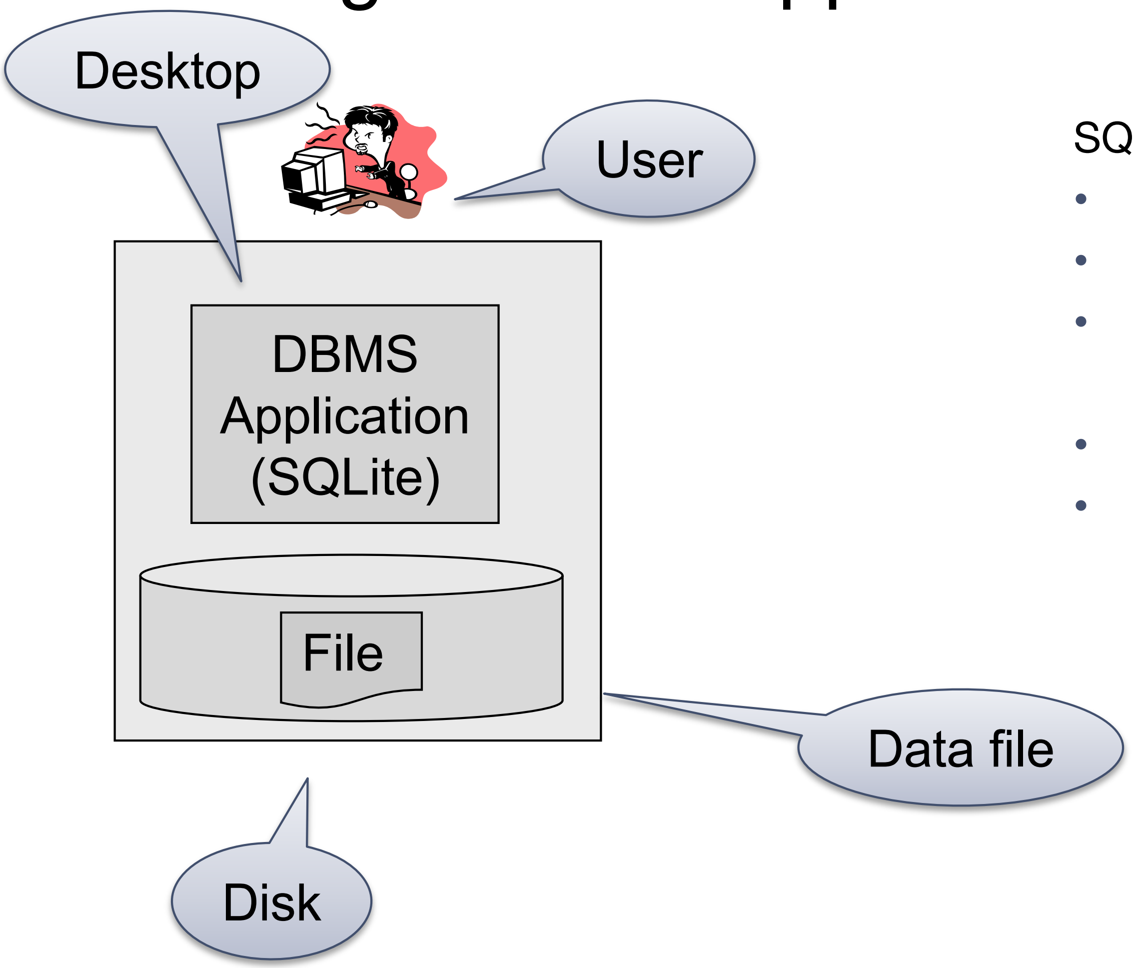
- OLAP (Online Analytical Processing)
  - aka "Decision Support"
  - Queries have many joins, and group-by's
    E.g., sum revenues by store, product, clerk, date
  - No updates

NoSQL Motivation

- Originally motivated by Web 2.0 applications
  - E.g., Facebook, Amazon, Instagram, etc
  - Startups need to scaleup from 10 to 10^7 clients quickly

- Needed: very large scale OLTP workloads
- Give up on consistency, give up OLAP
- NoSQL: reduce functionality
  - Simpler data model
  - Very restricted updates

Structuring RDBMS Apps: “Serverless”

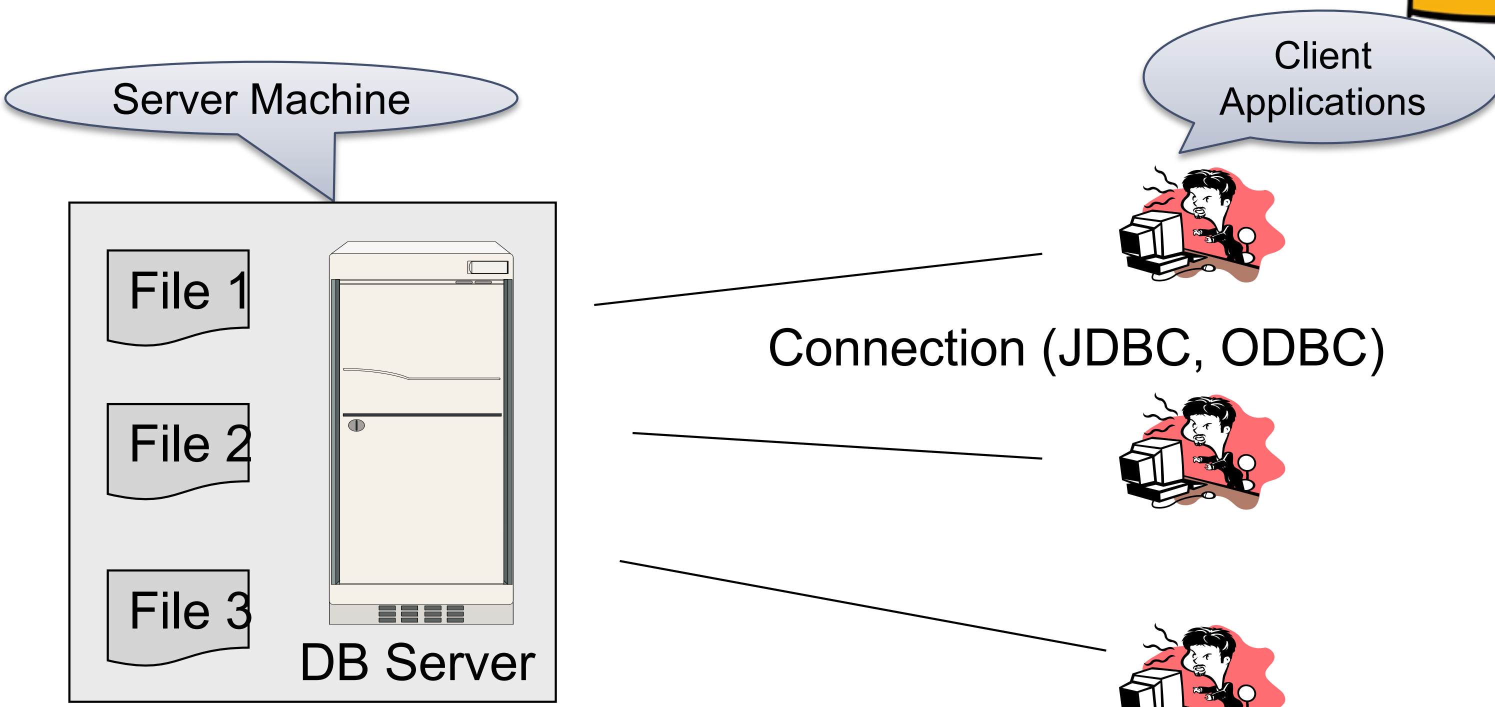


SQLite:

- One data file
- One user
- One DBMS application

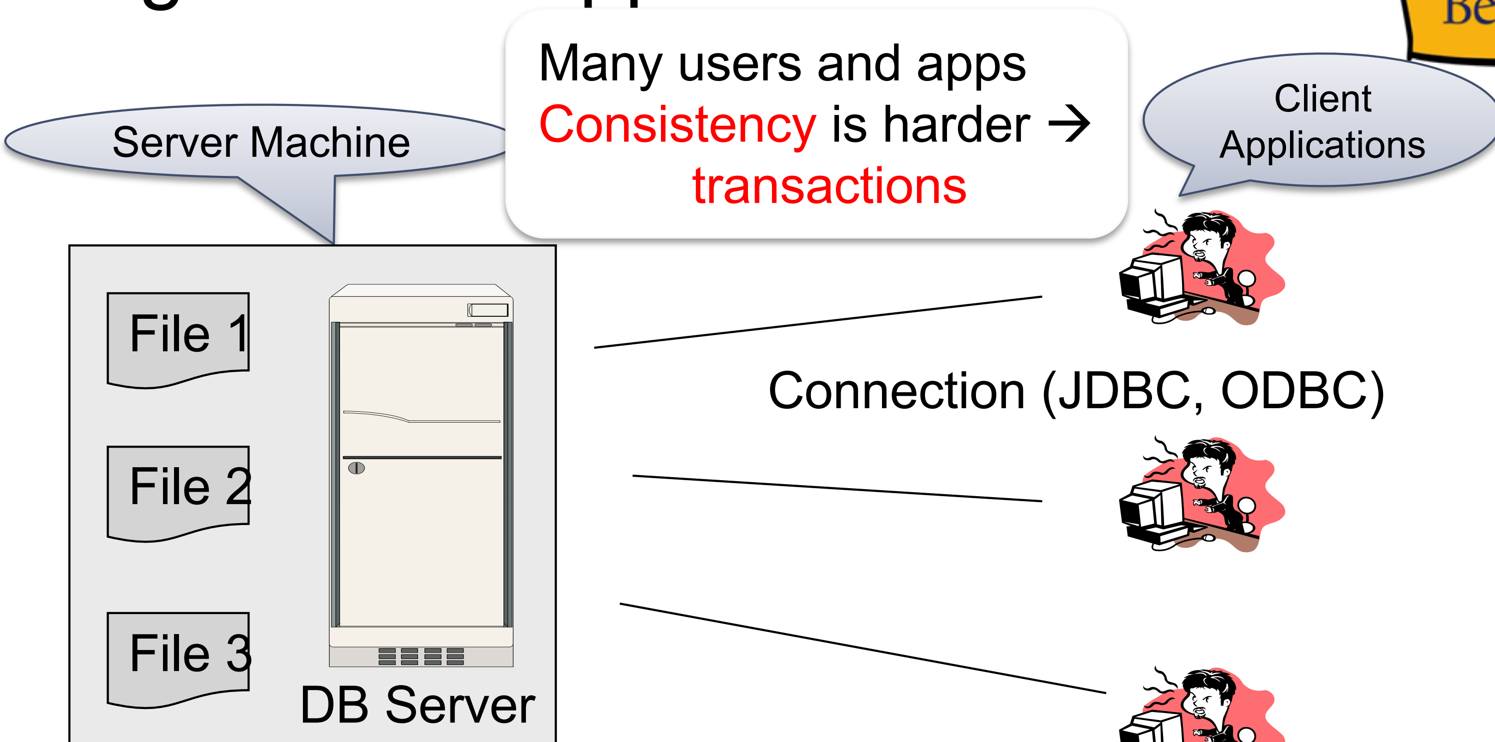
- Consistency is easy
- But only a limited number of scenarios work with such model

Structuring RDBMS Apps: Client-Server



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

Structuring RDBMS Apps: 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 (AWS, SQL Azure)

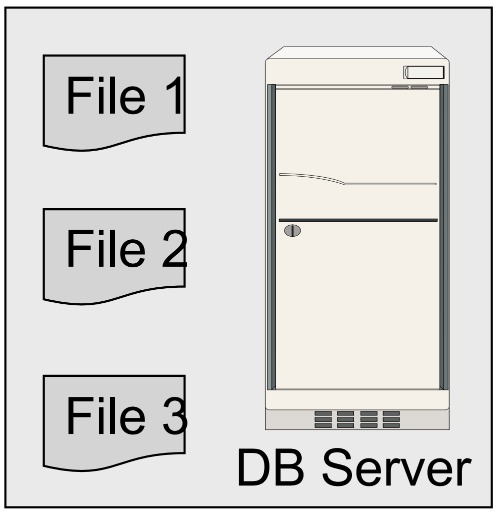
Client-Server

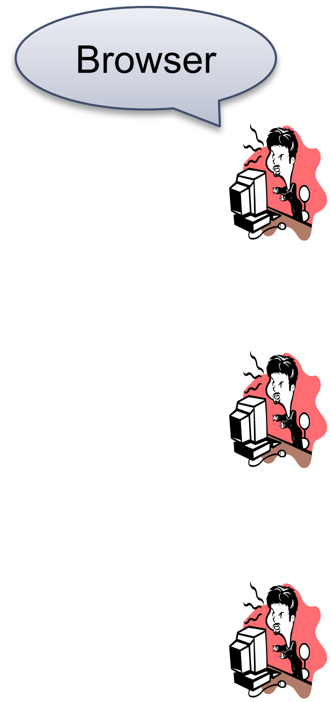
- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft's Management Studio (for SQL Server), or
  - psql (for postgres)
  - Your Java/C++/Python/etc program

Client-Server

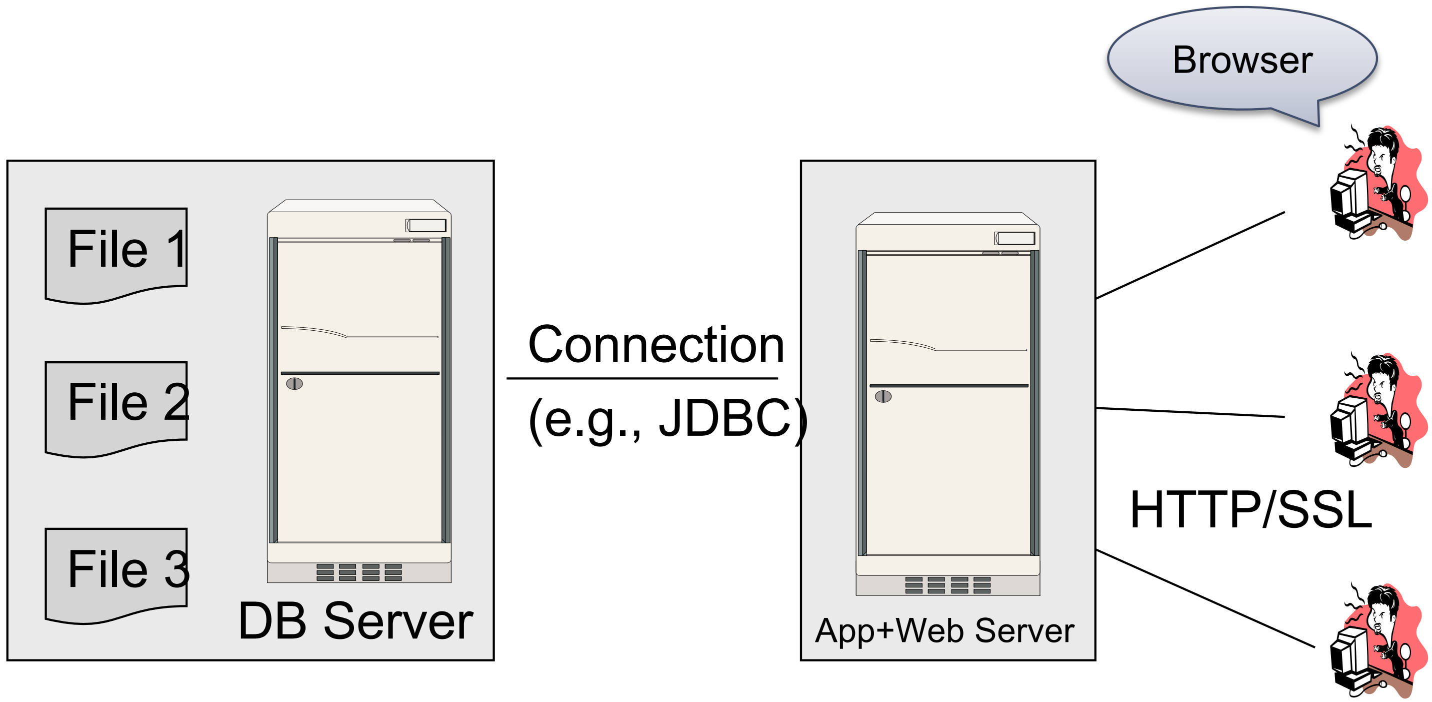
- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft's Management Studio (for SQL Server), or
  - psql (for postgres)
  - Your Java/C++/Python/etc program
- Clients "talk" to server using JDBC/ODBC protocol

Web Apps: 3 Tier

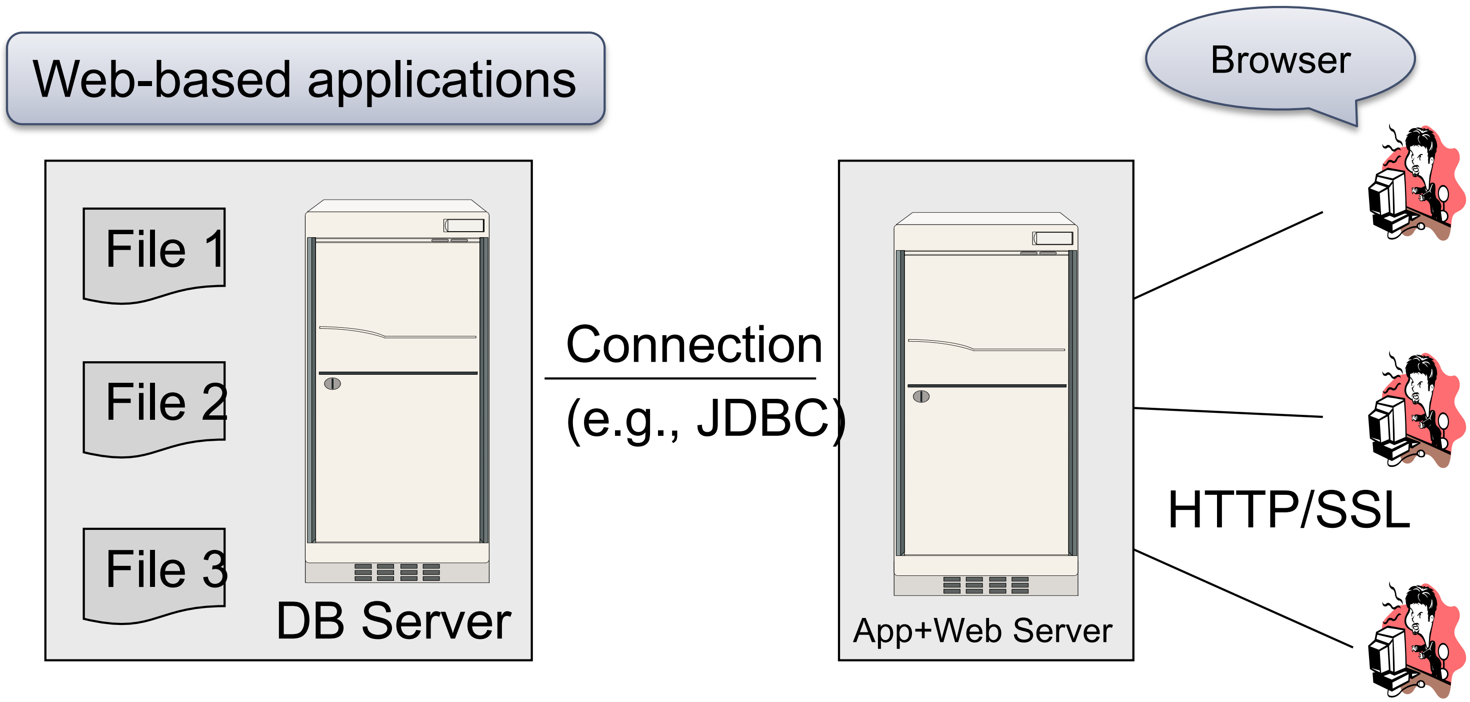




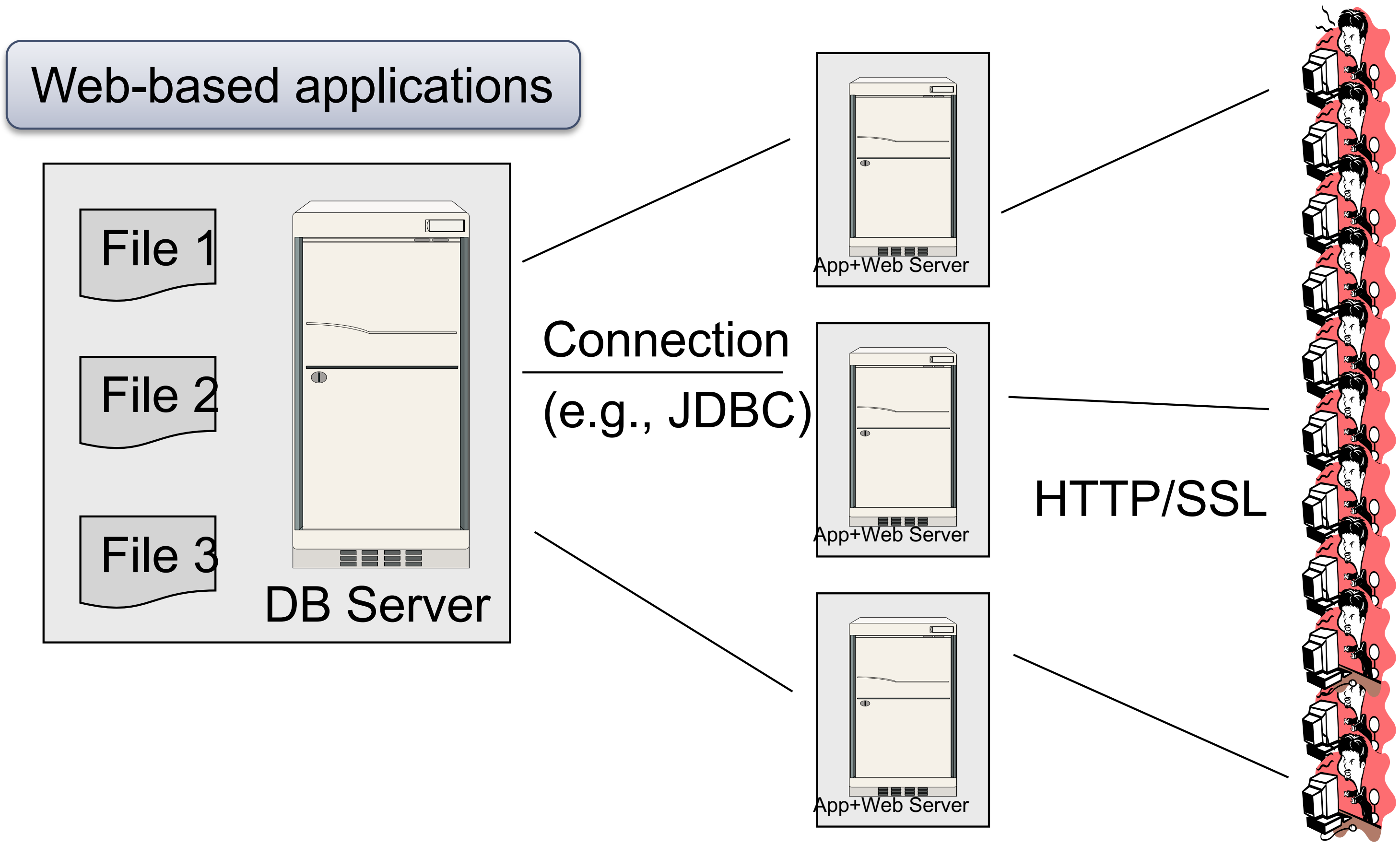
Web Apps: 3 Tier



Web Apps: 3 Tier



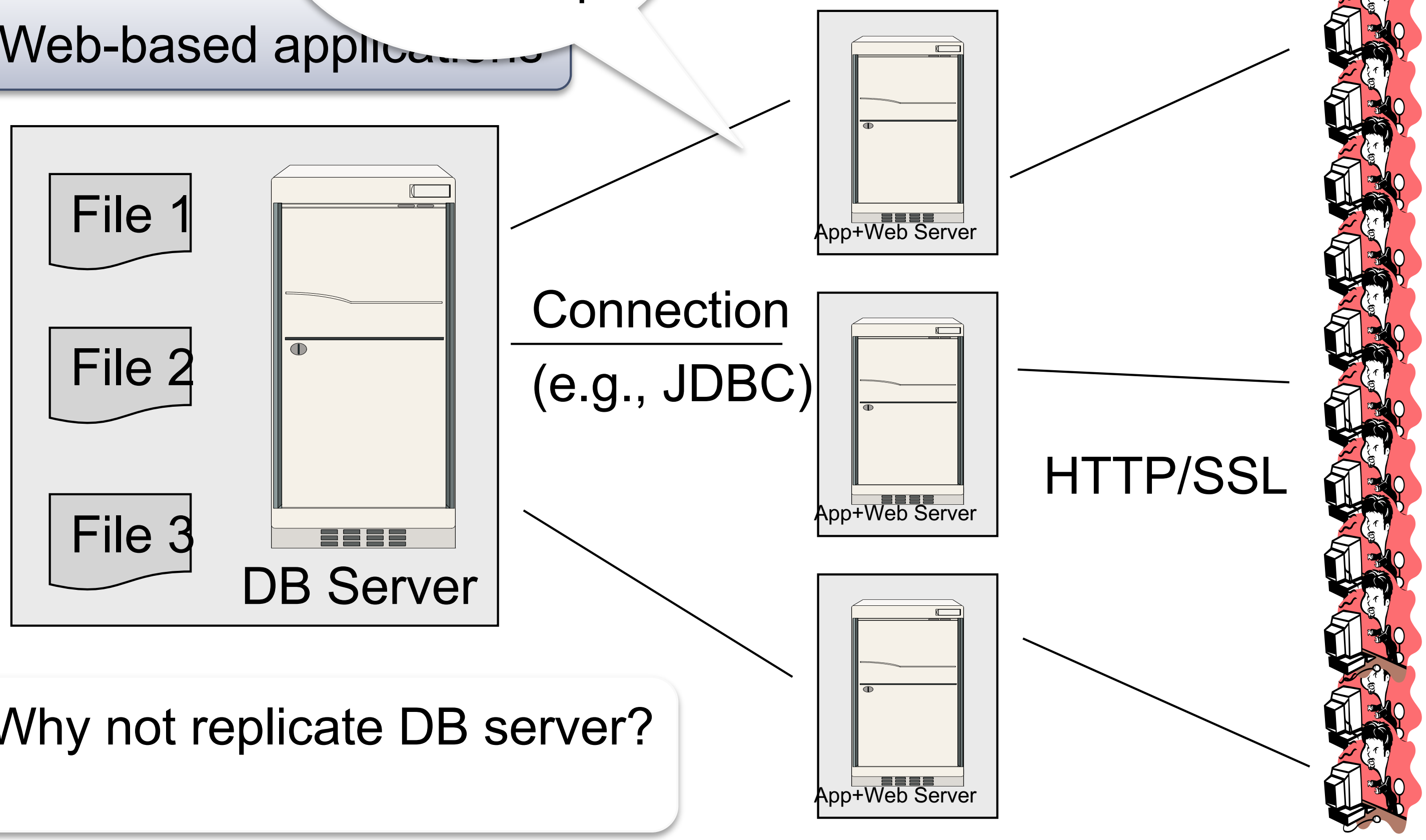
Web Apps: 3 Tier



Web Apps: 3 T

Replicate App server for scaleup

Web-based applications

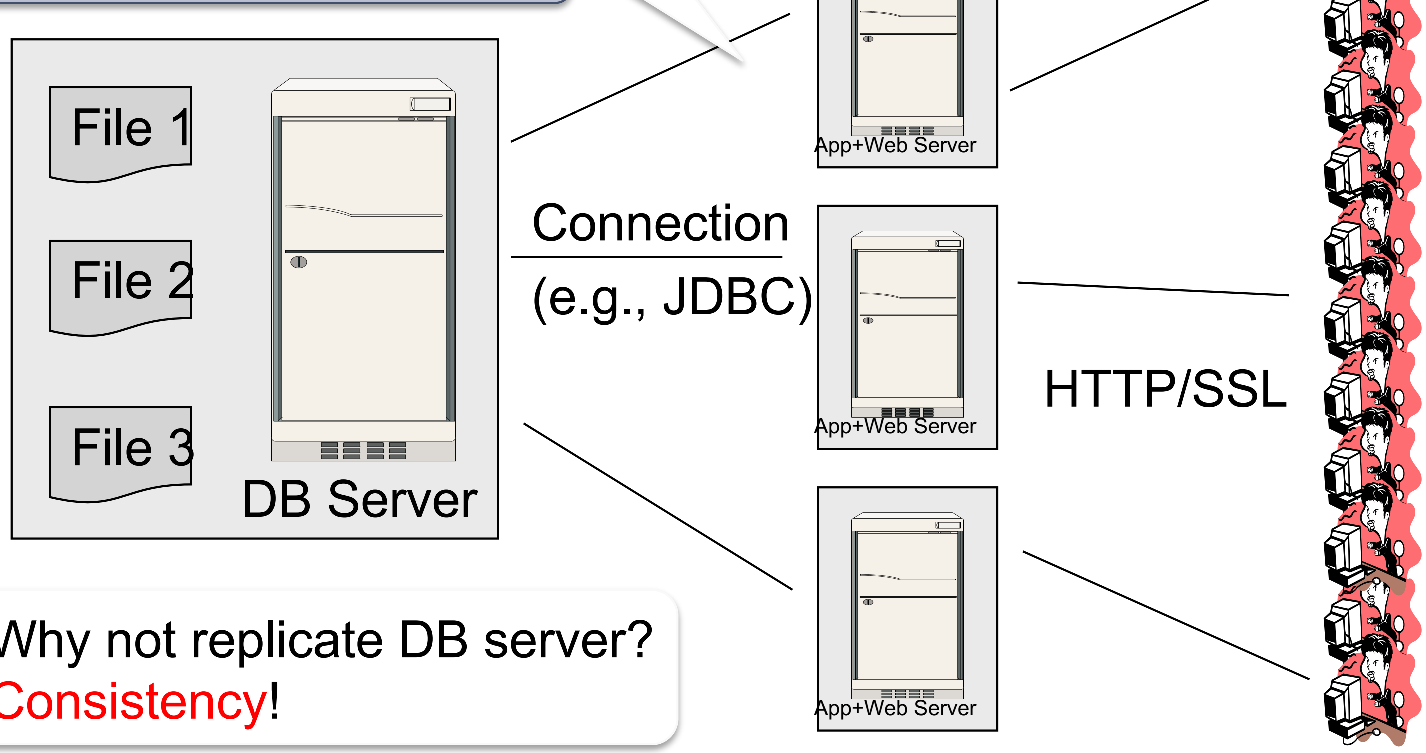


Why not replicate DB server?

Web Apps: 3 T

Replicate App server for scaleup

Web-based applic



Why not replicate DB server?
Consistency!

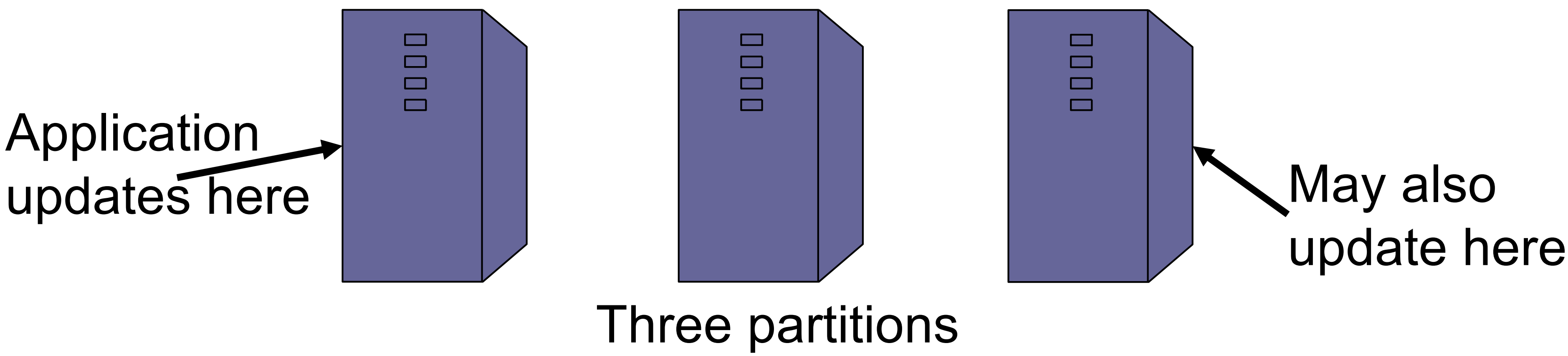
Replicating the Database

- Two basic approaches:
  - Scale up through partitioning – “sharding”
  - Scale up through replication

- Consistency is much harder to enforce

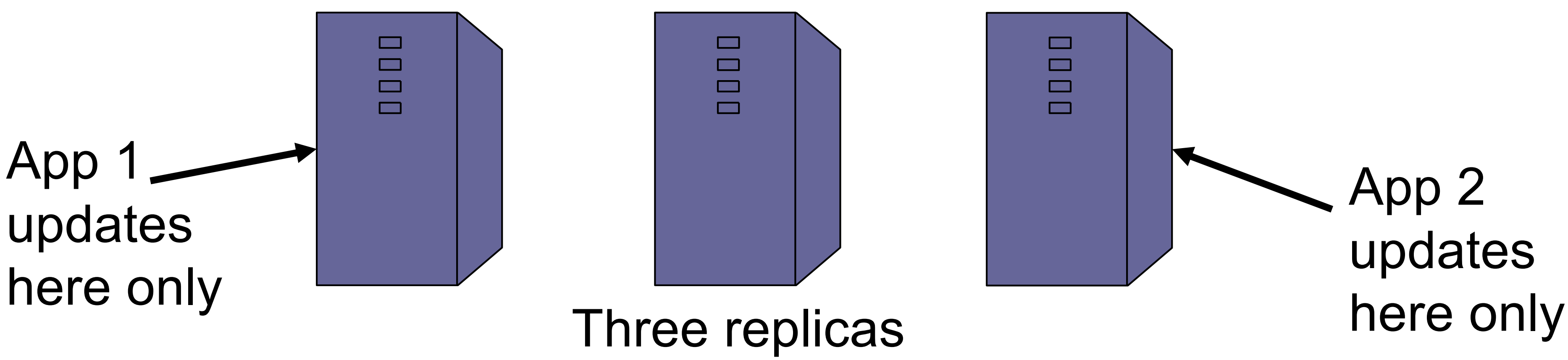
Scale Through Partitioning

- 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. E.g., `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

Chem 1A

- Relational DB
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- NoSQL
  - **B**asic **A**vailability
    - Application must handle partial failures itself
  - **S**oft State
    - DB state can change even without inputs
  - **E**ventually Consistency
    - DB will “eventually” become consistent

- i.e., ACID vs BASE

Data Models

Taxonomy based on data models:

- Key-value stores
  - ☞ e.g., Amazon Dynamo, Voldemort, Memcached
- Extensible Record Stores
  - e.g., HBase, Cassandra, PNUTS
- 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** – w/ hash function
  - No replication: key k is stored at server h(k)
  - Multi way-way replication: e.g., key k stored at h1(k),h2(k),h3(k)

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

Example

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

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

Example

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

- How would you represent the Flights data as key, value pairs?
- Option 1: key=fid, value=entire flight record

Example

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

- 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

Example

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

- 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
  - Store every (key,value) pair on server h(key)
- 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., Amazon Dynamo, Voldemort, Memcached
- Extensible Record Stores
  - ☞ e.g., HBase, Cassandra, PNUTS
- Document stores
  - e.g., SimpleDB, CouchDB, MongoDB

Extensible Record Stores

- Also called wide-column stores
- Based on Google's BigTable
- HBase is an open source implementation of BigTable

- Data model:
  - Variant 1: key = rowID, value = record
  - Variant 2: key = (rowID, columnID), value = field
    - Or multiple columnIDs in the key

- Will not discuss in class

Data Models

Taxonomy based on data models:

- Key-value stores
  - e.g., Amazon Dynamo, Voldemort, Memcached
- Extensible Record Stores
  - e.g., HBase, Cassandra, PNUTS
- 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: *value* to be structured data
  - JSON or Protobuf or XML
  - Called a “document” but it’s just data

We will discuss 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 Algebra

- 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 Types

- Primitive: number, string, Boolean, null
- Object: collection of name-value pairs:
  - `{“name1”: value1, “name2”: value2, …}`
  - “`name`” is also called a “key”
- Array: *ordered* list of values:
  - `[obj1, obj2, obj3, ...]`

Avoid Using Duplicate Keys

The standard allows them, but many implementations don't

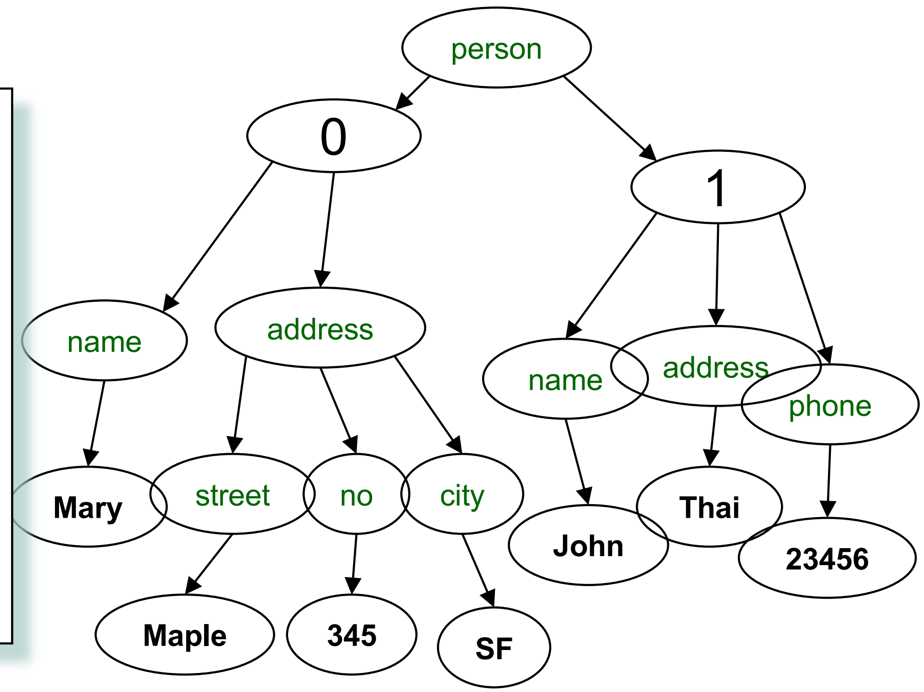
```
{"id":"07",
    "title": "Databases",
    "author": "Garcia-Molina",
    "author": "Ullman",
    "author": "Widom"
}
```

Use an ordered list instead

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

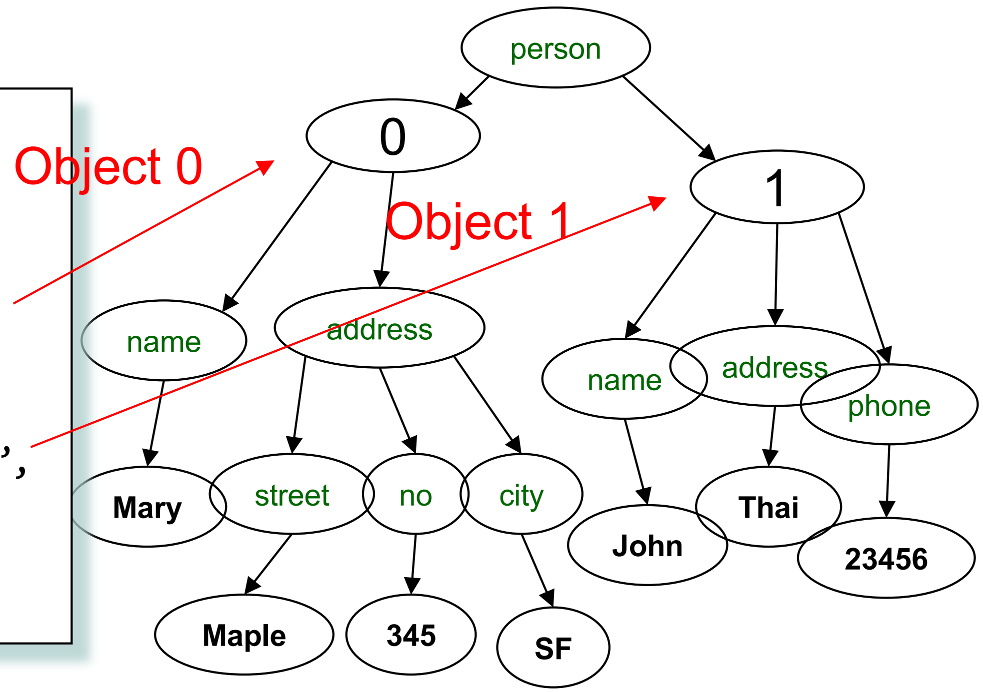
JSON Semantics: a Tree !

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



JSON Semantics: a Tree !

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



Recall: arrays are *ordered* in JSON!

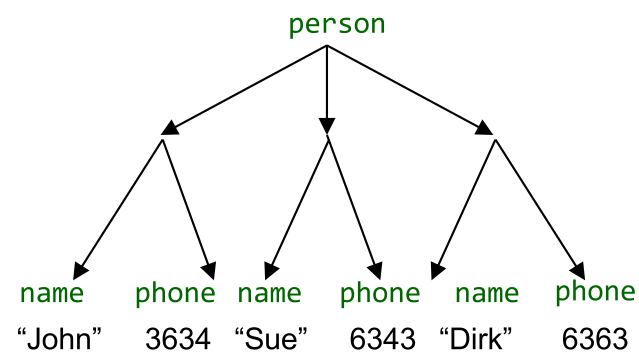
Intro to Semi-structured Data

- 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
- ⇨ JSON is more flexible
  - Schema can change per tuple

Storing JSON in RDBMS

- Using JSON as a data type provided by RDBMSs
  - Declare a column that contains either json or jsonb (binary)
    - `CREATE TABLE people (person json)` [or jsonb for binary]
  - In our previous example, we will have one row per person
    - i.e., a row corresponding to that person's attributes
  - Queries now mix relational and semi-structured syntax
    - ```
      SELECT * FROM people
      WHERE person @> `{"name": "Mary"}';
      ```
- Translate JSON documents into relations
      ```

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 multiple relations based on 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"}
   ]
  }
 ]
}
```

Mapping Relational Data to JSON

Many-many relationships are more difficult to represent

Person

name	phone
John	3634
Sue	6343

Product

prodName	price
Gizmo	19.99
Phone	29.99
Gadget	9.99

Orders

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

Options for the JSON file:

- 3 flat relations: Person,Orders,Product
- Person→Orders→Products products are duplicated
- Product→Orders→Person persons are duplicated

Mapping Semi-structured Data to Relations

- Missing attributes:

```
{"person":
     [{"name":"John", "phone":1234},
      {"name":"Joe"}]
}
```

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	NULL

Mapping Semi-structured Data to Relations

- Repeated attributes

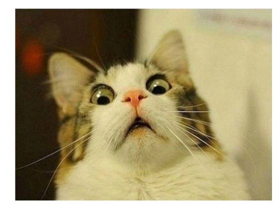
```
{"person":
     [{"name":"John", "phone":1234},
      {"name":"Mary", "phone":[1234,5678]}]
}
```

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456

???



Mapping Semi-structured Data to Relations

- Attributes with different types in different objects

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

Structured name !

- Nested collections
- Heterogeneous collections

- These are difficult to represent in the relational model

Discussion: Why Semi-Structured Data?

- Semi-structured data works well as *data exchange formats*
  - i.e., exchanging data between different apps
  - Examples: XML, JSON, Protobuf (protocol buffers)

- Increasingly, systems use them as a data model for DBs:
  - SQL Server supports for XML-valued relations
  - CouchBase, MongoDB, Snowflake: JSON
  - Dremel (BigQuery): Protobuf

Query Languages for Semi-Structured Data

- XML: XPath, XQuery (see textbook Ch 27)
  - Supported inside many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines
- Protobuf: SQL-ish language (Dremel) used internally by google, and externally in BigQuery
- JSON:
  - CouchBase: N1QL
  - AsterixDB: SQL++ (based on SQL)
  - MongoDB: has a pattern-based language
  - JSONiq: http://www.jsoniq.org/

Semistructured Data Model

- Several file formats: JSON, protobuf, XML
- Data model = Tree

- Query language take non first normal form into account as we will see
  - Various “extra” constructs introduced as a result
    - Nesting & Unnesting, strict aggregates, splitting