# Discussion 14

## No SQL

# 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**

# Two Classes of Relational Database Apps

| Property | OLTP | OLAP |
|---|---|---|
| Main read pattern | Small number of records per query, fetched by key | Aggregate over large number of records |
| Main write pattern | Random-access, low-latency writes from user input | Bulk import or event stream |
| Primarily used by | End user/customer, via web application | Internal analysis, for decision support |
| What data represents | Latest state of data (current point in time) | History of events that happend over time |
| Dataset size | Gigabytes to terabytes | Terabytes to petabytes |

# Motivation

- Originally motivated by Web 2.0 applications
- Needed: very large scale OLTP workloads
- Give up on consistency, give up OLAP
- NoSQL: reduce functionality
  - Simpler data model
  - Very restricted updates

# Scaling the Database

- Two basic approaches:
  - Scale up through partitioning – “sharding”
    - Partition the database across machines in a cluster
    - Can increase throughput
    - Easy for writes but reads become expensive!
  - Scale up through replication
    - Create multiple copies of each database partition. Spread queries across these replicas
    - Can increase throughput and lower latency, improve fault-tolerance
    - Easy for reads but writes become expensive!

- Consistency is much harder to enforce

# 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

- NoSQL: simplified data model
  - Given up on functionality
  - Application must now handle joins and consistency

# ACID vs BASE

- 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)
- **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)

# 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

# Document stores: 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

# 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

```
{  "book": [
     {"id":"01",
      "language": "Java",
      "author": "H. Javeson",
      "year": 2015
     },
     {"id":"07",
      "language": "C++",
      "edition": "second"
      "author": "E. Sepp",
      "price": 22.25
     }
   ]
}
```

# JSON Types

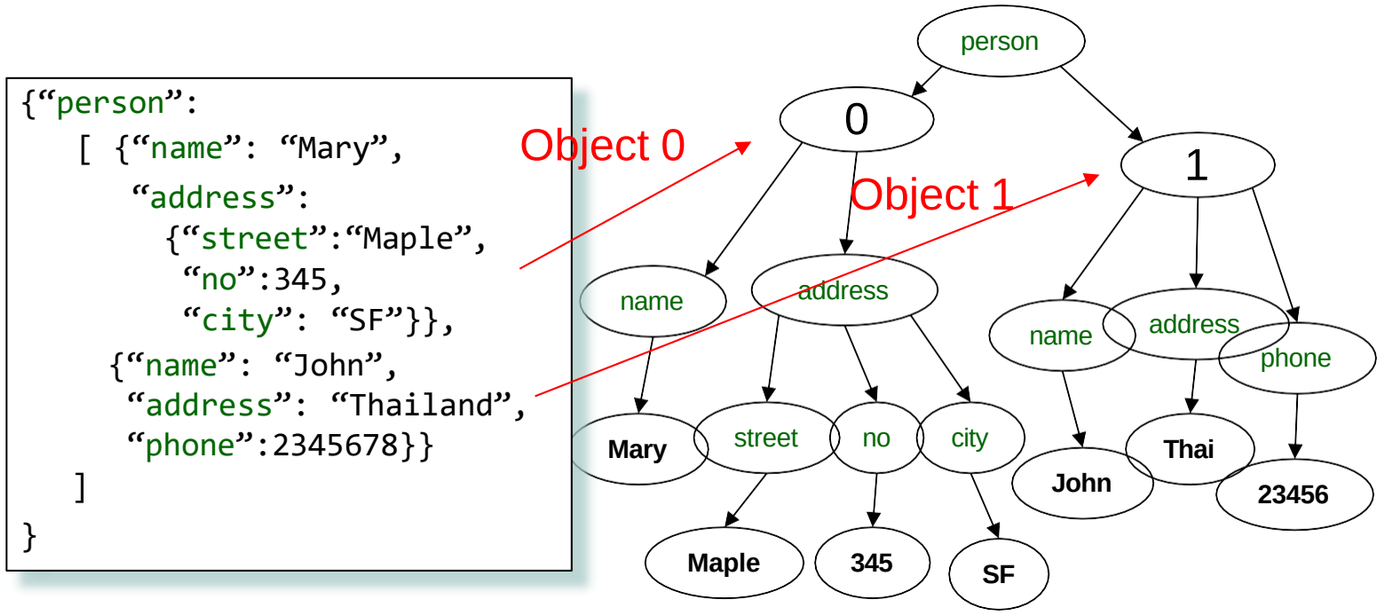
- Primitive: number, string, Boolean, null

- Object: collection of name-value pairs:
  - {“name1”: value1, “name2”: value2, …}
  - “name” is also called a “key”

- Array: *ordered* list of values:
  - [obj1, obj2, obj3, ...]

# 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!

# Relational -> JSON

Single Table:

- key = table name
- value: array of objectives
- objective <-> row in table, with key = table attribute, value = tuple element

One-to-many relationship:

- nested structure
- linked rows in the 2nd table are embedded or “inlined” into the objects representing the row of the 1st table

# Relational -> JSON

Many-to-many relationship

First option: represent each relation as a flat JSON array, each element is an object representing a row

- no redundant data, but relies on the application to join the tables to get related data

The second and third options: key off of the one of the primary-keyed tables, inline the contents of the relationship table and the last table within each object

- the elements of the last table will be duplicated for each linked element of the first table
- application does not need to perform any joins to get the full record
- updates to the document store will need to handle updating all duplicated rows

# 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

# 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

# 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
      ```

# 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